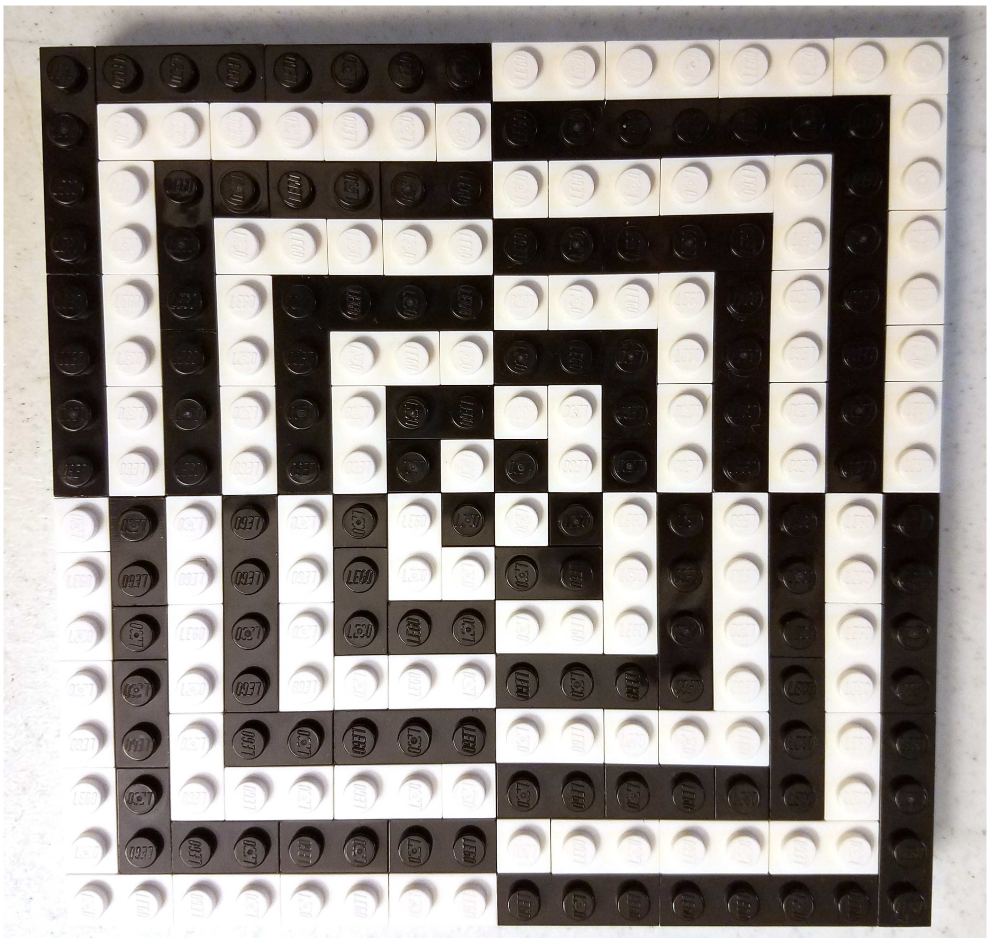
An Active Introduction to Discrete Mathematics and Algorithms



Charles A. Cusack
cusack@hope.edu

David A. Santos

Version 2.6.3
March 30, 2018

History

- An Active Introduction to Discrete Mathematics and Algorithms, 2016, Charles A. Cusack. Minor revisions. Algorithm Analysis chapter had a few additions.
- An Active Introduction to Discrete Mathematics and Algorithms, 2015, Charles A. Cusack. Minor revisions. Algorithm Analysis chapter had major revisions.
- An Active Introduction to Discrete Mathematics and Algorithms, 2014, Charles A. Cusack. This is a significant revision of the 2013 version (thus the slight change in title).
- An Introduction to Discrete Mathematics and Algorithms, 2013, Charles A. Cusack. This document draws some content from each of the following.
  - Discrete Mathematics Notes, 2008, David A. Santos.
  - More Discrete Mathematics, 2007, David A. Santos.
  - Number Theory for Mathematical Contests, 2007, David A. Santos.
  - Linear Algebra Notes, 2008, David A. Santos.
  - Precalculus, An Honours Course, 2008, David Santos.

  These documents are all available from http://www.opensourcemath.org/books/santos/, but the site appears not to be consistently available.

About the cover

The image on the cover is an example of mathematical art using Lego bricks. It shows 9 different Latin squares of order 16. Go to https://www.instagram.com/ferzle/ to see more pictures of Dr. Cusack's Lego art.

Contents

Preface

This book is an attempt to present some of the most important discrete mathematics concepts to computer science students in the context of algorithms. I wrote it for use as a textbook for half of a course on discrete mathematics and algorithms.

Some of the material is drawn from several open-source books by David Santos. Other material is from handouts I have written and used over the years. I have extensively edited the material from both sources, both for clarity and to emphasize the connections between the material and algorithms where possible. I have also added a significant amount of new material. The format of the material is also significantly different than it was in the original sources.

I should mention that I never met David Santos, who apparently died in 2011. I stumbled upon his books in the summer of 2013 when I was searching for a discrete mathematics book to use in a new course. When I discovered that I could adapt his material for my own use, I decided to do so. Since clearly he has no knowledge of this book, he bears no responsibility for any of the edited content. Any errors or omissions are therefore mine.

This is still a work in progress, so I appreciate any feedback you have. Please send any typos, formatting errors, other errors, suggestions, etc., to cusack@hope.edu.

I would like to thank the following people for submitting feedback/errata (listed in no particular order): Dan Zingaro, Mike Jipping, Steve Ratering, Victoria Gonda, Nathan Vance, Cole Watson, Kalli Crandell, John Dood, Coty Franklin, Kyle Magnuson, Katie Brudos, Jonathan Senning, Matthew DeJongh, Julian Payne, Josiah Brouwer, and probably several others I forgot to mention.

Charles A. Cusack
July, 2014

How to use this book

As the title of the book indicates, this is not a book that is just to be read. It was written so that the reader interacts with the material. If you attempt to just read what is written and take no part in the exercises that are embedded throughout, you will likely get very little out of it. Learning needs to be active, not passive. The more active you are as you 'read' the book, the more you will get out of it. That will translate to better learning. And it will also translate to a higher grade. So whether you are motivated by learning (which is my hope) or merely by getting a certain grade, your path will be the same–use this book as described below.

The content is presented in the following manner. First, concepts and definitions are given–generally one at a time. Then one or more examples that illustrate the concept/definition will be given. After that you will find one or more exercises of various kinds. This is where this book differs from most. Instead of piling on more examples that you merely read and *think* you understand, you will be asked to solve some for yourself so that you can be more confident that you really *do* understand.

Some of the exercises are just called *Exercises*. They are very similar to the examples, except that you have to provide the solution. There are also *Fill in the details* which provide part of the solution, but ask you to provide some of the details. The point of these is to help you think about some of the finer details that you might otherwise miss. There are also *Questions* of various kinds that get you thinking about the concepts. Finally, there are *Evaluate* exercises. These ask you to look at solutions written by others and determine whether or not they are correct. More precisely, your goal is to try to find as many errors in the solutions as you can. Usually there will be one or more errors in each solution, but occasionally a correct solution will be given, so pay careful attention to every detail. The point of these exercises is to help you see mistakes before you make them. Many of these exercises are based on solutions from previous students, so they often represent the common mistakes students make. Hopefully if you see someone else make these mistakes, you will be less likely to make them yourself.

The point of the exercises is to get you thinking about and interacting with the material. As you encounter these, you should write your solution in the space provided. After you have written your solution, you should check your answer with the solution provided. You will get the most out of them if you first do your best to give a complete solution on your own, and then *always* check your solution with the one provided to make sure you did it correctly. If yours is significantly different, make sure you determine whether or not the differences are just a matter of choice or if there is something wrong with your solution.

If you get stuck on an exercise, you should re-read the previous material (definitions, examples, etc.) and see if that helps. Then give it a little more thought. For *Fill in the details* questions, sometimes reading what is past a blank will help you figure out what to put there. If you get *really* stuck on an exercise, look up the solution and make sure you fully understand it. But don't jump to the solution too quickly or too often without giving an honest attempt at solving the exercise yourself. When you do end up looking up a solution, you should always try to rewrite it in the space provided *in your own words*. You should not just copy it word for word. You won't learn as much if you do that. Instead, do your best to fully understand the solution. Then, without looking at the solution, try to re-solve the problem and write your solution in the space provided. Then check the solution again to make sure you got it right.

It is highly recommended that you act as your own grader when you check your solutions. If your solution is correct, put a big check mark in the margin. If there are just a few errors, use a different colored writing utensil to mark and fix your errors. If your solution is way off, cross it out (just put a big 'X' through it) and write out your second attempt, using a separate sheet of paper if necessary. If you couldn't get very far without reading the solution, you should somehow indicate that. So that you can track your errors, I highly recommend crossing out incorrect solutions (or portions of solutions) instead of erasing them. Doing this will also allow you to look back and determine how well you did as you were working through each chapter. It may also help you determine how to spend your time as you study for exams. This whole process will help you become better at evaluating your own work. This is important because you should be confident in your answers, but only when they are correct. Grading yourself will help you gain confidence when you are correct and help you quickly realize when you are not correct so that you do not become confident about the wrong things. Another reason that grading your solutions is important is so that when you go back to re-read any portion of the book, you will know whether or not what you wrote was correct.

It is important that you read the solutions to the exercises after you attempt them, even if you think your solution is correct. The solutions often provide further insight into the material and should be regarded as part of any reading assignment given.

Make sure you read carefully. When you come upon an *Evaluate* exercise, do not mistake it for an example. Doing so might lead you down the wrong path. Never consider the content of an *Evaluate* exercise to be correct unless you have verified with the solution that it is really correct. To be safe, when re-reading, always assume that the *Evaluate* exercises are incorrect, and never use them as a model for your own problem solving. To help you, we have tried to differentiate these from other example and exercise types by using a different font.

Finally, there is an expectation that you are able to solve every exercise on your own. (Note that I am talking about the exercises embedded into the chapters, not the homework problems at the end of each chapter.) If there are exercises that you are unable to complete, you need to get them cleared up immediately. This might mean asking about them in class, going to see the professor or a teaching assistant, and/or going to a help center/tutor. Whatever it takes, make sure you have a clear understanding of how to solve all of them.

Chapter 1

Motivation

The purpose of a discrete mathematics course in the computer science curriculum is to give students a foundation in some of the mathematical concepts that are foundational to computer science. By "foundational," we mean both that the field of computer science was built upon (some of) them and that they are used to varying degrees in the study of the more advanced topics in computer science.

Computer science students sometimes complain about taking a discrete mathematics course. They do not understand the relevance of the material to the rest of the computer science curriculum or to their future career. This can lead to lack of motivation. They also perceive the material to be difficult.

To be honest, some of the topics are difficult. But the majority of the material is very accessible to most students. One problem is that learning discrete mathematics takes effort, and when something doesn't sink in instantly, some students give up too quickly. The perceived difficulty together with a lack of motivation can lead to lack of effort, which almost always leads to failure. Even when students expend effort to learn, they can let their perceptions get the best of them. If someone believes something is hard or that they can't do it, it often leads to self-fulfilling prophecy. This is perhaps human nature. On the other hand, if someone believes that they can learn the material and solve the problems, chances are they will. The bottom line is that a positive attitude can go a long way.

This book was written in order to ensure that the student *has to* expend effort while reading it. The idea is that if you are allowed to simply read but not required to interact with the material, you can easily read a chapter and get nothing out. For instance, your brain can go on 'autopilot' when something doesn't sink in and you might get nothing out of the remainder of your time reading. By requiring you to solve problems and answer questions as you read, your brain is forced to stay engaged with the material. In addition, when you incorrectly solve a problem, you know immediately, giving you a chance to figure out what the mistake was and correct it before moving on to the next topic. When you correctly solve a problem, your confidence increases. We strongly believe that this feature will go a long way to help you more quickly and thoroughly learn the material, assuming you use the book as instructed.

What about the problem of relevance? In other words, what is the connection between discrete mathematics and other computer science topics? There are several reasons that this connection is unclear to students. First, we don't always do a very good job of making the connection clear. We teach a certain set of topics because it is the set of topics that has always been taught in such a course. We don't always think about the connection ourselves, and it is easy to forget that this connection is incredibly important to students. Without it, students can suffer from a lack of motivation to learn the material.

The second reason the connection is unclear is because one of the goals of such a course is simply to help students to be able to think mathematically. As they continue in their education and career, they will most certainly use some of the concepts they learn, yet they may be totally unaware of the fact that some of their thoughts and ideas are based on what they learned in a discrete mathematics course. Thus, although the students gain a benefit from the course, it is essentially impossible to convince them of this during the course.

The third reason that the connection is unclear is that given the time constraints, it is impossible to provide all of the foundational mathematics that is relevant to the advanced computer science courses *and* make the connection to those advanced topics clear. Making these connections would require an in-depth discussions of the advanced topics. The connections are generally made, either implicitly or explicitly, in the courses in which the material is needed.

This book attempts to address this problem by making connections to one set of advanced topics–the design and analysis of algorithms. This is an ideal application of the discrete mathematics topics since many of them are used in the design and analysis of algorithms. We also do not have to go out of our way too far to provide the necessary background, as we would if we attempted to make connections to topics such as networking, operating systems, architecture, artificial intelligence, database, or any number of other advanced topics. As already mentioned, the necessary connections to those topics will be made when you take courses that focus on those topics.

The goal of the rest of this chapter is to further motivate you to want to learn the topics that will be presented in this book. We hope that after reading it you *will* be more motivated. For some students, the topics are interesting enough on their own, whether or not they can be applied elsewhere. For others, this is not the case. One way or another, you must find motivation to learn this material.

1.1 Some Problems

In this section we present a number of problems for you to attempt to solve. You should make an honest attempt to solve each. We suspect that most readers will be able to solve at most a few of them, and even then will probably not use the most straightforward techniques. On the other hand, after you have finished this book you should be able to solve most, if not all of them, with little difficulty.

There are two main reasons we present these problems to you now. First, we hope they help you gauge your learning. That is, we hope that you *do* experience difficulty trying to to solve them now, but that when you revisit them later, they will seem much easier. Second, we hope they provide some motivation for you to learn the content. Although all of these problems may not interest you, we hope that you are intrigued by at least some of them.

Problem A: The following algorithm supposedly computes the sum of the first n integers. Does it work properly? If it does not work, explain the problem and fix it.

```
sum1ToN(int n) {
    return n + sum1ToN(n-1);
}
```

Problem B: The Mega Millions lottery involves picking five different numbers from 1 to 56, and one number from 1 to 46. I purchased a ticket last week and was surprised when none of my six numbers matched. Should I have been surprised? What are the chances that a randomly selected ticket will match none of the numbers?

Problem C: I programmed an algorithm recently to solve an interesting problem. The input is an array of size n. When $n = 1$, it took 1 second to run. When $n = 2$, it took 7 seconds. When $n = 3$, it took 19 seconds. When $n = 4$, it took 43 seconds. Assume this pattern continues.

(a) How large of an array can I run the algorithm on in less than 24 hours?

(b) How large can n be if I can wait a year for the answer?

Problem D: Is the following a reasonable implementation of the QUICKSORT algorithms? In other words, is it correct, and is it efficient? (Notice that the only difference between this and the standard algorithm is that this one is implemented on a `LinkedList` rather than an `array`.)

```
Quicksort(LinkedList A,int l,int r) {
   if(r > l) {
      int p = RPartition(A,l,r);
      Quicksort(A,l,p-1);
      Quicksort(A,p+1,r);
   }
}

int RPartition(LinkedList A,int l,int r) {
    int piv=l+(rand()%(r-l+1));
    swap(A,l,piv);
    int i = l+1;
    int j = r;
    while (1) {
        while (A.get(i) <= A.get(l) && i<r)
            i++;
        while (A.get(j) >= A.get(l) && j>l)
            j--;
        if (i >= j) {
            swap(A,j,l);
            return j;
        } else {
            swap(A,i,j);
        }
    }
}

void swap(LinkedList A, index i, index j) {
    int temp = A.get(i);
    A.set(i,A.get(j));
    A.set(j,temp);
}
```

Problem E: I have an algorithm that takes two inputs, n and m. The algorithm treats n differently when it is less than zero, between zero and 10, and greater than 10. It treats m differently based on whether or not it is even. I want to write some test code to make sure the algorithm works properly for all possible inputs. What pairs (n, m) should I test? Do these tests guarantee correctness? Explain.

Problem F: Consider the `stoogeSort` algorithm given here:

```
void stoogeSort(int[] A,int L,int R){
    if(R<=L) { // Array has at most one element so it is sorted
        return;
    }
    if(A[R]<A[L]) {
        int temp = A[L];  // Swap first and last element
        A[L] = A[R];      // if they are out of order
        A[R] = temp;
    }
    if(R-L>1){ // If the list has at least 3 elements
        int third=(R-L+1)/3;
        stoogeSort(A,L,R-third); // Sort first two-thirds
        stoogeSort(A,L+third,R); // Sort last two-thirds
        stoogeSort(A,L,R-third); // Sort first two-thirds again
    }
}
```

(a) Does `stoogeSort` correctly sort an array of integers?

(b) Is `stoogeSort` a good sorting algorithm? Specifically, how long does it take, and how does it compare to other sorting algorithms?

Problem G: In how many ways may we write the number 19 as the sum of three positive integer summands? Here order counts, so, for example, $1 + 17 + 1$ is to be regarded different from $17 + 1 + 1$.

Problem H: Can the following code be simplified? If so, give equivalent code that is as simple as possible.

```
if ((!x.size() <=0  && x.get(0) != 11) || x.size() > 0)
{
    if(!(x.get(0)==11 && (x.size() > 13 || x.size() < 13))
        && (x.size() > 0 || x.size() == 13))
    {
        //do something
    }
}
```

Problem I: A cryptosystem was recently proposed. One of the parameters of the cryptosystem is a nonnegative integer n, the meaning of which is unimportant here. What is important is that someone has proven that the system is insecure for a given n if there is more than one integer m such that $2 \cdot m \leq n \leq 2 \cdot (m + 1)$.

(a) For what value(s) of n, if any, can you prove or disprove that there is more than one integer m such that $2 \cdot m \leq n \leq 2 \cdot (m + 1)$?

(b) Given your answer to (a), does this prove that the cryptosystem is either secure or insecure? Explain.

Problem J: A certain algorithm takes a positive integer, n, as input. The first thing the algorithm does is set $n = n \bmod 5$. It then uses the value of n to do further computations. One friend claims that you can fully test the algorithm using just the inputs 1, 2, 3, 4, and 5. Another friend claims that the inputs 29, 17, 38, 55, and 6 will work just as well. A third friend responds with "then why not just use 50, 55, 60, 65, and 70? Those should work just as well as your stupid lists." A fourth friend claims that you need many more test cases to be certain. A fifth friend says that you can never be certain no matter how many test cases you use. Which friend or friends is correct? Explain.

Problem K: Write an algorithm to swap two integers without using any extra storage. (That is, you can't use any temporary variables.)

Problem L: Recall the *Fibonacci sequence*, defined by the recurrence relation

$$f_n = \begin{cases} 0 & \text{if } n{=}0 \\ 1 & \text{if } n{=}1 \\ f_{n-1} + f_{n-2} & \text{if } n > 1. \end{cases}$$

So $f_2 = 1$, $f_3 = 2$, $f_4 = 3$, $f_5 = 5$, $f_6 = 8$, etc.

(a) One friend claims that the following algorithm is an elegant and efficient way to compute f_n.

```
int Fibonacci(int n) {
    if(n <= 1) {
        return(n);
    } else {
        return(Fibonacci(n-1)+Fibonacci(n-2));
    }
}
```

Is he right? Explain.

(b) Another friend claims that he has an algorithm that computes f_n that takes constant time–that is, no matter how large n is, it always takes the same amount of time to computer f_n. Is it possible that he has such an algorithm? Explain.

Problem M: You are at a party with some friends and one of them claims "I just did a quick count, and it turns out that at this party, there are an odd number of people who have shaken hands with an odd number of other people." Can you prove or disprove that this friend is correct?

Problem N: You need to settle an argument between your boss (who can fire you) and your professor (who can fail you). They are trying to decide who to invite to the Young Accountants Volleyball League. They want to invite freshmen who are studying accounting and are over 6 feet tall. They have a list of everyone they could potentially invite.

1. Your boss says they should make a list of all freshmen, a list of all accounting majors, and a list of everyone over 6 feet tall. They should then combine the lists (removing duplicates) and invite those on the combined list.
2. Your professor says they should make a list of everyone who is not a freshman, a list of anyone who does not do accounting, and a list of everyone who is 6 feet tall or less. They should make a fourth list that contains everyone who is on all three of the prior lists. Finally, they should remove from the original list everyone on this fourth list, and invite the remaining students.

Who is correct? Explain.

Chapter 2

Proof Methods

The ability to write proofs is important to computer scientists for a variety of reasons. Proofs are particularly relevant to the study of algorithms. When you write an algorithm it is important that the algorithm performs as expected, both in terms of producing the correct answer and doing so quickly. That is, proofs are necessary in *algorithm correctness* and *algorithm analysis*.

In this chapter we will introduce you to the basics of mathematical proofs. Along the way we will review some mathematical concepts/definitions you have probably already seen, and introduce you to some new ones that we will find useful as we proceed. We will continue to write proofs and learn more advanced proof techniques as the book continues.

2.1 Direct Proofs

A *direct proof* is one that follows from the definitions. Facts previously learned help many a time when writing a direct proof. We will begin by seeing some direct proofs about something you should already be very familiar with: even and odd integers.

Definition 2.1. *Recall that*

- *an* **even integer** *is one of the form $2k$, where k is an integer.*
- *an* **odd integer** *is one of the form $2k + 1$ where k is an integer.*

Example 2.2. Use the definition of even to prove that the sum of two even integers is even.

Proof: If x and y are even, then $x = 2a$ and $y = 2b$ for some integers a and b. Then $x + y = 2a + 2b = 2(a + b)$, which is even since $a + b$ is an integer. □

Example 2.3. Use the definitions of even and odd to prove that the sum of an even integer and an odd integer is odd.

Proof: Let a be an even integer and b be an odd integer. Then $a = 2f$ and $b = 2g + 1$ for some integers f and g. Then $a + b = 2f + (2g + 1) = 2(f + g) + 1$. Since $f + g$ is an integer, $a + b$ is an odd integer. □

Note: *The next example is the first of many* **Fill in the details** *exercises in which* **you** *need to supply some of the details. After you have filled in the blanks, compare your answers with the solutions. The answers are given with semicolons (;) separating the blanks.*

⋆**Fill in the details 2.4.** Use the definitions of even and odd to prove that the sum of two odd integers is even.

> **Proof:** If x and y are odd, then $x = 2c + 1$ and $y =$ __________ for some integers c and d. Then $x + y = 2c + 1 + 2d + 1 = 2(c + d + 1)$. Now __________ is an integer, so $2(c + d + 1)$ is an __________ integer. □

Note: *Did you notice the ⋆ in the heading of the previous example? This indicates that a solution is provided. If you are reading the PDF file, clicking on the ⋆ will take you to the solution. Clicking on the number in the solution will take you back.*

Example 2.5. Use the definitions of even and odd to prove that the product of two odd integers is odd.

> **Proof:** Let a and b be odd integers. Then $a = 2l + 1$ and $b = 2m + 1$ for some integers l and m. Then $a \cdot b = (2l+1)(2m+1) = 4ml+2l+2m+1 = 2(2ml+l+m)+1$ which is odd since $2ml + m + l$ is an integer. □

⋆**Fill in the details 2.6.** Use the definitions of even and odd to prove that the product of an even integer and an odd integer is even.

> **Proof:** Let a be an even integer and b be an odd integer. Then $a =$ __________ and $b =$ __________ for ____________________. Given that, we can see that $a \cdot b = (2n)(2o + 1) =$ ___________________________. Since __________ is an integer, $a \cdot b$ is _____________. □

These examples may seem somewhat ridiculous since they are proving such obvious facts. However, keep in mind that our goal is to learn techniques for writing proofs. As we proceed the proofs will become more complicated, but we will continue to follow the same basic techniques we are using here. In other words, the fact that we are proving facts about even and odd integers is not at all important. What is important are the techniques we are learning in the process.

You may be asking yourself "why are we wasting our time proving such obvious results"? If so, ask yourself this: Would you rather be asked to prove more complicated things right away?

Think about how you learned to read and write. You started by reading books that only had a few simple words. As you progressed, the books and the words in them got longer. The vocabulary increased. You encountered increasingly complex sentence and paragraph structures. The same is true when you learned to write. You began by writing the letters of the alphabet. Then you learned to write words, followed by sentences, paragraphs, and eventually essays.

Learning to read and write proofs follows the same procedure. In order to know how to write correct proofs you first need to see some examples of them. But you need to go beyond just seeing them–you need to *understand* them. That is the goal of examples like the previous one. Your brain needs to be engaged with the material as you work through the book. You *must* work through all of the examples in order to get the most out of this book.

Note: *Next you will see the first of many* **Exercises**. *These give you an opportunity to solve a problem from start to finish and then check your answer with the solution provided. It is important that you try each of these on your own before looking at the solution. You will not get as much out of the book if you skip these or jump straight to the answer without trying them yourself.*

⋆**Exercise 2.7.** Use the definition of even to prove that the product of two even integers is even.
Proof:

Note: *The next example is an* **Evaluate** *example. These examples give a problem and then provide one or more solutions to the problem based on previous student solutions. Your job is to evaluate each solution by finding any mistakes. Mistakes include not only incorrect algebra and logic, but also unclear presentation, skipped steps, incorrect assumptions, over-simplification, etc. When you come across these examples you should write down every error you can find. Once you are pretty sure you know all of the problems (if there are any), compare your evaluation to the one given in the solutions.*

⋆**Evaluate 2.8.** Evaluate the following proof that supposedly uses the definition of odd to prove that the product of two odd integers is odd.

> Proof: By definition of odd numbers, let a be an odd integer $2n+1$ let b be an odd integer $2q+1$. Then $(2n+1)(2q+1) = 4nq+2n+1 = 2(2nq+1)+1$. Since $2nq+1$ is an integer, $2(2nq+1)+1$ is an odd integer by definition of odd. □

Evaluation ____

____

____

Sometimes students get frustrated because they think that too many details are required in a proof. Why are mathematicians such sticklers on the details? The next example is the first of many that will try to demonstrate why the seemingly little details matter.

Note: *The* **Question** *examples are similar to the* **Evaluate** *ones except that they ask a specific question. Write down your answer in the space provided and then compare your answer with the one in the solutions.*

⋆**Question 2.9.** What is wrong with the following "proof" that the sum of an even and an odd number is even?

> Proof: Let $a = 2n$ be an even integer and $b = 2m+1$ be an odd integer. Then $a+b = 2n+2m+1 = 2(n+m+1/2)$. Since we wrote $a+b$ as a multiple of 2, it is even. Therefore the sum of an even and an odd number is even. □

Answer ____

____

We will find the following definitions useful throughout the book.

Definition 2.10. *Let b and a be integers with $a \neq 0$. We say that b is* **divisible by** *a if there exists an integer c such that $b = ac$. If b is divisible by a, we also say that b is a* **multiple** *of a, a is a* **factor** *or* **divisor** *of b, and that a* **divides** *b, written as $a|b$. If a does not divide b, we write $a \nmid b$.*

Example 2.11. Since $6 = 2 \cdot 3$, $2|6$, and $3|6$. But $4 \nmid 6$ since we cannot write $6 = 4 \cdot c$ for any integer c.

Example 2.12. Prove that the product of two even integers is divisible by 4.

Proof: Let $2h$ and $2k$ be even integers. Then $(2h)(2k) = 4(hk)$. Since hk is an integer, $4(hk)$ is divisible by 4. □

⋆**Fill in the details 2.13.** Prove that if x is an integer and 7 divides $3x + 2$, then 7 also divides $15x^2 - 11x - 14$.

Proof: Since 7 divides $3x + 2$, we know that $3x + 2 = 7a$, where a is

________________. Notice that

$$\begin{aligned} 15x^2 - 11x - 14 &= (\underline{\hspace{3cm}})(\underline{\hspace{3cm}}) \\ &= \underline{\hspace{2cm}}a(5x - 7). \end{aligned}$$

Therefore __________________________. □

Example 2.14. Let a and b be integers such that $a|b$ and $b|a$. Prove that either $a = b$ or $a = -b$.

Proof: If $a|b$, we can write $b = ac$ for some integer c. Similarly, if $b|a$, we can write $a = bd$ for some integer d. Then we can write $b = ac = (bd)c$. Dividing both sides by b (which is legal, since $b|a$ implies $b \neq 0$), we can see that $cd = 1$. Since c and d are integers, we know that either $c = d = 1$ or $c = d = -1$. In the first case, we have that $a = b$, and in the second case, we have that $a = -b$. □

⋆**Evaluate 2.15.** Prove that if n is an integer, then $n^3 - n$ is divisible by 6.

Proof: We have $n^3 - n = (n-1)n(n+1)$, the product of three consecutive integers. Among three consecutive integers at least one is even and exactly one is divisible by 3. Since 2 and 3 do not have common factors, 6 divides the quantity $(n-1)n(n+1)$, and so $n^3 - n$ is divisible by 6. □

Evaluation ____

Definition 2.16. *A positive integer $p > 1$ is* **prime** *if its only positive factors are* 1 *and* p. *A positive integer $c > 1$ which is not prime is said to be* **composite**.

⋆**Evaluate 2.17.** Prove or disprove that if a is a positive even integer, then it is composite.

Proof: Let a be an even number. By definition of even, $a = 2k$ for some integer k. Since $a > 0$, clearly $k > 0$. Since a has at least two factors, 2 and k, a is composite. □

Evaluation ____

Note: *Notice that according to the definitions given above,* 1 *is neither prime nor composite. This is one of the many things that makes* 1 *special.*

⋆**Exercise 2.18.** Prove that 2 is the only even prime number.
(Hint: Assume a is an even number other than 2 and prove that a is composite.)

Proof ____

⋆**Question 2.19.** Did you notice that the proof in the solution to the previous exercise (you read it, right?) did not consider the case of 0 or negative even numbers. Was that O.K.? Explain why or why not.

Answer ____

Definition 2.20. *For a non-negative integer n, the quantity $n!$ (read* **"n factorial"***) is defined as follows. $0! = 1$ and if $n > 0$ then $n!$ is the product of all the integers from 1 to n inclusive:*

$$n! = 1 \cdot 2 \cdots n.$$

Example 2.21. $3! = 1 \cdot 2 \cdot 3 = 6$, and $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$.

Example 2.22. Prove that if $n > 0$, then $n! \leq n^n$.

Proof: Since $1 \leq n$, $2 \leq n$, $\cdots$, and $(n-1) \leq n$, it is easy to see that

$$\begin{aligned} n! &= 1 \cdot 2 \cdot 3 \cdots n \\ &\leq n \cdot n \cdot n \cdots n \\ &= n^n. \end{aligned}$$

$\square$

⋆**Evaluate 2.23.** Prove that if $n > 4$ is composite, then n divides $(n-1)!$.

Proof: Since n is composite, $n = ab$ for some integers $1 < a < n-1$ and $1 < b < n-1$. By definition of factorial, $a|(n-1)!$ and $b|(n-1)!$. Therefore $n = ab$ divides $(n-1)!$ $\square$

Evaluation ____

Since the previous proof wasn't correct, let's fix it.

Example 2.24. Prove that if $n > 4$ is composite, then n divides $(n-1)!$.

Proof: If n is not a perfect square, then we can write $n = ab$ for some integers a and b with $1 < a < b < n-1$. Thus, $(n-1)! = 1 \cdots a \cdots b \cdots (n-1)$. Since a and b are distinct numbers on the factor list, $n = ab$ is clearly a factor of $(n-1)!$.

If n is a perfect square, then $n = a^2$ for some integer $2 < a < n-1$. Since $a > 2$, $2a < a^2 = n$. Thus, $2a < n$, so $(n-1)! = 1 \cdots a \cdots 2a \cdots (n-1)$. Then $a(2a) = 2n$ is a factor of $(n-1)!$, which means that n is as well. $\square$

⋆**Question 2.25.** Why was it O.K. to assume $1 < a < b < n-1$ in the previous proof?

Answer ____

⋆**Question 2.26.** In the second part of the previous proof, why could we say that $a > 2$?

Answer ____

Example 2.27. Prove the *Arithmetic Mean-Geometric Mean Inequality*, which states that for all non-negative real numbers x and y,

$$\sqrt{xy} \leq \frac{x+y}{2}.$$

Proof: Since x and y are non-negative, $\sqrt{x}$ and $\sqrt{y}$ are real numbers, so $\sqrt{x}-\sqrt{y}$ is a real number. Since the square of any real number is greater than or equal to 0 we have

$$(\sqrt{x}-\sqrt{y})^2 \geq 0.$$

Expanding (recall the FOIL method?) we get

$$x - 2\sqrt{xy} + y \geq 0.$$

Adding $2\sqrt{xy}$ to both sides and dividing by 2, we get

$$\frac{x+y}{2} \geq \sqrt{xy},$$

yielding the result. □

The previous example illustrates the creative part of writing proofs. The proof started out considering $\sqrt{x}-\sqrt{y}$, which doesn't seem to be related to what we wanted to prove. But hopefully after you read the entire proof you see why it makes sense. If you are saying to yourself "I would never have thought of starting with $\sqrt{x}-\sqrt{y}$?," or "How do you know where to start?," I am afraid there are no easy answers. Writing proofs is as much of an art as it is a science. There are three things that can help, though. First, *don't be afraid to* ***experiment***. If you aren't sure where to begin, try starting at the end. Think about the end goal and work backwards until you see a connection. Sometimes working both backward and forward can help. Try some algebra and see where it gets you. But in the end, make sure your proof goes from beginning to end. In other words, the order that you figured things out should not necessarily dictate the order they appear in your proof.

The second thing you can do is to ***read example proofs***. Although there is some creativity necessary in proof writing, it is important to follow proper proof writing techniques. Although there are often many ways to prove the same statement, there is often one technique that works best for a given type of problem. As you read more proofs, you will begin to have a better understanding of the various techniques used, know when a particular technique might be the best choice, and become better at writing your own proofs. If you see several proofs of similar problems, and the proofs look very similar, then when you prove a similar problem, your proof should probably resemble those proofs. This is one area where some students struggle—they submit proofs that look nothing like any of the examples they have seen, and they are often incorrect. Perhaps it is because they are afraid that they are plagiarizing if they mimic another proof too closely. However, mimicking a proof is not the same as plagiarizing a sentence. To be clear, by 'mimic', I don't mean just copy exactly what you see. I mean that you should read and understand several examples. Once you understand the technique used in those examples, you should be able to see how to use the same technique in your proof. For instance, in many of the examples related to even numbers, you may have noticed that they start with statement like *"Assume x is even. Then $x = 2a$ for some integer a."* So if you need to write a proof related to even numbers, what sort of statement might make sense to begin your proof?

The third thing that can help is ***practice***. This applies not only to writing proofs, but to learning many topics. An analogy might help here. Learning is often like sports—you don't learn how to play basketball (or insert your favorite sport, video game, or other hobby that takes some skill) by reading books and/or watching people play it. Those things can be helpful (and in some cases necessary), but you will never become a proficient basketball player unless you practice. Practicing a sport involves running many drills to work on the fundamentals and then applying the skills you learned to new situations. Learning many topics is exactly the same. First you need to do lots of exercises to practice the fundamental skills. Then you can apply those skills to new situations. When you can do that well, you know you have a good understanding of the topic. So if you want to become better at writing proofs, you need to write more proofs.

$\star$**Question 2.28.** What three things can help you learn to write proofs?

1. ______________________________

2. ______________________________

3. ______________________________

2.2 Implication and Its Friends

This section is devoted to developing some of the concepts that will be necessary for us to discuss the ideas behind the next few proof techniques.

Definition 2.29. *A* **boolean proposition** *(or simply* **proposition***) is a statement which is either* **true** *or* **false***. We call this the* **truth value** *of the proposition.*

Although not technically interchangeable, you may sometimes see the word *statement* instead of *proposition*. Context should help you determine whether or not a given usage of the word *statement* should be understood to mean *proposition*.

Definition 2.30. *An* **implication** *is a proposition of the form "if p, then q," where p and q are propositions.*

It is sometimes written as $p \to q$, which is read "p implies q." It is a statement that asserts that if p is a true proposition then q is a true proposition.

An implication is true unless p is true and q is false.

Example 2.31. The proposition "If I do well in this course, then I can take the next course" is an implication. However, the proposition "I can do well in this course and take the next course" is *not* an implication.

Example 2.32. Consider the implication

"If you read *xkcd*, then you will laugh."[a]

If you read *xkcd* and laugh, you are being consistent with the proposition. If you read *xkcd* and *do not laugh*, then you are demonstrating that the proposition is false.

But what if you don't read *xkcd*? Are you demonstrating that the proposition is true or false? Does it matter whether or not you laugh? It turns out that you are *not* disproving it in this case–in other words, the proposition is still true if you don't read *xkcd*, whether or not you laugh. Why? Because the statement is not saying anything about laughing by itself. It is only asserting that ***IF*** you read *xkcd*, then you will laugh. In other words, it is a *conditional statement*, with the condition being that you read *xkcd*. The statement is saying nothing about anything if you don't read *xkcd*.

So the bottom line is that if you do not read *xkcd*, the statement is still true.

[a] If you are unfamiliar with *xkcd*, go to http://xkcd.com.

⋆**Question 2.33.** When is the implication "If you read *xkcd*, then you will laugh" false?

Answer ______________________________

____

⋆**Exercise 2.34.** Consider the implication "If you build it, they will come." What are all of the possible ways this proposition could be false?

Solution ______________________________

____

____

Given an implication $p \to q$, there are three related proposition. But first we need to discuss the negation of a proposition.

Definition 2.35. *Given a proposition p, the* **negation** *of p, written $\neg p$, is the proposition "not p" or "it is not the case that p."*

Example 2.36. If p is the proposition "$x \leq y$" then $\neg p$ is the proposition "it is not the case that $x \leq y$," or "$x > y$".

Note: *It is easy to incorrectly negate sentences, especially when they contain words like "and", "or", "implies", and "if." This will become easier after we study logic in Chapter 4.*

Definition 2.37. *The* **contrapositive** *of a proposition of the form "if p, then q" is the proposition "if q is not true, then p is not true" or "if not q, then not p" or $\neg q \to \neg p$.*

⋆**Question 2.38.** What is the contrapositive of the proposition "If you know Java, then you know a programming language"?

Answer ______________________________

____

Theorem 2.39. *An implication is true if and only if its contrapositive is true. Stated another way, an implication and its contrapositive are equivalent.*

⋆**Fill in the details 2.40.** Prove Theorem 2.39.

Proof: Let $p \to q$ be our implication. According to the definition of implication, it is false when p is true and q is false and __________ otherwise. Put another way, it is true unless p is true and q is false. The contrapositive, $\neg q \to \neg p$, is false when $\neg q$ is true and _______ is false, and true otherwise. Notice that this is equivalent to q being _______ and _______ being true. Thus, the contrapositive is true unless ________________________ and ________________________. But this is exactly when $p \to q$ is true. □

Definition 2.41. *The* **inverse** *of a proposition of the form "if p, then q" is the proposition "if p is not true, then q is not true" or "if not p, then not q" or $\neg p \to \neg q$.*

⋆**Question 2.42.** What is the inverse of the proposition "If you know Java, then you know a programming language"?

Answer ____

⋆**Question 2.43.** Are a proposition and its inverse equivalent? Explain, using the proposition from Question 2.42 as an example.

Answer ____

____

____

Definition 2.44. *The* **converse** *of a proposition of the form "if p, then q" is the proposition "if q, then p" or $q \to p$.*

⋆**Question 2.45.** What is the converse of the proposition "If you know Java, then you know a programming language"?

Answer ____

⋆**Question 2.46.** Are a proposition and its converse equivalent? Explain using the proposition about Java/programming languages.

Answer ________________________________

____

____

As you have just seen, the *inverse* and *converse* of a proposition are not equivalent to the proposition. However, it turns out that The *inverse* and *converse* of a proposition are equivalent to each other. You will be asked to prove this in Problem 2.2. If you think about it in the right way, it should be fairly easy to prove.

Example 2.47. Here is an implication and its friends:

1. **Implication** If I get to watch "The Army of Darkness," then I will be happy.
2. **Inverse** If I do not get to watch "The Army of Darkness," then I will not be happy.
3. **Converse** If I am happy, then I got to watch "The Army of Darkness."
4. **Contrapositive** If I am not happy, then I didn't get to watch "The Army of Darkness."

⋆**Question 2.48.** Using the propositions from the previous example, answer the following questions.

(a) Give an explanation of why an *implication* might be true, but the *inverse* false.

Answer ________________________________

____

____

(b) Explain why an *implication* is saying the exact same thing as its *contrapositive*. (Don't just say "By Theorem 2.39.")

Answer ________________________________

____

____

Implications can be tricky to fully grasp and it is easy to get your head turned around when dealing with them. We will discuss them in quite a bit of detail throughout the next few sections in order to help you understand them better. We will also revisit them in Chapter 4.

2.3 Proof by Contradiction

In this section we will see examples of *proof by contradiction*. For this technique, when trying to prove a premise, we assume that its negation is true and deduce incompatible statements from this. This implies that the original statement must be true. Let's start by seeing a few examples. Then we'll describe the idea in more detail.

Example 2.49. Prove that if $5n+2$ is odd, then n is odd.

> **Proof:** Assume that $5n+2$ is odd, but that n is even. Then $n = 2k$ for some integer k. This implies that $5n+2 = 5(2k)+2 = 10k+2 = 2(5k+1)$, which is even. But this contradicts our assumption that $5n+2$ is odd. Therefore it must be the case that n is odd. □

The idea behind this proof is that if we are given the fact that $5n+2$ is odd, we are asserting that n must be odd. How do we prove that n is odd? We could try a direct proof, but it is actually easier to use a proof by contradiction in this case. The idea is to consider what would happen if n is *not* odd. What we showed was that if n is not odd, then $5n+2$ has to be even. But we *know* that $5n+2$ is odd because that was our initial assumption. How can $5n+2$ be both odd and even? It can't. In other words, our proof lead to a contradiction–an impossibility. Therefore, something is wrong with the proof. But what? If n is indeed even, our proof that $5n+2$ is even is correct. So there is only one possible problem–n must not be even. The only alternative is that n is odd. Can you see how this proves the statement "if $5n+2$ is odd, then n is odd?"

Note: *If you are somewhat confused at this point that's probably O.K. Keep reading, and re-read this section a few times if necessary. At some point you will have an "Aha" moment and the idea of contradiction proofs will make sense.*

Example 2.50. Prove that if $n = ab$, where a and b are positive integers, then either $a \leq \sqrt{n}$ or $b \leq \sqrt{n}$.

> **Proof:** Let's assume that $n = ab$ but that the statement "either $a \leq \sqrt{n}$ or $b \leq \sqrt{n}$" is false. Then it must be the case that $a > \sqrt{n}$ and $b > \sqrt{n}$. But then $ab > \sqrt{n}\sqrt{n} = n$. But this contradicts the fact that $ab = n$. Since our assumption that $a > \sqrt{n}$ and $b > \sqrt{n}$ lead to a contradiction, it must be false. Therefore it must be the case that either $a \leq \sqrt{n}$ or $b \leq \sqrt{n}$. □

Sometimes your proofs will not directly contradict an assumption made but instead will contradict a statement that you otherwise know to be true. For instance, if you ever conclude that $0 > 1$, that is a contradiction. The next example illustrates this.

⋆**Fill in the details 2.51.** Show, without using a calculator, that $6 - \sqrt{35} < \frac{1}{10}$.

Proof: Assume that $6 - \sqrt{35} \geq \frac{1}{10}$. Then $6 - \frac{1}{10} \geq$ __________. If we multiple both sides by 10 and do a little arithmetic, we can see that $59 \geq$ ____________ .

Squaring both sides we obtain ________________, which is clearly ____________.

Thus it must be the case that $6 - \sqrt{35} < \frac{1}{10}$. □

Now that we have seen a few examples, let's discuss contradiction proofs a little more formally. Here is the basic concept of contradiction proofs: You want to prove that a statement p is true. You "test the waters" by seeing what happens if p is *not* true. So you assume p is false and use proper proof techniques to arrive at a contradiction. By "contradiction" I mean something that cannot possibly be true. Since you proved something that is not true, and you used proper proof techniques, then it must be that your assumption was incorrect. Therefore the negation of your assumption—which is the original statement you wanted to prove—must be true.

⋆**Evaluate 2.52.** Use the definition of even and odd to prove that if a and b are integers and ab is even, then at least one of a or b is even.

Proof 1: By definition of even numbers, let a be an even integer $2n$, and by the definition of odd numbers, let b be an odd integer $2q+1$. Then $(2n)(2q+1) = 4nq + 2n = 2(2nq+1)$. Since $2nq+1$ is an integer, $2(2nq+1)$ is an even integer by definition of even.

Evaluation ____

____

____

Proof 2: If true, either one is odd and the other even, or they are both even, so we will show that the product of an even and an odd is even, and that the product of two evens integers is even.
Let $a = 2k$ and $b = 2x+1$. $(2k)(2x+1) = 4kx + 2k = 2(2kx+k)$. $2kx+k$ is an integer so $2(2kx+k)$ is even.
Let $a = 2k$ and $b = 2x$. $(2k)(2x) = 4kx = 2(2kx)$ since $2kx$ is an integer, $2(2kx)$ is even.
Thus, if a and b are integers, ab is even, at least one of a or b is even.

Evaluation ____

____

Proof 3: Let a and b be integers and assume that ab is even, but that neither a nor b is even. Then both a and b are odd, so $a = 2n + 1$ and $b = 2m + 1$ for some integers n and m. But then $ab = (2n + 1)(2m + 1) = 2mn + 2n + 2m + 1 = 2(nm + n + m) + 1$, which is odd since $nm + n + m$ is an integer. This contradicts the fact that ab is even. Therefore either a or b must be even.

Evaluation

For some students, the trickiest part of contradiction proofs is what to contradict. Sometimes the contradiction is the fact that p is true. At other times you arrive at a statement that is clearly false (e.g. $0 > 1$). Generally speaking, you should just try a few things (e.g. do some algebra) and see where it leads. With practice, this gets easier. In fact, with enough practice this will probably become one of your favorite techniques. When a direct proof doesn't seem to be working this is usually the next technique I try.

Example 2.53. Let a_1, a_2, …, a_n be real numbers. Prove that at least one of these numbers is greater or equal to the average of the numbers.

Proof: The average of the numbers is $A = (a_1 + a_2 + \ldots + a_n)/n$. Assume that none of these numbers is greater than or equal to A. That is, $a_i < A$ for all $i = 1, 2, \ldots n$. Thus $(a_1 + a_2 + \ldots + a_n) < nA$. Solving for A, we get $A > (a_1 + a_2 + \ldots + a_n)/n = A$, which is a contradiction. Therefore at least one of the numbers is greater than or equal to the average. $\square$

Our next contradiction proof involves *permutations*. Here is the definition and an example in case you haven't seen these before.

Definition 2.54. *A* **permutation** *is a function from a finite set to itself that reorders the elements of the set.*

Note: *We will discuss both functions and sets more formally later. For now, just think of a set as a collection of objects of some sort and a function as a black box that produces an output when given an input.*

Example 2.55. Let S be the set $\{a, b, c\}$. Then (a, b, c), (b, c, a) and (a, c, b) are permutations of S. (a, a, c) is not a permutation of S because it repeats a and does not contain b. (b, d, a) is not permutations of S because d is not in S, and c is missing.

⋆**Exercise 2.56.** List all of the permutations of the set $\{1, 2, 3\}$. (Hint: There are 6.)

Answer ______________________________

____

Note: *In many contexts, when a list of objects occurs in* **curly braces**, *the order they are listed does not matter (e.g.* $\{a, b, c\}$ *and* $\{b, c, a\}$ *mean the same thing). On the other hand, when a list occurs in* **parentheses**, *the order* **does** *matter (e.g.* (a, b, c) *and* (b, c, a) **do not** *mean the same thing).*

Example 2.57. Let $(a_1, a_2, \ldots, a_n)$ be an arbitrary permutation of the numbers $1, 2, \ldots, n$, where n is an odd number. Prove that the product $(a_1 - 1)(a_2 - 2) \cdots (a_n - n)$ is even.

Proof: Assume that the product is odd. Then all of the differences $a_k - k$ must be odd. Now consider the sum $S = (a_1 - 1) + (a_2 - 2) + \cdots + (a_n - n)$. Since the a_k's are a just a reordering of $1, 2, \ldots, n$, $S = 0$. But S is the sum of an odd number of odd integers, so it must be odd. Since 0 is not odd, we have a contradiction. Thus our initial assumption that all of the $a_k - k$ are odd is wrong, so at least one of them is even and hence the product is even. $\square$

⋆**Question 2.58.** Why did the previous proof begin by assuming that the product was odd?

Answer ______________________________

____

____

⋆**Question 2.59.** In the previous proof, we asserted that $S = 0$. Why was this the case?

Answer ______________________________

____

____

We will use facts about rational/irrational numbers to demonstrate some of the proof techniques. In case you have forgotten, here are the definitions.

Definition 2.60. *Recall that*

- *A* **rational number** *is one that can be written as p/q, where p and q are integers, with $q \neq 0$.*
- *An* **irrational number** *is a real number that is not rational.*

Example 2.61. Prove that $\sqrt{2}$ is irrational. We present two slightly different proofs. In both, we will use the fact that any positive integer greater than 1 can be factored uniquely as the product of primes (up to the order of the factors).

Proof 1: Assume that $\sqrt{2} = \dfrac{a}{b}$, where a and b are positive integers with $b \neq 0$. We can assume a and b have no factors in common (since if they did, we could cancel them and use the resulting numerator and denominator as a and b). Multiplying by b and squaring both sides yields $2b^2 = a^2$. Clearly 2 must be a factor of a^2. Since 2 is prime, a must have 2 as a factor, and therefore a^2 has 2^2 as a factor. Then $2b^2$ must also have 2^2 as a factor. But this implies that 2 is a factor of b^2, and therefore a factor of b. This contradicts the fact that a and b have no factors in common. Therefore $\sqrt{2}$ must be irrational.

Proof 2: Assume that $\sqrt{2} = \dfrac{a}{b}$, where a and b are positive integers with $b \neq 0$. Multiplying by b and squaring both sides yields $2b^2 = a^2$. Now both a^2 and b^2 have an even number of prime factors. So $2b^2$ has an odd number of primes in its factorization and a^2 has an even number of primes in its factorization. This is a contradiction since they are the same number. Therefore $\sqrt{2}$ must be irrational.

⋆**Question 2.62.** In proof 2 from the previous example, why do a^2 and b^2 have an even number of factors?

Answer ______________________________

____

____

Now that you have seen a few more examples, it is time to begin the discussion about how/why contradiction proofs work. We will start with the following idea that you may not have thought of before. It turns out that if you start with a false assumption, then you can prove that *anything* is true. It may not be obvious how (e.g. How would you prove that all elephants are less than 1 foot tall given that $1 + 1 = 1$?), but in theory it is possible. This is because statements of the form "p implies q" are true if p (called the *premise*) is false, regardless of whether or not q (called the *conclusion*) is true or false.

Example 2.63. The statement "If chairs and tables are the same thing, then the moon is made of cheese" is true. This may seem weird, but it is correct. Since chairs and tables are not the same thing, the premise is false so the statement is true. But it is important to realize that the fact that the whole statement is true doesn't tell us anything about whether or not the moon is made of cheese. All we know is that *if* chairs and tables were the same thing, then the moon *would have to* be made out of cheese in order for the statement to be true.

Example 2.64. Consider what happens if your parents tell you "If you clean your room, then we will take you to get ice cream." If you don't clean your room and your parents don't take you for ice cream, did your parents tell a lie? No. What if they *do* take you for ice cream? They still haven't lied because they didn't say they wouldn't take you if you didn't clean your room. In other words, if the premise is false, the whole statement is true regardless of whether or not the conclusion is true.

It is important to understand that when we say that a statement of the form "p implies q" is true, we are *not* saying that q is true. We are only saying that *if p is true, then q has to be true.* We aren't saying anything about q by itself. So, if we know that "p implies q" is true, and we also know that p is true, then q must also be true. This is a rule called *modus ponens*, and it is at the heart of contradiction proofs as we will see shortly.

⋆**Exercise 2.65.** It might help to think of statements of the form "p implies q" as rules where breaking them is equivalent to the statement being false. For instance, consider the statement *"If you drink alcohol, you must be 21."* If we let p be the statement "you drink alcohol" and q be the statement "you are 21," the original statement is equivalent to "p implies q".

1. If you drink alcohol and you are 21, did you break the rule? ____________

2. If you drink alcohol and you are not 21, did you break the rule? ____________

3. If you do not drink alcohol and you are 21, did you break the rule? ____________

4. If you do not drink alcohol and you are not 21, did you break the rule? ____________

5. Generalize the idea. If you have a statement of the form "p implies q", where p and q can be either true or false statements, exactly when can the statement be false?

   ____

6. If you do not drink alcohol, does it matter how old you are? ____________

7. Can a statement of the form "p implies q" be false if p is false? Explain.

   ____

   ____

Now we are ready to explain the idea behind contradiction proofs. We want to prove some statement p is true. We begin by assuming it is false—that is, we assume $\neg p$ is true. We use this fact to prove that q—some false statement—is true. In other words, we prove that the statement "$\neg p$ implies q" is true, where q is some false statement. But if $\neg p$ is true, and "$\neg p$ implies q" is true, modus ponens tells us that q has to be true. Since we know that q is false, something is wrong. We only have two choices: either $\neg p$ is false or "$\neg p$ implies q" is false. If we used proper proof techniques to establish that "$\neg p$ implies q" is true, then that isn't the problem. Therefore, the only other possibility is that $\neg p$ is false, implying that p must be true. That is how/why contradiction proofs work.

Let's analyze the second proof from Example 2.61 in light of this discussion. The *only* assumption we made was that $\sqrt{2}$ is rational ($\neg p$="$\sqrt{2}$ is rational"). From this (and only this), we were able to show that a^2 has both an even and an odd number of factors (q="a^2 has an even and an odd number of factors", and we showed that "$\neg p$ implies q" is true). Thus, we know for certain that if $\sqrt{2}$ is rational, then a^2 has an even and an odd number of factors.[1] This fact is indisputable since we proved it. If it is also true that $\sqrt{2}$ is rational, modus ponens implies that a^2 has an even and an odd number of factors. This is also indisputable. But we know that a^2 cannot have both an even and odd number of factors. In other words, we have a contradiction. Therefore, something that has been said somewhere is wrong. Everything we said is indisputable except for one thing–that $\sqrt{2}$ is rational. That was never something we proved—we just assumed it. So it has to be the case that this is false, which means that $\sqrt{2}$ must be irrational.

To summarize, if you want to prove that a statement is true using a contradiction proof, assume the statement is false, use this assumption to get a contradiction (i.e. prove a false statement), and declare that it must therefore be true.

Notice that what q is doesn't matter. In other words, given the assumption $\neg p$, the goal in a contradiction proof is to establish that *any* false statement is true. This is both a blessing and a curse. The blessing is that any contradiction will do. The curse is that we don't have a clear goal in mind, so it can sometimes be difficult to know what to do. As mentioned previously, this becomes easier as you read and write more proofs.

If this discussion has been a bit confusing, try re-reading it. The better you understand the theory behind contradiction proofs, the better you will be at writing them. We will revisit some of these concepts in the chapter on logic, so the more you understand from here, the better off you will be when you get there. O.K., enough theory. Let's see some more examples!

[1] We did not prove that a^2 has an even and an odd number of factors. We proved that *if $\sqrt{2}$ is rational*, then a^2 has an even and an odd number of factors. It is very important that you understand the difference between these two statements.

⋆**Fill in the details 2.66.** Let a, b be real numbers. Prove that if $a < b + \epsilon$ for all $\epsilon > 0$, then $a \leq b$.

Proof: We will prove this by contradiction. Assume that __________.[a]

Subtracting b from both sides and dividing by 2, we get ____________> 0. Since the inequality $a < b + \epsilon$ holds for every $\epsilon > 0$ in particular it holds for

$\epsilon =$____________.[b] This implies that

$$a < b + \frac{a-b}{2} = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ .$$

If we ____________________________________ (to the previous equation), we

obtain $a < b$. But we started with the assumption that ____________ which is

a ____________. Therefore, ________________________. □

[a] Hint: What assumption do we always make when doing a contradiction proof?
[b] Same as the previous blank

The following beautiful proof goes back to Euclid. It uses the assumption that any integer greater than 1 is either a prime or a product of primes.

Example 2.67 (Euclid)**.** Show that there are infinitely many prime numbers.

Proof: Assume that there are only a finite number of primes and let $\{p_1, p_2, \ldots, p_n\}$ be a list of all the primes. Consider the number

$$N = p_1 p_2 \cdots p_n + 1.$$

This is a positive integer that is clearly greater than 1. Observe that none of the primes on the list $\{p_1, p_2, \ldots, p_n\}$ divides N, since division by any of these primes leaves a remainder of 1. Since N is larger than any of the primes on this list, it is either a prime or divisible by a prime outside this list. But we assumed the list above contained all of the prime numbers. This is a contradiction. Therefore there must be infinitely many primes. □

⋆**Fill in the details 2.68.** If a, b, c are odd integers, prove that $ax^2 + bx + c = 0$ does not have a rational number solution.

Proof: Suppose $\frac{p}{q}$ is a rational solution to the equation. We may assume that p and q have no prime factors in common, so either p and q are both odd, or one is odd and the other even. Since $\frac{p}{q}$ is a solution, we know that

$$\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad} = 0.$$

If we ________________________, we obtain $ap^2 + bpq + cq^2 = 0$.

If both p and q are odd, then $ap^2 + bpq + cq^2$ is ____________ which contradicts the fact that it is ______.

If p is even and q odd, then ____

____.

If p is odd and q even, then ____

____.

Since all possibilities leads to a contradiction, __________________________.

____

□

2.4 Proof by Contraposition

Consider the statement "If it rains, then the ground will get wet." It should be pretty easy to convince yourself that this is essentially equivalent to the statement "If the ground is not wet, then it didn't rain." In fact, since the second statement is just the contrapositive of the first, Theorem 2.39 tells us that they are equivalent. Again, by *equivalent* we simply mean that either both statements are true or both statements are false. This is the idea behind the proof technique in this section.

Definition 2.69. *A* **proof by contraposition** *is a proof of a statement of the form "if p, then q" that proves contrapositive statement instead. That is, it proves the equivalent statement "if not q, then not p."*

Example 2.70. Prove that if $5n + 2$ is odd, then n is odd.

> **Proof:** We will instead prove that if n is even (not odd), then $5n + 2$ is even (not odd). Since this is the contrapositive of the original statement, a proof of this will prove that that the original statement is true.
> Assume n is even. The $n = 2a$ for some integer a. Then $5n + 2 = 5(2a) + 2 = 2(5a + 1)$. Since $5a + 1$ is an integer, $2(5a + 1)$ is even. $\square$

Be careful with proof by contraposition. Do not make the mistake of trying to prove the *converse* or *inverse* instead of the *contrapositive*. In that case, you may write a correct proof, but it would be a proof of the wrong thing.

In the next example we will see the similarities and differences between contradiction proofs and proofs by contraposition.

Example 2.71. Prove that if $5n + 2$ is even, then n is even.

Proof by contraposition:
We will prove the equivalent statement that if n is odd, then $5n + 2$ is odd. Assume n is odd. Then $n = 2k + 1$ for some integer k. Then we have that

$$\begin{aligned} 5n + 2 &= 5(2k + 1) + 2 \\ &= 10k + 5 + 2 \\ &= 10k + 7 \\ &= 2(5k + 3) + 1 \end{aligned}$$

Since $5k + 3$ is an integer, this shows that $5n + 2$ is odd.

Proof by contradiction:
Assume that $5n + 2$ is even but that n is odd. Since n is odd, $n = 2k + 1$ for some integer k. Therefore

$$\begin{aligned} 5n + 2 &= 5(2k + 1) + 2 \\ &= 10k + 5 + 2 \\ &= 10k + 7 \\ &= 2(5k + 3) + 1 \end{aligned}$$

which is odd since $5k + 3$ is an integer. But we assumed that $5n + 2$ was even, which is a contradiction. Therefore our assumption that n is odd must be incorrect, so n is even.

⋆**Evaluate 2.72.** Let n be an integer. Use the definition of even/odd to prove that if $3n+2$ is even, then n is even using a proof by contraposition.

Proof 1: We need to show that if n is even, then $3n+2$ is even. If n is even, then $n=2k$ for some integer k. Then $3n+2=3(2k+2)=6k+6=2(3k)+2(3)$, which is even because it is the sum of two even integers.

Evaluation ______________________________

____

____

Proof 2: We need to show that if n is odd, then $3n+2$ is odd. If n is odd then $n=2k+1$ for some integer k. Then $3n+2=3(2k+1)+2=6k+3+2=6k+5=5(\frac{6}{5}k+1)$, which is clearly odd.

Evaluation ______________________________

____

____

Proof 3: We need to show that if n is odd, then $3n+2$ is odd. If n is odd then $n=2k+1$ for some integer k. Then $3n+2=3(2k+1)+2=6k+5$, which is odd by the definition of odd.

Evaluation ______________________________

____

____

2.5 Other Proof Techniques

There are many other proof techniques. We conclude this chapter with a small sampling of the more common and/or interesting ones. We will see a few other important proof techniques later in the book.

Definition 2.73. *A* **trivial proof** *is a proof of a statement of the form "if p, then q" that doesn't use p in the proof.*

Example 2.74. Prove that if $x > 0$, then $(x+1)^2 - 2x > x^2$.

Proof: It is easy to see that

$$\begin{aligned}(x+1)^2 - 2x &= (x^2 + 2x + 1) - 2x \\ &= x^2 + 1 \\ &> x^2.\end{aligned}$$

Notice that we never used the fact that $x > 0$ in the proof. □

Definition 2.75. *A* **proof by counterexample** *is used to disprove a statement by giving an example of it being false.*

Example 2.76. Prove or disprove that the product of two irrational numbers is irrational.

Proof: We showed in Example 2.61 that $\sqrt{2}$ is irrational. But $\sqrt{2} * \sqrt{2} = 2$, which is an integer so it is clearly rational. Thus the product of 2 irrational number is not always irrational. □

Example 2.77. Prove or disprove that "Everybody Loves Raymond" (or that "Everybody Hates Chris").

Proof: Since I don't really love Raymond (and I don't hate Chris), the statement is clearly false. □

⋆**Exercise 2.78.** Prove or disprove that the sum of any two primes is also prime.

Proof ____

____

____

Definition 2.79. *A* **proof by cases** *breaks up a statement into multiple cases and proves each one separately.*

We have already seen several examples of proof by cases (e.g. Examples 2.24 and 2.68), but it never hurts to see another example.

Example 2.80. Prove that if $x \neq 0$ is a real number, then $x^2 > 0$.

Proof: If $x \neq 0$, then either $x > 0$ or $x < 0$.
If $x > 0$ (case 1), then we can multiply both sides of $x > 0$ by x, giving $x^2 > 0$.
If $x < 0$ (case 2), then we can write y=-x, where $y > 0$. Then $x^2 = (-y)^2 = (-1)^2y^2 = y^2 > 0$ by case 1 (since $y > 0$). Thus $x^2 > 0$. In either case, we have shown that $x^2 > 0$. $\square$

⋆**Fill in the details 2.81.** Let s be a positive integer. Prove that the closed interval $[s, 2s]$ contains a power of 2.

Proof: If s is a power of 2 then ______________________________
If s is not a power of 2, then it is strictly between two powers of 2. That is,

$2^{r-1} < s < 2^r$ for some integer r. Then ______________________________

____

____

$\square$

2.6 If and Only If Proofs

Sometimes we will run into "if and only if" (abbreviated *iff*) statements. That is, statements of the form p **if and only if** q. This is equivalent to the statement "p implies q and q implies p." Thus, to prove that an iff statement is true, you need to prove a statement and its *converse*. "p implies q" is sometimes called the *forward direction* and the converse is sometimes called the *backwards direction*. Sometimes the converse statement is proven by *contaposition*, so that instead of proving q implies p, $\neg p$ implies $\neg q$ is proven.

⋆**Question 2.82.** Why is there a choice between proving q **implies** p and proving $\neg p$ **implies** $\neg q$ when proving the backwards direction?

Answer ____

____

Example 2.83. Prove that x is even if and only if $x + 10$ is even.

> **Proof:** If x is even, then $x = 2k$ for some integer k. Then $x + 10 = 2k + 10 = 2(k+5)$. Since $k+5$ is an integer, then $x+10$ is even. Conversely, if $x+10$ is even, then $x + 10 = 2k$ for some integer k. Then $x = (x+10) - 10 = 2k - 10 = 2(k-5)$. Since $k - 5$ is an integer, then x is even. Therefore x is even iff $x + 10$ is even. $\square$

As we have mentioned before, the examples in this section are quite trivial and may seem ridiculous–since they are so obvious, why are we bothering to prove them? The point is to use the proof techniques we are learning. We will use the techniques on more complicated problems later. For now we want the focus to be on proper use of the techniques. That is more easily accomplished if you don't have to think too hard about the details of the proof.

⋆**Exercise 2.84.** Prove that x is odd iff $x + 20$ is odd using direct proofs for both directions

⋆**Exercise 2.85.** Prove that x is odd iff $x + 20$ is odd using using a direct proof for the forward direction and a proof by contraposition for the backward direction.

⋆**Fill in the details 2.86.** The two most common ways to prove p iff q are

1. Prove that ____________________ and ____________________, or

2. Prove that ____________________ and ____________________.

⋆**Evaluate 2.87.** Use the definition of odd to prove that x is odd if and only if $x - 4$ is odd.

Proof 1: Assume x is odd. Then $x = 2k + 1$ for some integer k. Then $x - 4 = 2k + 1 - 4 = 2k - 3$, which is odd. Now assume that $x - 4$ is odd. Since $(2k + 1) - 4$ is odd, then $x = 2k + 1$ is clearly odd.

Evaluation ____

____

____

Proof 2: Assume x is odd. Then $x = 2k + 1$, so $x - 4 = (2k + 1) - 4 = 2(k - 2) + 1$, which is odd since $k - 2$ is an integer. Now assume $x - 4$ is even. Then $x - 4 = 2k$ for some integer k. Then $x = 2k + 4 = 2(k + 2)$, which is even since $k + 2$ is an integer.

Evaluation ____

____

2.7 Common Errors in Proofs

If you arrive at the right conclusion, does that mean your proof is correct? Some students seem to think so, but this is absolutely false. Let's consider the following example.

Example 2.88. Is the following proof that $\frac{16}{64} = \frac{1}{4}$ correct? Why or why not?

> **Proof:** This is true because if I cancel the 6 on the top and the bottom, I get $\frac{16}{64} = \frac{1\cancel{6}}{\cancel{6}4} = \frac{1}{4}$ □
>
> **Evaluation:** You probably know that you can't cancel arbitrary digits in a fraction, so this is not a valid proof. In addition, consider this: If this proof is correct, then it could be used to prove that $\frac{16}{61} = \frac{1\cancel{6}}{\cancel{6}1} = \frac{1}{1} = 1$, which is clearly false.

Note: *The point of the previous example is this: Don't confuse the fact that what you are trying to prove is true with whether or not your proof actually* proves *that it is true. An incorrect proof of a correct statement is no proof at all.*

A common mistake when writing proofs is to make one or more invalid assumptions without realizing it. The problem with this is that it generally means you are not proving what you set out to prove, but since the proof seems to "work", the mistake isn't always obvious. The next few examples should illustrate what can go wrong if you aren't careful.

⋆**Question 2.89.** What is wrong with this proof that the sum of two even integers is even?

> Proof: Let x and y be even integers. Then $x = 2a$ for some integer a and $y = 2a$ for some integer a. So $x + y = 2a + 2a = 2(a + a)$. Since $a + a$ is an integer, $2(a + a)$ is even, so the sum of two even integers is even. □

Answer ____

____

Since the statement in the previous example is true, it can be difficult to appreciate why the proof is wrong. The proof seems to prove the statement but as you saw in the solution, it actually doesn't. It proves a more specific statement. If it seems like we are being too nit-picky, consider the next example.

⋆**Question 2.90.** What is wrong with the following proof that the sum of two even integers is divisible by 4?

Proof: Let x and y be two even integers. Then $x = 2a$ for some integer a and $y = 2a$ for some integer a. So $x + y = 2a + 2a = 4a$. Since a is an integer, $4a$ is divisible by 4, so the sum of two even integers is divisible by 4. □

Answer ____

____

Another common mistake students make when trying to prove an identity/equation is to start with what they want to prove and work both sides of it until they demonstrate that they are equal. I want to stress that *this is an invalid proof technique.* Again, if this seems like I am making something out of nothing, consider this example:

⋆**Question 2.91.** Consider the following supposed proof that $-1 = 1$.

Proof:

$$\begin{aligned} -1 &= 1 \\ (-1)^2 &= 1^2 \\ 1 &= 1 \end{aligned}$$

Therefore $-1 = 1$. □

How do you know that this proof is incorrect? (Think about the obvious reason, not any technical reason.)

Answer ____

Notice that each step of algebra in the previous proof is correct. For instance, if $a = b$, then $a^2 = b^2$ is correct. And $(-1)^2$ and 1^2 are both equal to 1. So the majority of the proof contains proper techniques. It contains just one problem: It starts by assuming something that isn't true. Unfortunately, one error is all it takes for a proof to be incorrect.

Note: *When writing proofs,* **never** *assume something that you don't already know to be true!*

⋆**Question 2.92.** When you are given an equation to prove, should you prove it by writing it down and working both sides until you get them both to be the same? Why or why not?

Answer ____

____

Let's be clear about this issue. If an equation is correct, you can work both sides until they are the same. But as Example 2.91 demonstrated, if an equation is *not correct*, sometimes you can also work both sides until they are the same. This should tell you something about this technique.

⋆**Question 2.93.** You are given an equation. You work both sides of it until they are the same. Should you now be convinced that the equation is correct? Why or why not?

Answer ______________________________

____

Note: *If you already know that an equation is true, then working both sides of it (for some purpose other than demonstrating it is true)* is *a valid technique. However, it is more common to start with a known equation and work just one side until it is what we want.*

There are plenty of other common errors in proofs. We will see more of them throughout the remainder of the book, especially in the *Evaluate* examples.

2.8 More Practice

Now you will have a chance to practice what you have learned throughout this chapter with some more exercises. Now that they aren't in a particular section, you will have to figure out what technique to use.

⋆**Exercise 2.94.** Let $p < q$ be two *consecutive* odd primes (two primes with no other primes between them). Prove that $p + q$ is a composite number. Further, prove that it has at least three, not necessarily distinct, prime factors. (Hint: think about the average of p and q.)
Proof:

⋆**Evaluate 2.95.** Prove or disprove that if x and y are rational, then x^y is rational.

Proof 1: Because x and y are both rational, assume $x = a/b$ where a and b are integers and $b \neq 0$. We can assume that a and b have no factors in common (since if they did we could cancel them and use the resulting numbers as our new a and b). Then $x^y = \frac{a^y}{b^y}$, so x^y is rational.

Evaluation ____

Proof 2: Notice that x^y is just x multiplied by itself y times. A rational number multiplied by a rational number is rational, so x^y is rational.

Evaluation ____

Since none of the proofs in the previous example were correct, you need to prove it.

⋆**Exercise 2.96.** Prove or disprove that if x and y are rational, then x^y is rational.
Proof:

⋆**Evaluate 2.97.** Prove or disprove that if x is irrational, then $1/x$ is irrational.

Proof 1: If x is rational, assume it is an integer. If x is an integer, it is rational. 1/x is an integer over an integer, so it is rational. Therefore if x is rational, 1/x is rational, so by contrapositive reasoning, if x is irrational, 1/x is irrational.

Evaluation ______________________________

____

____

Proof 2: Assume that x is irrational. Then it cannot be expressed as an integer over an integer. Then clearly 1/x cannot be expressed as an integer over an integer.

Evaluation ______________________________

____

____

Proof 3: Assume that x is rational. Then $x = \frac{p}{q}$, where p and q are integers and $q \neq 0$. But then $\frac{1}{x} = \frac{1}{\frac{p}{q}} = \frac{q}{p}$, so it is rational. Since we proved the contrapositive, the statement is true.

Evaluation ______________________________

____

____

Proof 4: We will prove the contrapositive. Assume that $1/x$ is rational. Since it is rational, $1/x = a/b$ for some integers a and b, with $b \neq 0$. Solving for x we get $x = b/a$, so x is rational.

Evaluation ______________________________

____

____

Proof 5: I will prove the contrapositive statement: If $1/x$ is rational, then x is rational. Assume $1/x$ is rational. Then $\frac{1}{x} = \frac{a}{b}$ for some integers a and $b \neq 0$. We know that $1/x \neq 0$ (since otherwise $x \cdot 0 = 1$, which is impossible), so $a \neq 0$. Multiplying both sides of the previous equation by x we get $x\frac{a}{b} = 1$. Now if we multiply both sides by $\frac{b}{a}$ (which we can do since $a \neq 0$), we get $x = \frac{b}{a}$. Since a and b are integers with $a \neq 0$, x is rational.

Evaluation ______________________________

____

____

⋆**Evaluate 2.98.** Mersenne primes are primes that are of the form $2^p - 1$, where p is prime. Are all numbers of this form prime? Give a proof/counterexample.

Proof 1: Restate the problem as if $2^p - 1$ is prime then p is prime. Assume p is not prime so $p = st$, where s and t are integers. Thus $2^p - 1 = 2^{st} - 1 = (2^s - 1)(2^{st-s} + 2^{st-2s} + \cdots + 2^s + 1)$. Because neither of these factors is 1 or $2^p - 1$
$\rightarrow 2^p - 1$ is not prime (contradiction)
$\rightarrow p$ is prime
$\rightarrow$ All numbers of the form $2^p - 1$ (with p a prime) are prime.

Evaluation ______________________________

____

____

Proof 2: Numbers of the form 2^p only have 2 as a factor. Since $2^p - 1$ is clearly odd, it does not have 2 as a factor. Therefore it must not have any factors. So it is prime.

Evaluation ______________________________

____

⋆**Exercise 2.99.** Let p be prime. Prove that not all numbers of the form $2^p - 1$ are prime.
Proof:

2.9 Problems

Problem 2.1. Prove that a number and its square have the same parity. That is, the square of an even number is even and the square of an odd number is odd.

Problem 2.2. Prove that the inverse of an implication is true if and only if the converse of the implication is true.

Problem 2.3. Let a and b be integers. Consider the problem of proving that if at least one of a or b is even, then ab is even. Is this equivalent to the statement from Evaluate 2.52? Explain, using the appropriate terminology from this chapter.

Problem 2.4. Rephrase the statement from Evaluate 2.52 without using the words *even* or *not*. Using terminology from this chapter, how did you come up with the alternative phrasing?

Problem 2.5. Prove or disprove that there are 100 consecutive positive integers that are not perfect squares. (Recall: a number is a perfect square if it can be written as a^2 for some integer a.)

Problem 2.6. Consider the equation $n^4 + m^4 = 625$.

(a) Are there any *integers* n and m that satisfy this equation? Prove it.

(b) Are there any *positive integers* n and m that satisfy this equation? Prove it.

Problem 2.7. Consider the equation $a^3 + b^3 = c^3$ over the integers (that is, a, b, and c have to all be integers).

(a) Prove that the equations has infinitely many solutions.

(b) If we restrict a, b, and c to the positive integers, are there infinitely many solutions? Are there any? Justify your answer. (Hint: Do a web search for "Fermat's Last Theorem.")

Problem 2.8. Prove that a is even if and only if a^2 is even.

Problem 2.9. Prove that ab is odd iff a and b are both odd.

Problem 2.10. Let n be an odd integer and k an integer. Prove that kn is odd iff k is odd.

Problem 2.11. Let n be an integer.

(a) Prove that if n is odd, then $3n + 4$ is odd.

(b) Is it possible to prove that n is odd iff $3n + 4$ is odd? If so, prove it. If not, explain why not (i.e. give a counter example).

(c) If we don't assume n has to be an integer, is it possible to prove that n is odd iff $3n + 4$ is odd? If so, prove it. If not, explain why not (i.e. give a counter example).

Problem 2.12. Let n be an integer.

(a) Prove that if n is odd, then $4n + 3$ is odd.

(b) Is it possible to prove that n is odd iff $4n + 3$ is odd? If so, prove it. If not, explain why not (i.e. give a counter example).

Problem 2.13. Prove that the product of two rational numbers is rational.

Problem 2.14. Prove or disprove: Every positive integer can be written as the sum of the squares of two integers.

Problem 2.15. Prove that the product of a non-zero rational number and an irrational number is irrational.

Problem 2.16. Prove that if n is an integer and $5n + 4$ is even, then n is even using a

(a) direct proof

(b) proof by contraposition

(c) proof by contradiction

Problem 2.17. Prove or disprove that n^2-1 is composite whenever n is a positive integer greater than or equal to 1.

Problem 2.18. Prove or disprove that n^2-1 is composite whenever n is a positive integer greater than or equal to 3.

Problem 2.19. Prove or disprove that $P = NP$.[2]

[2] A successful solution to this will earn you an A in the course. You are free to use Google or whatever other resources you want for this problem, but you must fully understand the solution you submit.

Chapter 3

Programming Fundamentals and Algorithms

The purpose of this chapter is to review some of the programming concepts you should have picked up in previous classes while introducing you to some basic algorithms and new terminology that we will find useful as we continue our study of discrete mathematics. We will also practice our skills at proving things by sometimes proving that an algorithm does as specified.

Algorithms are presented in a syntax similar to Java and C++. This can be helpful since you should already be familiar with it. On the other hand, this sort of syntax ties our hands more than one often likes when discussing algorithms. What I mean is that when discussing algorithms, we often want to gloss over some of the implementation details. For instance, we may not care about data types, or how parameters are passed (i.e. by value or by reference), but by using a Java-like syntax we are forcing ourselves to use particular data types and pass parameters in a certain way.

Consider an algorithm that swaps two values (we will see an implementation of this shortly). The concept is the same regardless of what type of data is being swapped. But given our choice of syntax, we will give an implementation that assumes a particular data type. Most of the time the algorithms presented can be modified to work with other data types.

The issue of pass-by-value versus pass-by-reference is more complicated. We will have a brief discussion of this later, but the bottom line is that whenever you implement an algorithm from *any* source, you need to consider how this and other language-specific features might change how you understand the algorithm, how you implement it, and/or whether you even *can*.

3.1 Algorithms

An *algorithm* is a set of instructions that accomplishes a task in a finite amount of time.

Example 3.1 (Area of a Trapezoid)**.** Write an algorithm that gives the area of a trapezoid with height h and bases a and b.

Solution: One possible solution is

```
double AreaTrapezoid(double a, double b, double h) {
    return h*(a+b)/2;
}
```

Note: *Notice that we use the* **return** *keyword to indicate what value should be passed to whoever calls an algorithm. For instance, if someone calls* `x=AreaTrapazoid(a, b, h)`*, then* x *will be assigned the value* $h*(a+b)/2$ *since this is what was* returned *by the algorithm. Those who know* Java*,* C*,* C++*, or just about any other programming language should already be familiar with this concept.*

⋆**Exercise 3.2.** Write an algorithm that returns the area of a square that has sides of width w.

```
double areaSquare(double w) {


}
```

Definition 3.3. *The* **assignment operator***,* $=$*, assigns to the left-hand argument the value of the right-hand argument.*

Example 3.4. The statement `x = a + b` means "assign to x the value of a plus the value of b."

Note: *Most modern programming languages use* $=$ *for assignment. Other symbols used include* $:=$*,* $=:$*,* $<<$*,* $\leftarrow$*, etc.*

As it turns out, the most common symbol for assignment ($=$*) is perhaps the most confusing for someone who is first learning to program. One of the most common assignment statements is* `x = x + 1;`*. What this* means *is "assign to the* x *its current value plus one." However, what it* looks like *is the mathematical statement "*x *is equal to* $x+1$*", which is false for every value of* x*. If this has tripped you up in the recent past or still does, fear not. Eventually you will instinctively interpret it correctly, probably forgetting you were ever confused by it.*

Example 3.5 (Swapping variables)**.** Write an algorithm that will interchange the values of two variables, x and y. That is, the contents of x becomes that of y and vice-versa.

Solution: We introduce a temporary variable t in order to store the contents of x in y without erasing the contents of y. For simplicity, we will assume the data is of type `Object`.

```
void swap(Object x, Object y) {
    Object t = x;  // Store x in a temporary variable
    x = y;         // x now has the original value of y
    y = t;         // y now has the original value of x
}
```

It can be very useful to be able to prove that an algorithm actually does what we think it does. Then when an error is found in a program we can focus our attention on the portions of the code that we are uncertain about, ignoring the code that we *know* is correct.

Example 3.6. Prove that the algorithm in Example 3.5 works correctly.

Proof: Assume the values a and b are passed into `swap`. Then at the beginning of the algorithm, $x = a$ and $y = b$. We need to prove that after the algorithm is finished, $x = b$ and $y = a$.

After the first line, x and y are unchanged and $t = a$ since it was assigned the value stored in x, which is a. After the second line, $x = b$ since it is assigned the value stored in y, which is b. Currently $x = b$, $y = b$, and $t = a$. Finally, after the third line, $y = a$ since it is assigned the value stored in t, which is a. Since x is still b, and $y = a$, the algorithm works correctly. □

Note: *The correctness of this algorithm (and a few others in this chapter) is based on the assumption that the variables are* **passed by reference** *rather than* **passed by value**.

In C and C++, it is possible to pass by value or by reference, although we didn't use the proper syntax to do so. The `*` *or* `&` *you sometimes see in argument lists is related to this. In Java, everything is passed by value and it is impossible to pass by reference. However, because non-primitive variables in Java are essentially reference variables (that is, they store a reference to an object, not the object itself), in some ways they act as if they were passed by reference. This is where things start to get complicated. I don't want to get into the subtleties here, especially since there are arguments about whether or not these are the best terms to use. Let me give an analogy instead.*[a]

If I share a Google Doc with you, I am passing by reference. We both have a reference to the same document. If you change the document, I will see the changes. If I change the document, you will see the changes. On the other hand, if I e-mail you a Word document, I am passing by value. You have a copy of the document I have. Essentially, I copied the current **value** *(or contents) of the document and gave that to you. If you change the document, my document will remain unchanged. If I change my document, your document will remain unchanged. This sounds pretty simple. However, it gets more complicated. In Java, you can create a "primitive" Word document, but in a sense you can't create an "object" Word document. Instead, a Google Doc is created and you are given access (i.e. a reference) to it. This is why in some ways primitive and object variables seem to act differently in Java.*

I've already said too much. You will/did learn a lot more about this issue in another course. Here is the bottom line: The assumption being made in the various swap algorithms is that when a variable is passed in, the algorithm has direct access to that variable *and not just a copy of it. Thus if changes are made to that variable in the algorithm, it is changing the variable that was passed in. This subtlety does not matter for most of the algorithms here.*

[a] Inspired by a response on http://stackoverflow.com/questions/373419/whats-the-difference-between-passing-by-reference-vs-passing-by-value

Note: *We should be absolutely clear that it is* **impossible** *to implement the* `swap` *method from Example 3.5 in Java. In fact, there is no way to implement a method that swaps two arbitrary values in Java. As we will see shortly, it* **is** *possible to implement a method that swaps two elements from an array.*

Note: *One final note before we move on. Whether or not the* `swap` *method (or any method) can be implemented, we can still use it in other algorithms as if it can. This is because when discussing algorithms we are usually more concerned about the* **idea** *behind the algorithm, not all of the implementation details. Using a method like* `swap` *in another algorithm often makes it easier to understand the overall concept of that algorithm. If we actually wanted to implement an algorithm that uses* `swap`*, we would simply need to replace the call to* `swap` *with some sort of code that accomplishes the task if* `swap` *is impossible to implement.*

⋆**Question 3.7.** Does the following swap algorithm work properly? Why or why not?

```
void swap(Object x, Object y) {
    x = y;
    y = x;
}
```

Answer ____

____

Example 3.8. Write an algorithm that will interchange the values of two variables x and y *without introducing a third variable*, assuming they are of some numeric type.

Solution: The idea is to use sums and differences to store the values. Assume that initially $x = a$ and $y = b$.

```
void swap(number x, number y) {
    x = x + y;  // x = a+b        and  y = b
    y = x - y;  // x = a+b        and  y = a+b-b = a
    x = x - y;  // x = a+b-a = b  and  y = a
}
```

Notice that the comments in the code sort of provide a proof that the algorithm is correct, although keep reading for an important disclaimer.

Example 3.9. It was mentioned that the comments in the algorithm from Example 3.8 provide a proof of its correctness. What possibly faulty assumption is being made?

Solution: It is assumed that the arithmetic is performed with absolute precision, and that is not always the case. For instance, after the first line we are told that $x = a + b$. What if $a = 10,000,000,000$ and $b = .00000000001$? Will x really be *exactly* $10,000,000,000.00000000001$? If it isn't, the algorithm will not work properly.

Problem 3.16 explores whether or not the algorithm in Example 3.8 works for integer types–specifically 2's complement numbers.

3.2 The *mod* operator and Integer Division

Definition 3.10. *The* **mod** *operator is defined as follows: for integers a and n such that $a \geq 0$ and $n > 0$,* $\boldsymbol{a}$ **mod** $\boldsymbol{n}$ *is the integral non-negative remainder when* $\boldsymbol{a}$ *is divided by* $\boldsymbol{n}$*. Observe that this remainder is one of the* $\boldsymbol{n}$ *numbers*

$$\mathbf{0}, \quad \mathbf{1}, \quad \mathbf{2}, \quad \ldots, \quad \boldsymbol{n}-\mathbf{1}.$$

Java, C, C++, and many other languages use `%` *for* **mod** *(e.g.* `int x = a % n` *instead of* `int x = a mod n`*).*

Example 3.11. Here are some example computations:

$234 \bmod 100 = 34$	$1961 \bmod 37 = 0$	$6 \bmod 5 = 1$
$38 \bmod 15 = 8$	$1966 \bmod 37 = 5$	$11 \bmod 5 = 1$
$15 \bmod 38 = 15$	$1 \bmod 5 = 1$	$16 \bmod 5 = 1$

⋆**Exercise 3.12.** Compute the following:

(a) $345 \bmod 100 =$ ______ (d) $15 \bmod 9 =$ ______ (g) $19 \bmod 12 =$ ______

(b) $23 \bmod 15 =$ ______ (e) $27 \bmod 9 =$ ______ (h) $31 \bmod 12 =$ ______

(c) $15 \bmod 4 =$ ______ (f) $7 \bmod 12 =$ ______ (i) $47 \bmod 12 =$ ______

Definition 3.13. *For integers a, b, and n, where $n > 0$, we say that a is* **congruent to** b **modulo** n *if n divides $a-b$ (that is, $a-b = kn$ for some integer k). We write this as $a \equiv b \pmod{n}$.*

There are a few other (equivalent) ways of defining congruence modulo n.

- *$a \equiv b \pmod{n}$ iff a and b have the same remainder when divided by n.*
- *$a \equiv b \pmod{n}$ iff $a - b$ is a multiple of n.*

If $a - b \neq kn$ for any integer k, then a is not congruent to b modulo n, and we write this as $a \not\equiv b \pmod{n}$.

Example 3.14. Notice that $21 - 6 = 15 = 3 \cdot 5$, so $21 \equiv 6 \pmod{5}$.

Notice that if $a \equiv b \pmod{n}$ and $0 \leq b < n$, then b is the remainder when a is divided by n.

Example 3.15. Prove that for every integer n, ($n^2 \bmod 4$) is either 0 or 1.

Proof: Since every integer is either even (of the form $2k$) or odd (of the form $2k+1$) we have two possibilities:

$$\begin{aligned} (2k)^2 &= 4k^2 &\equiv 0 \pmod 4, \text{or} \\ (2k+1)^2 &= 4(k^2+k)+1 &\equiv 1 \pmod 4. \end{aligned}$$

Thus, n^2 has remainder 0 or 1 when divided by 4. □

Example 3.16. Prove that the sum of two squares of integers leaves remainder 0, 1 or 2 when divided by 4.

Proof: According to Example 3.15, the squares of integers have remainder 0 or 1 when divided by 4. Thus, when we add two squares, the possible remainders when divided by 4 are 0 $(0+0)$, 1 $(0+1$ or $1+0)$, and 2 $(1+1)$. □

Example 3.17. Prove that 2003 is not the sum of two squares.

Proof: Notice that $2003 \equiv 3 \pmod 4$. Thus, by Example 3.16 we know that 2003 cannot be the sum of two squares. □

The proof of the following is left as an exercise.

Theorem 3.18. *$a \equiv b \pmod n$ iff*[a] *$a \bmod n = b \bmod n$.*

[a] **iff** is shorthand for **if and only if**.

Example 3.19. Since, $1961 \bmod 37 = 0 \neq 4 = 1966 \bmod 37$, we know that $1961 \not\equiv 1966 \pmod{37}$ by Theorem 3.18.

Note: *Our definition of* mod *required that $n > 0$ and $a \geq 0$. However, it is possible to define $a \bmod n$ when a is negative. Unfortunately, there are two possible ways of doing so based on how you define the remainder when the dividend is negative. One possible answer is negative and the other is positive. However, they always differ by n, so computing one from the other is easy.*

Example 3.20. Since $-13 = (-2) * 5 - 3$ and $-13 = (-3) * 5 + 2$, we might consider the remainder of $-13/5$ as either -3 or 2. Thus, $-13 \bmod 5 = -3$ and $-13 \bmod 5 = 2$ both seem like reasonable answers. Fortunately, the two possible answers differ by 5. In fact, you can always obtain the positive possibility by adding n to the negative possibility.

⋆**Exercise 3.21.** Fill in the missing numbers that are congruent to 1 (mod 4) (listed in increasing order)

_______, -11, _______, -3, 1, 5, _______, _______, 17, _______

Note: *When using the* mod *operator in computer programs in situations where the dividend might be negative, it is important to know which definition your programming language/compiler uses.* Java *returns a negative number when the dividend is negative. In* C*, the answer depends on the compiler.*

⋆**Exercise 3.22.** If you write a *C* program that computes $-45 \bmod 4$, what are the two possible answers it might give you?

Answer ______________________________

The next exercise explores a reasonable idea: What if we want the answer to a mod b to always be between 0 and $b-1$, even if a is negative? In other words, we want to force the correct positive answer even if the compiler for the given language might return a negative answer.

⋆**Evaluate 3.23.** Although different programming languages and compilers might return different answers to the computation a mod b when $a < 0$, they always return a value between $-(b-1)$ and $b-1$. Given that fact, give an algorithm that will always return an answer between 0 and $b-1$, regardless of whether or not a is negative. Try to do it without using any conditional statements.

Solution 1: Use $(a \pmod b)+b-1)/2$. Since it always returns a value between $-(b-1)$ and $b-1$ by adding $b-1$ to both sides you get a value between 0 and $2b-2$. You then divide by 2 to hit the target range of a return value that is between 0 and $b-1$ whether the number is positive or negative.

Evaluation ______________________________

Solution 2: Just return the absolute value of a mod b.

Evaluation ______________________________

Solution 3: Use the following:

```
int c = a % b;
if(c<0) {
    return -c;
} else {
    return c;
}
```

Evaluation ______________________________

____

____

Solution 4: Use (a mod b) mod b.

Evaluation ______________________________

____

____

⋆**Exercise 3.24.** Devise a correct solution to the Evaluate 3.23.
Answer:

Definition 3.25. *The* **floor** *of a real number x, written $\lfloor x \rfloor$, is the largest integer that is less than or equal to x. The* **ceiling** *of a real number x, written $\lceil x \rceil$, is the smallest integer that is greater than or equal to x.*

Example 3.26. $\lfloor 4.5 \rfloor = 4$, $\lceil 4.5 \rceil = 5$, $\lfloor 7 \rfloor = \lceil 7 \rceil = 7$.
In general, if n is an integer, then $\lfloor n \rfloor = \lceil n \rceil = n$.

$\star$**Exercise 3.27.** Determine each of the following.

1. $\lfloor 9.9 \rfloor =$ ______
2. $\lceil 9.9 \rceil =$ ______
3. $\lfloor 9.00001 \rfloor =$ ______
4. $\lceil 9.00001 \rceil =$ ______
5. $\lfloor 9 \rfloor =$ ______
6. $\lceil 9 \rceil =$ ______

Theorem 3.28. *Let a be an integer and x be a real number. Then $a \leq x$ if and only if $a \leq \lfloor x \rfloor$.*

Proof: *If $a \leq \lfloor x \rfloor$, then $a \leq \lfloor x \rfloor \leq x$ is clear. On the other hand, assume $a \leq x$. Then a is an integer that is less than or equal to x. Since $\lfloor x \rfloor$ is the largest integer that is less than or equal to x, $a \leq \lfloor x \rfloor$.* $\square$

$\star$**Evaluate 3.29.** Implement an algorithm that will round a real number `x` to the closest integer, but **rounds *down* at .5**. You can *only* use numbers, basic arithmetic (+, −, *, /), and `floor(y)` and/or `ceiling(y)` (which correspond to $\lfloor y \rfloor$ and $\lceil y \rceil$). Don't worry about the data types (i.e. returning either a `double` or an `int` is fine as long as the value stored represents an integer value).

Solution 1: `return floor(x+.49)`.

Evaluation ______________________________

____

Solution 2: `return floor(x+1/2)`.

Evaluation ______________________________

____

Solution 3: `return ceiling(x+.5)`.

Evaluation ______________________________

____

Solution 4: `return ceiling(x-.5)`.

Evaluation ______________________________

____

Corollary 3.30. *Let a, b, and c be integers. Then $a \leq b/c$ if and only if $a \leq \lfloor b/c \rfloor$.*

Proof: *Since b/c is a real number, this is a special case of Theorem 3.28.* □

The floor function is important because in many programming languages, including Java, C, and C++, integer division *truncates*. That is, when you compute n/k for integers n and k, the result is rounded so it is close to zero. That means that if $n, k \geq 0$, n/k rounds down to $\lfloor n/k \rfloor$. But if $n < 0$, n/k rounds *up* to $\lceil n/k \rceil$. So in Java, C, and C++, $3/4 = -3/4 = 0$, $11/5 = 2$ and $-11/5 = -2$, for instance.

⋆**Exercise 3.31.** Compute each of the following, assuming they are expressions in Java, C, or C++.

(a) 9/10 = ______	(e) 15/10 = ______	(i) -5/10 = ______
(b) 10/10 = ______	(f) 19/10 = ______	(j) -10/10 = ______
(c) 11/10 = ______	(g) 20/10 = ______	(k) -15/10 = ______
(d) 14/10 = ______	(h) 90/10 = ______	(l) -20/10 = ______

⋆**Evaluate 3.32.** Let n and m be positive integers with $m > 2$. Assuming integer division truncates, write an algorithm that will compute n/m, but will *round* the result instead of truncating it (round up at or above .5, down below .5). For instance, 5/4 should return 1, but 7/4 should return 2 instead of 1. You may only use basic integer arithmetic, not including the mod operator.

Solution 1: `floor(n/m+0.5)`

Evaluation ______________________________

____

Solution 2: `floor((n/m) + 1/2)`

Evaluation ______________________________

____

Solution 3: `(int) (n/m+0.5)`

Evaluation ______________________________

____

Although the previous example may seem like it is based on an unnecessary restriction, this is taken from a real-world situation. When writing code for an embedded device (e.g. a thermostat or microwave oven), code space is often of great concern. Performing just a single calculation using doubles/floats can add a lot of code since it needs to add certain code to deal with the data type. Sometimes the amount of code added is too much since embedded processors have a lot less space than the processor in your laptop or desktop computer. Because of this, some embedded programmers do everything they can to avoid all non-integer computations in their code when it is possible.

⋆**Exercise 3.33.** Give a correct solution to round-instead-of-truncate problem from the previous example.
Answer:

3.3 `If-then-else` Statements

Definition 3.34. *The* **if-then-else** *control statement has the following syntax:*

```
if(expression) {
    blockA
else {
    blockB
}
```

and evaluates as follows. If `expression` *is true then the statements in* `blockA` *are executed. Otherwise, the statements in* `blockB` *are executed.*

Example 3.35. Write an algorithm that will determine the maximum of two integers. Prove your algorithm is correct.

Solution: The following algorithm will work.

```
int max(int x, int y) {
    if(x >= y) {
        return x;
    } else {
        return y;
    }
}
```

There are three possible cases. If $x > y$, then x is the maximum, and it is returned since the algorithm returns x if $x \geq y$. If $x = y$, then they are both the maximum, so returning either is correct. In this case it returns x, the correct answer. If $x < y$, then y is the maximum and the algorithm returns y, which is the correct answer. In any case it returns the correct answer.

⋆**Exercise 3.36.** Write an algorithm that will determine the maximum of three numbers that uses the algorithm from Example 3.35. Prove that your algorithm is correct.

```
int max(int x, int y, int z) {









}
```

Proof ____

____

____

The previous exercise is an example of something that you are already familiar with: *code reuse*. We *could have* written an algorithm from scratch, but it is much easier to use one that already exists. Not only is the resulting algorithm simpler, it is easier to prove that it is correct since we know that algorithm it uses is correct.

⋆**Exercise 3.37.** Write an algorithm that prints "Hello" if one enters a number between 4 and 6 (inclusive) and "Goodbye" otherwise. Use the function `print(String s)` to print. You are not allowed to use any boolean operators like **and**, **or**, etc.

```
void HelloGoodbye(int x) {









}
```

For simplicity, we will sometimes use `print` to output results and not worry about whitespace (e.g. spaces and newlines). Think of it as being equivalent to Java's `System.out.print(i+" ")` or C++'s `cout<<i<<" "`, or C's `printf("%d ",i)` if you would like.

⋆**Question 3.38.** The solution given for the previous example uses three print statements, with two identical print statements appearing in different places. Is it possible to write the algorithm using only two print statements while maintaining the restriction that you cannot use **and** and **or**? If so, give that version of the algorithm. If not, explain why not.
Answer:

3.4 The `for` loop

Here is the first of the two types of loops we will consider.

Definition 3.39. *The* **for** *loop has the following syntax:*

```
for(initialize;condition;increment) {
    blockA
}
```

where

- `initialize` *is one or more statements that set up the initial conditions and is executed once at the beginning of the loop.*
- `condition` *is the condition that is checked each time through the loop. If* `condition` *is true, the statements in* `blockA` *are executed followed by the code in* `increment`*. This process repeats until* `condition` *is false.*
- `increment` *is code that ensures the loop progresses. Typically* `increment` *is just a simple increment statement, but it can be anything.*

Example 3.40. Write an algorithm that returns $n!$ when given n.

Solution: Here is one possible algorithm.

```
int factorial(int n) {
    if(n==0) {   return 1;
    } else {
        int fact = 1;
        for(int i=1;i<=n;i++) {
            fact = fact*i;
        }
        return fact;
    }
}
```

⋆**Question 3.41.** Does the `factorial` algorithm from Example 3.40 ever do something unexpected? If so, what does it do, when does it do it, and what should be done to fix it?

Answer ______________________________

____

____

⋆**Evaluate 3.42.** Evaluate these algorithms that supposedly compute $n!$ for values of $n > 0$. Don't worry about what they do when $n \leq 0$.

Solution 1:

```
int fact = 1;
for(int i=0;i<=n;i++) {
    fact = fact*i;
}
return fact;
```

Evaluation ______________________________

____

Solution 2:

```
int fact = 1;
for(int i=2;i<=n;i++) {
    fact = fact*i;
}
return fact;
```

Evaluation ______________________________

____

Solution 3:

```
int fact = 1;
for(int i=n;i>0;i--) {
    fact = fact*i;
}
return fact;
```

Evaluation ______________________________

____

Solution 4:

```
int fact = 1;
for(int i=1;i<n;i++) {
    fact = fact*(n-i);
}
return fact;
```

Evaluation ______________________________

____

⋆**Exercise 3.43.** Write an algorithm that will compute x^n, where x is a given real number and n is a given positive integer.

```
double power(double x, int n) {








}
```

3.5 Arrays

Definition 3.44. *An* **array** *is an aggregate of homogeneous types. The* **length** *of the array is the number of entries it has.*

A 1-dimensional array is akin to a mathematical vector. Thus if X is 1-dimensional array of length n then

$$X = (X[0], X[1], \dots, X[n-1]).$$

We will follow the convention of common languages like Java, C, and C++ by indexing the arrays from 0. This means that the last element is $X[n-1]$. We will always declare the length of the array at the beginning of a code fragment by means of a comment.

A 2-dimensional array is akin to a mathematical matrix. Thus if Y is a 2-dimensional array with 2 rows and 3 columns then

$$Y = \begin{bmatrix} Y[0][0] & Y[0][1] & Y[0][2] \\ Y[1][0] & Y[1][1] & Y[1][2] \end{bmatrix}.$$

Example 3.45. Write an algorithm that determines the maximum element of a 1-dimensional array of n integers.

Solution: We declare the first value of the array (the 0-th entry) to be the maximum (a *sentinel value*). Then we successively compare it to other $n-1$ entries. If an entry is found to be larger than it, that entry is declared the maximum.

```
MaxEntry(int[] X, int n) {
    int max = X[0];
    for(int i=1;i<n;i++) {
        if(X[i]>max) {
            max = X[i];
        }
    }
    return max;
}
```

If your primary language is Java, you might wonder why we did not use `X.length` in the previous algorithm. There are two reasons. First, not all languages store the length of an array as part of the array. For examples, C and C++ do not. In these languages you always need to pass the length along with an array. Second, sometimes you want to be able to process only part of an array. Written as we did above, the algorithm will return the maximum of the first n elements of an array. The algorithm works as long as the array has at least n elements.

Note: *If an algorithm has an array and a variable n as parameters, you can probably assume n is the length of the array unless it is otherwise specified.*

Example 3.46. Implement a method that swaps two elements of an array that works in Java and other languages that can't pass by reference.

Solution: Here is a method that swaps two elements of an *integer* array. Except for the type of the parameter and `temp` variable, this works for any data type.

```
swap(int[] X, int a, int b) {
    int temp = X[a];
    X[a]=X[b];
    X[b]=temp;
}
```

I don't want to get into the technical details of pass-by-value versus pass-by-reference since that is really the subject of another course. But briefly, this works because when the array is passed we have access to the individual array elements. Therefore when we change them, they are changed in the original array.

Example 3.47. An array $(X[0], \ldots X[n-1])$ is given. Without introducing another array, put its entries in reverse order.

Solution: Observe that we want to exchange the first and last element, the second and second-to-last element, etc. That is, we want to exchange $X[0] \leftrightarrow X[n-1]$, $X[1] \leftrightarrow X[n-2]$, $\ldots$, $X[k] \leftrightarrow X[n-k-1]$. But what value of k is correct? If we go all the way to $n-1$, the result will be that every element is swapped and then swapped back, so we will accomplish nothing. Hopefully you can see that if we swap elements when $k < n-k-1$, we will swap each element at most once. The "at most once" is because if the array has an odd number of elements, the middle element occurs when $k = n-k-1$, but we can skip it since it doesn't need to be swapped with anything. Notice that $k < n-k-1$ if and only if $2k < n-1$. Since k and n are integers, this is equivalent to $2k \leq n-2$. This is equivalent to $k \leq \lfloor (n-2)/2 \rfloor$ by Corollary 3.30. Thus, we need to swap the elements $0, 1, \ldots, \lfloor (n-2)/2 \rfloor$ with elements $n-1, n-2, \ldots, n-1-\lfloor (n-2)/2 \rfloor = n - \lfloor n/2 \rfloor$, respectively. The following algorithm implements this idea.

```
reverseArray(int[] X, int n) {
    for(int i=0;i<=(n-2)/2;i++) {
        swap(X,i,n-i-1);
    }
}
```

⋆**Question 3.48.** The previous algorithm went until i was $(n-2)/2$, not $\lfloor (n-2)/2 \rfloor$. Why is this O.K.? Does it depend on the language? Explain.

Answer ____

____

⋆**Question 3.49.** Does the following algorithm correctly reverse the elements of an array? Explain.

```
reverseArray(int[] X, int n) {
    for(int i=0;i<n/2;i++) {
        swap(X,i,n-i-1);
    }
}
```

Answer ____

____

Hopefully the previous example helps you realize that you need to be careful when working with arrays. Formulas related to array indices change depending on whether arrays are indexed starting at 0 or 1. In addition, formulas involving the number of elements in an array can be tricky, especially when the formulas relate to partitioning the array into pieces (e.g. into two halves). These can both lead to the so-called "off by one" error that is common in computer science. The next example illustrates these problems, and one way to deal with it.

Example 3.50. Give a formula for the index of the middle element of an array of size n. If there are two middle elements (e.g. n is even), use the first one.

Solution: Clearly the answer should be somewhere close to $n/2$. Unfortunately, if n is odd, $n/2$ isn't an integer. And clearly the answer won't be the same when indexing starting at both 0 and 1. Maybe we should try a few concrete examples.

Let's first assume indexing starts at 1. If $n = 9$, the middle element is the 5th element, which has index $5 = \lceil 9/2 \rceil$. If $n = 10$, the middle element is also the 5th element. Then the index is $5 = 10/2 = \lceil 10/2 \rceil$. Thus the formula $\lceil n/2 \rceil$ should work. You should plug in a few more values to convince yourself that this is correct.

Now let's assume indexing starts at 0. There are a a few equivalent formulas we can come up with. For starters, $\lceil n/2 \rceil - 1$ should work since this is just 1 less than the answer above, and the indices are all shifted by one. But let's come up with a formula from scratch. If $n = 9$, the index of the middle element is $4 = \lfloor 9/2 \rfloor$. If $n = 10$, the index is $4 \neq \lfloor 10/2 \rfloor$. So $\lfloor n/2 \rfloor$ works when n is odd, but not when n is even. This one is not as obvious as it was when we started indexing at 1. With a little thought, you may realize that $\lfloor (n-1)/2 \rfloor$ works.

⋆**Question 3.51.** The previous example seems to suggest that $\lceil n/2 \rceil - 1 = \lfloor (n-1)/2 \rfloor$ for all integers n. Is this correct? Do a few sample computations to try to convince yourself of your answer.

Answer ____

____

Note: *Always be very careful with formulas related to the index of an array. Double-check your logic by plugging in some values to be certain your formula is correct.*

Definition 3.52. *A* **boolean variable** *is a variable that can be either* **true** *or* **false**.

Definition 3.53. *The* **not** *unary operator changes the status of a boolean variable from* **true** *to* **false** *and vice-versa. In Java, C, and C++, the* not *operator is* ! *and it appears before the expression being negated (e.g.* `!x`*).*

The *not* operator is essentially the same thing as the *negation* we discussed earlier. The difference is context—we are applying *not* to a boolean variable, whereas we applied *negation* to a statement.

Example 3.54 (The Locker-Room Problem)**.** A locker room contains n lockers, numbered 1 through n. Initially all doors are open. Person number 1 enters and closes all the doors. Person number 2 enters and opens all the doors whose numbers are multiples of 2. Person number 3 enters and toggles all doors that are multiples of 3. That is, he closes them if they are open and opens them if they are closed. This process continues, with person i toggling each door that is a multiple of i. Write an algorithm to determine which lockers are closed when all n people are done.

Solution: Here is one possible approach. We use a boolean array `Locker` of size $n + 1$ to denote the lockers (we will ignore `Locker[0]`). The value **true** will denote an open locker and the value **false** will denote a closed locker.

```
LockerRoomProblem(boolean[] Locker, int n) {
    // Person 1: Close them all
    for(int i=1;i<=n;i++) {
       Locker[i]=false;
    }
    //People 2 through n: toggle appropriate ones
    for(int j=2;j<=n;j++) {
        for(k=j;k<=n;k++) {
            if(k%j==0) {
                Locker[k] = !Locker[k];
            }
        }
    }
    // Print the results
    print("Closed:");
    for(int l=1;l<=n;l++) {
        if(Locker[l]==false) {
            print(l);
            print(" ");
        }
    }
}
```

3.6 The `while` loop

Definition 3.55. *The* **while** *loop has syntax:*

```
while(condition) {
    blockA
}
```

The statements in `blockA` *will execute as long as* `condition` *evaluates to true.*

Example 3.56. An array X satisfies $X[0] \leq X[1] \leq \cdots \leq X[n-1]$. Write an algorithm that finds the number of entries that are different.

Solution: Here is one possible approach.

```
int differentElements(int[] X, int n) {
    int i = 0;
    int different = 1;
    while(i<n-1) {
        i++;
        if(x[i]!=x[i-1]) {
            different++;
        }
    }
    return different;
}
```

⋆**Exercise 3.57.** What will the following algorithm return for $n = 5$? Trace the algorithm carefully, outlining all your steps.

```
mystery(int n) {
    int x=0;
    int i=1;
    while(n>1) {
        if(n*i>4) {
            x=x+2*n;
        } else {
            x=x+n;
        }
        n=n-2;
        i++;
    }
    return x;
}
```

Answer ____

____

____

Theorem 3.58. *Let $n > 1$ be a positive integer. Either n is prime or n has a prime factor no greater than $\sqrt{n}$.*

Proof: *If n is prime there is nothing to prove. Assume that n is composite. Then n can be written as the product $n = ab$ with $1 < a \leq b$, where a and b are integers. If every prime factor of n were greater than $\sqrt{n}$, then $a > \sqrt{n}$ and $b > \sqrt{n}$. But then $n = ab > \sqrt{n}\sqrt{n} = n$, which is a contradiction. Thus n must have a prime factor no greater than $\sqrt{n}$.* $\square$

Example 3.59. To determine whether 103 is prime we proceed as follows. Observe that $\lfloor \sqrt{103} \rfloor = 10$ (According to Theorem 3.28, we only need concern ourselves with the floor). We now divide 103 by every prime no greater than 10. If one of these primes divides 103, then 103 is not a prime. Otherwise, 103 is a prime. Notice that $103 \bmod 2 = 1$, $103 \bmod 3 = 1$, $103 \bmod 5 = 3$, and $103 \bmod 7 = 5$. Since none of these remainders is 0, 103 is prime.

⋆**Exercise 3.60.** Give a complete proof of whether or not 101 is prime.

Proof ____

____

____

⋆**Exercise 3.61.** Give a complete proof of whether or not 323 is prime.

Proof ____

____

____

Example 3.62. Give an algorithm to determine if a given positive integer n is prime.

Solution: We first deal with a few base cases. If $n = 1$, it is not prime, and if $n = 2$ or $n = 3$ it is prime. Then we determine if n is even, in which case it is not prime. Finally, we loop through all of the odd values, starting with 3 and going to $\sqrt{n}$, determining whether or not n is a multiple of any of them. If so, it is not prime. If we get through all of this, then n has no factors less than or equal to $\sqrt{n}$ which means it must be prime. Here is the algorithm based on this description.

```
boolean isPrime(int n) {
    if(n<=1) {  // Anything less than 2 is not prime.
        return false; }
    if(n==2 || n==3) {  // 2 and 3 are special cases.
        return true; }
    if(n%2==0) {  // Discard even numbers right away.
```

```
            return false;
        } else {
            // Determine if it has any odd factors.
            int i = 1;
            while(i <= sqrt(n)) {
                i = i + 2;
                if(n%i==0) {
                    return false; }
            }
            return true;  // It had no factors.
        }
    }
```

Note: *It should be noted that although this algorithm in Example 3.62 works, it is not very practical for large values of n. In fact, there is no known algorithm that can factor numbers efficiently on a "classical" computer. The most commonly used public-key cryptosystems rely on the assumption that there is no efficient algorithm to factor a number. However, if you have a quantum computer, you are in luck. Shor's algorithm actually* **can** *factor numbers efficiently.*

⋆**Question 3.63.** Why did the algorithm in the previous example deal with even numbers differently?

Answer ____

____

⋆**Exercise 3.64.** Use the fact that integer division truncates to write an algorithm that reverses the digits of a given positive integer. For example, if 123476 is the input, the output should be 674321. You should be able to do it with one extra variable, one `while` loop, one mod operation, one multiplication by 10, one division by 10, and one addition.

```
int reverseDigits(int n) {









}
```

3.7 Problems

Note: *For the remainder of the book, whenever a problem asks for an algorithm, always assume it is asking for the most efficient algorithm you can find. You will likely lose points if your algorithm is not efficient enough.*

Problem 3.1. Let n be a positive integer. Recall that $a \equiv b \pmod{n}$ iff n divides $a - b$ (that is, $a - b = k \cdot n$ for some $k \in \mathbb{Z}$). Use this formal definition to prove each of the following:

(a) $a \equiv a \pmod{n}$. (Reflexive property)

(b) If $a \equiv b \pmod{n}$, then $b \equiv a \pmod{n}$. (Symmetric property)

(c) If $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$, then $a \equiv c \pmod{n}$. (Transitive property)

Problem 3.2. Implement the swap operation for integers without using an additional variable and without using addition or subtraction. (Hint: bit operations)

Problem 3.3. Prove or disprove that the following method correctly computes the maximum of two integers `x` and `y`, assuming that the `minimum` method correctly computes the minimum of `x` and `y`.

```
int maximum(int x, int y) {
    int min = minimum(x,y);
    int max = x + y - min;
    return max;
}
```

Problem 3.4. Give a recursive algorithm that computes $n!$. You can assume $n \geq 0$.

Problem 3.5. What will the following algorithm return for $n = 3$?

```
iCanDuzSomething(int n) {
    int x = 0;
    while(n>0) {
        for(int i=1;i<=n;i++) {
            for(int j=i;j<=n;j++) {
                x = x + i*j;
            }
        }
        n--;
    }
    return x;
}
```

Problem 3.6. Give an algorithm that will round a real number `x` to the closest integer, rounding *up* at .5. You can *only* use `floor(y)`, `ceiling(y)`, basic arithmetic (+, -, *, /) and/or numbers. You *cannot* use anything else, including conditional statements! *Prove that your algorithm is correct.*

Problem 3.7. Recall that Example 3.32 had the conditions that $n > 0$ and $m > 2$. Also recall that you gave a solution to this in Exercise 3.33. Also recall that integer division always truncates *toward zero*, so negative numbers truncate differently than positive ones.

(a) Does your solution work when $m = 2$? Justify your answer with a proof/counterexample.

(b) Does your solution work when $n \leq 0$? Justify your answer with a proof/counterexample.

(c) Give an algorithm that will work for any integer n and any non-zero m. Give examples that demonstrate that your algorithm is correct for the various cases and/or a proof that it always works. Make sure you consider all relevant cases (e.g., when it should round up and down, when n and m are positive/negative). You may only use basic integer arithmetic and conditional statements. You may *not* use *floor*, *ceiling*, *abs* (absolute value), etc. You also may *not* use the *mod* operator since how it works with negative numbers is not the same for every language.

Problem 3.8. Assume you have a function `random(int n)` that returns a random integer between 0 and $n-1$, inclusive. Give code/pseudocode for an algorithm `random(int a, int b)` that returns a random number between a and b, inclusive of both a and b. You may assume that $a < b$ (although in practice, this should be checked). You may only call `random(int n)` once and you may not use conditional statements. *Prove that your algorithm returns an integer in the required range.*

Problem 3.9. Assume you have a function `random()` that returns a non-negative random integer. Give code/pseudocode for an algorithm `random(int a, int b)` that returns a random integer between a and b, inclusive of both a and b. Each possible number generated should occur with approximately the same probability. You may assume that a and b are both positive and that $a < b$ (although in practice, this should be checked). You may use only basic integer arithmetic (including the mod operator) and you may only call `random()` once. You may not use loops, conditional statements, *floor*, *ceiling*, *abs* (absolute value), etc. *Prove that your algorithm returns an integer in the required range.*

Problem 3.10. The following method is a simplified version of a method that might be used to implement a hash table or in a cryptographic system. Assume that for one particular use the number returned by this function has to have the opposite parity (even/odd) of the parameter. For instance, `hash_it(4)` returns 49 which has the opposite parity of 4, so it works for 4. Prove or disprove that this function always returns a value of opposite parity of the parameter.

```
int hash_it(int x) {
    return x*x+6*x+9;
}
```

Problem 3.11. Give an algorithm that computes all of the primes that are less than or equal to n. For simplicity, you can just print all of the prime numbers up to n. Your algorithm should be as efficient as possible. One approach is to modify the algorithm from Example 3.62 by using an array to make it more efficient.

Problem 3.12. Prove or disprove that the following method computes the absolute value of `x`. For simplicity, assume that all of the calculations are performed with perfect precision. You may use the fact that $\sqrt{x^2} = x$ when $x \geq 0$ if it will help.

```
double absoluteValue(double x) {
    double square = x*x;
    double answer = sqrt(square);
    return answer;
}
```

Problem 3.13. Prove or disprove that the following method computes the absolute value of `x`. For simplicity, assume that all of the calculations are performed with perfect precision. You may use the fact that $(\sqrt{x})^2 = x$ when $x \geq 0$ if it will help.

```
double absoluteValue(double x) {
    double root = sqrt(x);
    double answer = root*root;
    return answer;
}
```

Problem 3.14. Problems 3.12 and 3.13 both assumed that "all of the calculations are performed with perfect precision". Is that a realistic assumption? Give an example of an input for which the each algorithm will work properly. Then give an example of an input for which each algorithm will *not* work properly. You can implement and run the algorithms to do some testing if you wish.

Problem 3.15. The following method is supposed to do some computations on a positive number that result in getting the original number back. Prove or disprove that this method always returns the *exact* value that was passed in. Unlike in the previous problems, here you should assume that although a `double` stores a real number as accurately as possible, it uses only a fixed amount of space. Thus, a `double` is unable to store the exact value of any irrational number–it instead stores an approximation.

```
double returnTheParameterUnmodified(double x) {
    double a = sqrt(x);
    double b = a*a;
    return b;
}
```

Problem 3.16. Prove or disprove that the algorithm from Example 3.8 actually *does* work properly with integer data types stored using 2's complement.[1] You may restrict to 8-bit numbers if it will help you think about it more clearly–a proof/counterexample for 8-bit number can easily be modified to work for 32- or 64-bit numbers. (Hint: If it doesn't work, what sort of numbers might it fail on?)

Problem 3.17. Use the first definition of congruence modulo n given in Definition 3.13 to prove Theorem 3.18. (Note: This is an if and only if proof, so you need to prove both ways.)

[1]When we say "works," we mean for *all* possible values of x and y.

Chapter 4

Logic

4.1 Propositional Logic

Definition 4.1. *A* **boolean proposition** *(or simply* **proposition***) is a statement which is either* **true** *or* **false** *(sometimes abbreviated as* **T** *or* **F***). We call this the* **truth value** *of the proposition.*

Whether the statement is *obviously* true or false does not enter in the definition. One only needs to know that its certainty can be established.

Example 4.2. The following are propositions and their truth values, if known:

(a) $7^2 = 49$. (**true**)

(b) $5 > 6$. (**false**)

(c) If p is a prime then p is odd. (**false**)

(d) There exists infinitely many primes which are the sum of a square and 1. (unknown)

(e) There is a God. (unknown)

(f) There is a dog. (**true**)

(g) I am the Pope. (**false**)

(h) Every prime that leaves remainder 1 when divided by 4 is the sum of two squares. (**true**)

(i) Every even integer greater than 6 is the sum of two distinct primes. (unknown)

⋆**Exercise 4.3.** Give the truth value of each of the following statements.

(a) __________ $0 = 1$.

(b) __________ 17 is an integer.

(c) __________ "Psych" is a TV show that aired on the *USA network*.

(d) __________ In 1999, it was possible to buy a red Swingline stapler.

Example 4.4. The following are not propositions, since it is impossible to assign a **true** or **false** value to them.

(a) Whenever I shampoo my camel.

(b) Sit on a potato pan, Otis!

(c) What I am is what I am, are you what you are or what?

(d) $x = x + 1$.

(e) This sentence is false.

⋆**Exercise 4.5.** For each of the following statements, state whether it is true, false, or not a proposition.

(a) __________ i can has cheezburger?

(b) __________ "Psych" was one of the best shows on TV when it was on the air.

(c) __________ I know, right?

(d) __________ This is a proposition.

(e) __________ This is not a proposition.

4.1.1 Compound Propositions

Definition 4.6. *A* **logical operator** *is used to combine one or more propositions to form a new one. A proposition formed in this way is called a* **compound proposition**. *We call the propositions used to form a compound proposition* **variables** *for reasons that should become evident shortly.*

Next we will discuss the most common logical operators. Some of these will be familiar to you. When you learned about Boolean expressions in your programming courses, you probably saw *NOT* (e.g. `if( !list.isEmpty() )`), *OR* (e.g. `if( x>0 || y>0 )`), and *AND* (e.g. `if( list.size() > 0 && list.get(0) > 1 )`). The notation we use will be different, however.

This is because the symbols you are familiar with are specific choices made by whoever created the programming language(s) you learned. Here we will use standard mathematical notation for the operators.

For each of the following definitions, assume p and q are propositions.

Definition 4.7. *The* **negation** *(or* **NOT***) of p, denoted by* $\neg \boldsymbol{p}$ *is the proposition "***it is not the case that $\boldsymbol{p}$***". $\neg \boldsymbol{p}$ is true when $\boldsymbol{p}$ is false, and vice-versa. Other notations include $\overline{\boldsymbol{p}}$, $\sim \boldsymbol{p}$, and $!\boldsymbol{p}$. Many programming languages use the last one.*

Example 4.8. If p is the proposition "$x < 0$", then $\neg p$ is the proposition "It is not the case that $x < 0$," or "$x \geq 0$."

⋆**Fill in the details 4.9.** Let p be the proposition "I am learning discrete mathematics."

Then $\neg p$ is the proposition ____.

The truth value of $\neg p$ is ______________.

⋆**Exercise 4.10.** Consider the statement "This is not a proposition."

(a) Use the fact that "This is a proposition" is a proposition to prove that "This is not a proposition" is a proposition. Then prove that its truth value is false.

Proof ____

____

____

(b) Use a contradiction proof to prove that "This is not a proposition" is a proposition. Then prove that its truth value is false.

Proof ____

____

____

⋆**Exercise 4.11.** You need a program to execute some code only if the size of a list is not 0. The variable is named `list`, and its size is `list.size()`. Give the expression that should go in the `if` statement. In fact, give two different expressions that will work.

1. ______________________________

2. ______________________________

Definition 4.12. *The* **conjunction** *(or* **AND***) of p and q, denoted by $\boldsymbol{p \wedge q}$, is the proposition "**p and q**". The conjunction of $\boldsymbol{p}$ and $\boldsymbol{q}$ is true when $\boldsymbol{p}$ and $\boldsymbol{q}$ are both true and false otherwise. Many programming languages use* `&&` *for conjunction.*

Example 4.13. Let p be the proposition "$x > 0$" and q be the proposition "$x < 10$." Then $p \wedge q$ is the proposition "$x > 0$ and $x < 10$," or "$0 < x < 10$." In a Java/C/C++ program, it would be "`x>0 && x<10`."

Example 4.14. Let p be the proposition "$x < 0$" and q be the proposition "$x > 10$." Then $p \wedge q$ is the proposition "$x < 0$ and $x > 10$." Notice that $p \wedge q$ is always false since if $x < 0$, clearly $x \not> 10$. But don't confuse the *proposition* with its *truth value*. When you see the statement '$p \wedge q$ is "$x < 0$ and $x > 10$"' and '$p \wedge q$ is false,' these are saying two different things. The first one is telling us what the proposition is. The second one is telling us its truth value. '$p \wedge q$ is false' is just a shorthand for saying '$p \wedge q$ has truth value false.'

⋆**Fill in the details 4.15.** If p is the proposition "I like cake," and q is the proposition "I like ice cream," then $p \wedge q$ is the proposition ______________________________.

Example 4.16. Write a code fragment that determines whether or not three numbers can be the lengths of the sides of a triangle.

Solution: Let a, b, and c be the numbers. For simplicity, let's assume they are integers. First we must have $a > 0$, $b > 0$, and $c > 0$. Also, the sum of any two of them must be larger than the third in order to form a triangle. More specifically, we need $a + b > c$, $b + c > a$, and $c + a > b$. Since we need all of these to be true, this leads to the following algorithm.

```
IsItATriangle(int a, int b, int c) {
    if(a>0 && b>0 && c>0 && a+b>c && b+c>a && a+c>b) {
        return true;
    } else { return false; }
}
```

Definition 4.17. *The* **disjunction** *(or* **OR***) of p and q, denoted by $\boldsymbol{p} \vee \boldsymbol{q}$, is the proposition "**p or q**". The disjunction of $\boldsymbol{p}$ and $\boldsymbol{q}$ is false when both $\boldsymbol{p}$ and $\boldsymbol{q}$ are false and true otherwise. Put another way, if $\boldsymbol{p}$ is true, $\boldsymbol{q}$ is true, or both are true, the disjunction is true. Many programming languages use `||` for disjunction.*

Example 4.18. Let p be the proposition "$x < 5$" and q be the proposition "$x > 15$." Then $p \vee q$ is the proposition "$x < 5$ or $x > 15$." In a Java/C/C++ program, it would be "`x<5 || x>15`."

⋆**Fill in the details 4.19.** Let p be the proposition "$x > 0$" and q be the proposition "$x < 10$." Then $p \vee q$ is the proposition ________________________________.

Notice that $p \vee q$ is always ____________ since it is ____________ if $x > 0$, and if $x \not> 0$, then clearly ________________, so it is ____________ then as well.

⋆**Exercise 4.20.** Let p be "you must be at least 48 inches tall to ride the roller coaster," and q be "you must be at least 18 years old to ride the roller coaster." Express each of the following propositions in English.

1. $\neg p$ is ____

2. $p \vee q$ is ____

3. $p \wedge q$ is ____

⋆**Exercise 4.21.** Give an algorithm that will return true if an array of integers either starts or ends with a 0, or false otherwise. Assume array indexing starts at 0 and that the array is of length n. Use only one conditional statement. Be sure to deal with the possibility of an empty array.

```
boolean startsOrEndsWithZero(int[] a, int n) {



}
```

⋆**Question 4.22.** Does the solution given for the previous exercise properly deal with arrays of size 0 and 1? Prove it.

Answer ______________________________

____

____

____

Definition 4.23. *The* **exclusive or** *(or* **XOR***) of p and q, denoted by $\boldsymbol{p \oplus q}$, is the proposition "***p is true or q is true, but not both***". The exclusive or of* $\boldsymbol{p}$ *and* $\boldsymbol{q}$ *is true when exactly one of* $\boldsymbol{p}$ *or* $\boldsymbol{q}$ *is true. Put another way, the exclusive or of* $\boldsymbol{p}$ *and* $\boldsymbol{q}$ *is true iff* $\boldsymbol{p}$ *and* $\boldsymbol{q}$ *have different truth values.*

Example 4.24. Let p be the proposition "$x > 10$" and q be the proposition "$x < 20$." Then $p \oplus q$ is the proposition "$x > 10$ or $x < 20$, but not both."

Note: *Notice that $\vee$ is an* **inclusive or***, meaning that it is true if both are true, whereas $\oplus$ is an* **exclusive or***, meaning it is false if both are true. The difference between $\vee$ and $\oplus$ is complicated by the fact that in English, the word "or" to can mean either of these depending on context. For instance, if your mother tells you "you can have cake or ice cream" she is likely using the exclusive or, whereas a prerequisite of "Math 110 or demonstrated competency with algebra" clearly has the inclusive or in mind.*

⋆**Exercise 4.25.** For each of the following, is the *or* inclusive or exclusive? Answer **OR** or **XOR** for each.

(a) ____________ The special includes your choice of a salad or fries.

(b) ____________ The list is empty or the first element is zero.

(c) ____________ The first list is empty or the second list is empty.

(d) ____________ You need to take probability or statistics before taking this class.

(e) ____________ You can get credit for either Math 111 or Math 222.

⋆**Exercise 4.26.** Let p be "list 1 is empty" and q be "list 2 is empty." Explain the difference in meaning between $p \vee q$ and $p \oplus q$.

Answer ____

⋆**Question 4.27.** Let p be the proposition "$x < 5$" and q be the proposition "$x > 15$."

(a) Do the statements $p \vee q$ and $p \oplus q$ mean the same thing? Explain.

Answer ____

(b) Practically speaking, are $p \vee q$ and $p \oplus q$ the same? Explain.

Answer ____

XOR is not used as often as AND and OR in logical expressions in programs. Some languages have an XOR operator and some do not. The issue gets blurry because some languages, like Java, have an explicit Boolean type, while others, like C and C++, do not. All of these languages have a *bitwise XOR* operator, but this is not the same thing as a *logical XOR* operator. We will return to this topic later. In the next section we will see how to implement $\oplus$ using $\vee$, $\wedge$, and $\neg$.

Definition 4.28. *The* **conditional statement** *(or* **implies***) involving p and q, denoted by* $\boldsymbol{p \to q}$*, is the proposition "***if $\boldsymbol{p}$, then $\boldsymbol{q}$***". It is false when* $\boldsymbol{p}$ *is true and* $\boldsymbol{q}$ *is false, and true otherwise. In the statement* $\boldsymbol{p \to q}$*, we call* $\boldsymbol{p}$ *the* **premise** *(or* **hypothesis** *or* **antecedent***) and* $\boldsymbol{q}$ *the* **conclusion** *(or* **consequence***).*

Example 4.29. Let p be "you earn 90% in the class," and q be "you will receive an A." Then $p \to q$ is the proposition "If you earn 90% in the class, then you will receive an A."

⋆**Question 4.30.** Assume that the proposition "If you earn 90% in this class, then you will receive an A" is true.

(a) What grade will you get if you earn 90%? Explain.

Answer ____

____

(b) If you receive an A, did you earn 90%? Explain.

Answer ____

____

(c) If you don't earn 90%, does that mean you didn't get an A? Explain.

Answer ____

____

Note: *The* **conditional** *operator is by far the one that is the most difficult to get a handle on for at least two reasons. First, the conditional statement $p \to q$ is not saying anything about p or q by themselves. It is only saying that if p is true, then q has to also be true. It doesn't say anything about the case that p is not true. This brings us to the second reason. Should $F \to T$ be true or false? Although it seems counterintuitive to some, it should be true. Again, $p \to q$ is telling us about the value of q when p is true (i.e., if p is true, the q* must be *true). What does it tell us if p is false? Nothing. As strange as it might seem, when p is false, the whole statement is true regardless of the truth value of q.*

If in the end you are still confused, you can (and should) simply fall back on the formal definition: $\boldsymbol{p \to q}$ **is false when** $\boldsymbol{p}$ **is true and** $\boldsymbol{q}$ **is false, and is true otherwise.** *In other words, if interpreting $p \to q$ as the English sentence "p implies q" is more harmful than helpful in understanding the concept, don't worry about why it doesn't make sense and just* *remember the definition.*[a]

[a]In mathematics, one tries to define things so they make sense immediately. Sometimes this is not possible (if the concept is very complicated and/or it just doesn't relate to something that is familiar). Sometimes a term or concept is defined poorly but because of prior use the definition sticks. Sometimes it makes perfect sense to some people and not to others, probably based on each person's background. I think this last possibility may be to blame in this case.

Definition 4.31. *The* **biconditional statement** *involving p and q, denoted by* $\boldsymbol{p \leftrightarrow q}$*, is the proposition "***$\boldsymbol{p}$ if and only if $\boldsymbol{q}$***" (or abbreviated as "***$\boldsymbol{p}$ iff $\boldsymbol{q}$***"). It is true when* $\boldsymbol{p}$ *and* $\boldsymbol{q}$ *have the same truth value, and false otherwise.*

Example 4.32. Let p be "you earn 90% in this class," and q be "you receive an A in this class." Then $p \leftrightarrow q$ is the proposition "You earn 90% in this class if and only if you receive an A."

⋆**Question 4.33.** Assume that the proposition "You will receive an A in the course if and only if you earn 90%" is true.

(a) What grade will you get if you earn 90%?

Answer ______________________________

(b) If you receive an A, did you earn 90%?

Answer ______________________________

(c) If you don't earn 90%, does that mean you didn't get an A?

Answer ______________________________

Now let's bring all of these operations together with a few more examples.

Example 4.34. Let a be the proposition "I will eat my socks," b be "It is snowing," and c be "I will go jogging." Here are some compound propositions involving a, b, and c, written using these variables and operators and in English.

With Variables/Operators	In English
$(b \vee \neg b) \to c$	Whether or not it is snowing, I will go jogging.
$b \to \neg c$	If it is snowing, I will not go jogging.
$b \to (a \wedge \neg c)$	If it is snowing, I will eat my socks, but I will not go jogging.
$a \leftrightarrow c$	When I eat my socks I go jogging, and when I go jogging I eat my socks. *or* I eat my socks if and only if I go jogging.

⋆**Fill in the details 4.35.** Let p be the proposition "*Iron Man* is on TV," q be "I will watch *Iron Man*," and r be "I own *Iron Man* on DVD." Fill in the missing information in the following table.

With Variables/Operators	In English
$p \to q$	
	If I don't own *Iron Man* on DVD and it is on TV, I will watch it.
$p \wedge r \wedge \neg q$	
	I will watch *Iron Man* every time it is on TV, and that is the only time I watch it.
	I will watch *Iron Man* if I own the DVD.

4.1.2 Truth Tables

Sometimes we will find it useful to think of compound propositions in terms of *truth tables*.

Definition 4.36. *A* **truth table** *is a table that shows the truth value of a compound proposition for all possible combinations of truth assignments to the variables in the proposition. If there are n variables, the truth table will have 2^n rows.*

The truth table for $\neg$ is given in Table 4.1 and the truth tables for all of the other operators we just defined are given in Table 4.2. In the latter table, the first two columns are the possible values of the two variables, and the last 5 columns are the values for each of the two-variable compound propositions we just defined for the given inputs.

p	$\neg p$
T	F
F	T

Table 4.1: Truth table for $\neg$

p	q	$(p \wedge q)$	$(p \vee q)$	$p \oplus q$	$(p \to q)$	$(p \leftrightarrow q)$
T	T	T	T	F	T	T
T	F	F	T	T	F	F
F	T	F	T	T	T	F
F	F	F	F	F	T	T

Table 4.2: Truth tables for the two-variable operators

Example 4.37. Construct the truth table of the proposition $a \vee (\neg b \wedge c)$.

Solution: Since there are three variables, the truth table will have $2^3 = 8$ rows. Here is the truth table, with several helpful intermediate columns.

a	b	c	$\neg b$	$\neg b \wedge c$	$a \vee (\neg b \wedge c)$
T	T	T	F	F	T
T	T	F	F	F	T
T	F	T	T	T	T
T	F	F	T	F	T
F	T	T	F	F	F
F	T	F	F	F	F
F	F	T	T	T	T
F	F	F	T	F	F

Note: *Notice that there are several columns in the truth table besides the columns for the variables and the column for the proposition we are interested in. These are "helper" or "intermediate" columns (those are not official definitions). Their purpose is simply to help us compute the final column more easily and without (hopefully) making any mistakes.*

⋆**Exercise 4.38.** Construct the truth table for $(p \to q) \wedge q$.

p	q	$p \to q$	$(p \to q) \wedge q$
T	T		
T	F		
F	T		
F	F		

Note: *As long as all possible values of the variables are included, the order of the rows of a truth table does not matter. However, by convention one of two orderings is usually used. Since there is an interesting connection to the binary representation of numbers, let's take a closer look at this connection in the next example.*

Example 4.39 (Ordering the rows of a Truth Table)**.** Notice that the values of the variables can be thought of as the index of the row. So if a proposition involves two variables, the values in the first two columns are used as a sort of index. We can order the rows by assigning a number to each row based on the values in these columns. The order used here essentially computes an index as follows: For the "index" columns, think of each T as a 0 and each F as a 1. Now treat the numbers in these columns as binary numbers and order the rows appropriately. For instance, if there are three variables, we can think of it as shown in the following table.

a	b	c				index
T	T	T	0	0	0	0
T	T	F	0	0	1	1
T	F	T	0	1	0	2
T	F	F	0	1	1	3
F	T	T	1	0	0	4
F	T	F	1	0	1	5
F	F	T	1	1	0	6
F	F	F	1	1	1	7

This is the ordering you should follow so that you can easily check your answers with those in the solutions. It also makes grading your homework easier.

The other common ordering does the same thing, but maps T to 1 and F to 0.

There is also a way of thinking about this recursively. That is, given an ordering for a table with n variables, we can compute an ordering for a table with $n+1$ variables. It works as follows: Make two copies of the columns corresponding to the variables, appending a T to the beginning of the first copy, and an F to the beginning of the second copy.

⋆**Exercise 4.40.** Construct the truth table of the proposition $(a \vee \neg b) \wedge c$. You're on your own this time to supply all of the details.

4.1.3 Precedence Rules

Consider the compound proposition $a \vee \neg b \wedge c$. Should this be interpreted as $a \vee (\neg b \wedge c)$, $(a \vee \neg b) \wedge c$, or even possibly $a \vee \neg(b \wedge c)$? Does it even matter? You already know that $3+(4*5) \neq (3+4)*5$, so it should not surprise you that where you put the parentheses in logical expressions matters, too. In fact, Example 4.37 gives the truth table for one of these and you just computed the truth table for another one in Exercise 4.40. If you compare them, you will see that they are not the same. The third interpretation is also different from both of these.

To correctly interpret compound propositions, the operators have an *order of precedence*. The order is $\neg$, $\wedge$, $\oplus$, $\vee$, $\rightarrow$, and $\leftrightarrow$. Also, $\neg$ has right-to-left associativity, all other operators listed have left-to-right associativity. Based on these rules, the correct way to interpret $a \vee \neg b \wedge c$ is $a \vee ((\neg b) \wedge c)$.

It is important to know the precedence rules for the logical operators (or at least be able to look it up) so you can properly interpret complex logical expressions. However, I generally prefer to always use enough parentheses to make it immediately clear, especially when I am writing code. It isn't difficult to remember that $\neg$ is first (that is, it always applies to what is immediately after it) so sometimes I don't use parentheses for it.

Example 4.41. According to the precedence rules, $\neg a \rightarrow a \vee b$ should be interpreted as $(\neg a) \rightarrow (a \vee b)$.

Example 4.42. According to the precedence rules, $a \wedge \neg b \rightarrow c$ should be interpreted as $(a \wedge (\neg b)) \rightarrow c$.

⋆**Exercise 4.43.** According to the precedence rules, how should $a \wedge b \vee c$ be interpreted?

Answer ______________________________

⋆**Question 4.44.** Are $(a \wedge b) \vee c$ and $a \wedge (b \vee c)$ equivalent? Prove your answer.

Answer ______________________________

⋆**Evaluate 4.45.** According to the associativity rules, how should $a \rightarrow b \rightarrow c$ be interpreted?

Solution: It should be interpreted as $(a \rightarrow b) \rightarrow c$. However, $a \rightarrow (b \rightarrow c)$ is equivalent, so it really doesn't matter.

Evaluation ______________________________

4.2 Propositional Equivalence

We have already informally discussed two propositions being *equivalent*. In this section, we will formally develop the notion of what it means for two propositions to be *equivalent* (or, more formally, *logically equivalent*). We will also provide you with a list of the most important logical equivalences, along with some examples of some that aren't necessarily as important, but make interesting examples. But first, we need some new terminology.

Definition 4.46. *A proposition that is always true is called a* **tautology**. *One that is always false is a* **contradiction**. *Finally, one that is neither of these is called a* **contingency**.

Example 4.47. Assume that x is a real number.

(a) The proposition "$x < 0$" is a contingency since its truth depends on the value of x.

(b) The proposition "$x^2 < 0$" is a contradiction since it is false no matter what x is.

(c) The proposition "$x^2 \geq 0$" is a tautology since it is true no matter what x is.

⋆**Fill in the details 4.48.** State whether each of the following propositions is a tautology, contradiction, or contingency. Give a brief justification.

(a) $p \vee \neg p$ is a ____________ since either p or $\neg p$ has to be true.

(b) $p \wedge \neg p$ is a ____________ since ____.

(c) $p \vee q$ is a ____________ since ____.

To prove something is a tautology, one must prove that it is always true. One way to do this is to show that the proposition is true for every row of the truth table. Another way is to argue (without using a truth table) that the proposition is always true, often using a proof by cases.

Example 4.49. Prove that $p \vee \neg p$ is a tautology.
Here are several proofs.

Proof 1: Since every row in the following truth table for $p \vee \neg p$ is T, it is a tautology.

p	$\neg p$	$p \vee \neg p$
T	F	T
F	T	T

Proof 2: By definition of disjunction, if p is true, then $p \vee \neg p$ is true. On the other hand, if p is false, $\neg p$ is true. In this case, $p \vee \neg p$ is still true, again by definition of disjunction. Since $p \vee \neg p$ is true regardless of the value of p, it is a tautology.

⋆**Evaluate 4.50.** Prove that $[p \wedge (p \to q)] \to q$ is a tautology.

Proof 1:

p	q	$p \to q$	$p \wedge (p \to q)$	$p \wedge (p \to q) \to q$
T	T	T	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	T

Evaluation ______________________________

____

Proof 2: One way to show that $p \wedge (p \to q) \to q$ is indeed a tautology is by filling out a truth table, as follows:

p	q	$p \to q$	$p \wedge (p \to q)$	$p \wedge (p \to q) \to q$
T	T	T	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	T

Since they all return true for $p \wedge (p \to q) \to q$, this proves that it is a tautology.

Evaluation ______________________________

____

____

Proof 3: One way to prove that this is a tautology is to make a couple of assumptions. First, since we know that for any statement $x \to y$ where y is true, then x can be either true or false. So let us assume that q is false for this case. From the left side of the statement, if p is true, we would have true and (true implies false), which is false, thus we would have false implies false, which is true, and if p is false, then we would have false and (false implies true), which comes out false. So in both cases where q is false, the statement equals out to false implies false, which is true, thus all four cases are true, thereby proving that $p \wedge (p \to q) \to q$ is a tautology.

Evaluation ______________________________

____

____

Proof 4: Since an implication can only be false when the premise is true and the conclusion is false, we only need to prove that this can't happen. So let's assume that $p \wedge (p \to q)$ is true but that q is false. Since $p \wedge (p \to q)$ is true, p is true and $p \to q$ is true (by definition of conjunction). But if p is true and q is false, $p \to q$ is false. This is a contradiction, so it must be the case that our assumption that $p \wedge (p \to q)$ is true but that q is false is incorrect. Since that was the only possible way for $p \wedge (p \to q) \to q$ to be false, it cannot be false. Therefore it is a tautology.

Evaluation ____

Proof 5: Because 'merica.

Evaluation ____

Now we are ready to move on to the main topic of this section.

Definition 4.51. *Let p and q be propositions. Then p and q are said to be* **logically equivalent** *if $p \leftrightarrow q$ is a tautology. An alternative (but equivalent) definition is that p and q are equivalent if they have the same truth table. That is, if they have the same truth value for all assignments of truth values to the variables.*

When p and q are equivalent, we write $\boldsymbol{p = q}$. An alternative notation is $\boldsymbol{p \equiv q}$.

Note: *$p = q$ is* **not** *a compound proposition. Rather it is a statement about the relationship between two propositions.*

There are many *logical equivalences* (or *identities/rules/laws*) that come in handy when working with compound propositions. Many of them (e.g. commutative, associative, distributive) will resemble the arithmetic laws you learned in grade school. Others are very different. The most common ones are given in Table 4.3.

We will provide proofs of some of these so you can get the hang of how to prove propositions are equivalent. One method is to demonstrate that the propositions have the same truth tables. That is, they have the same value on every row of the truth table. But just drawing a truth table isn't enough. A statement like "since p and q have the same truth table, $p = q$" is necessary to make a connection between the truth table and the equivalence of the propositions. Let's see a few examples.

Example 4.52. Prove the double negation law: $\neg(\neg p) = p$.

Proof: The following is the truth table for p and $\neg(\neg p)$.

p	$\neg p$	$\neg(\neg p)$
T	F	T
F	T	F

Since the entries for both p and $\neg(\neg p)$ are the same for every row, $\neg(\neg p) = p$. □

The two versions of De Morgan's Law are among the most important propositional equivalences for computer scientists. It is easy to make a mistake when trying to simplify expressions conditional statements, and a solid understanding of De Morgan's Laws goes a long way. In light of this, let's take a look at them next.

Example 4.53. Prove the first version of DeMorgan's Law: $\neg(p \vee q) = \neg p \wedge \neg q$

Proof: We can prove this by showing that in each case, both expression have the same truth table. Below is the truth table for $\neg(p \vee q)$ and $\neg p \wedge \neg q$ (the gray columns).

p	q	$p \vee q$	$\neg(p \vee q)$	$\neg p$	$\neg q$	$\neg p \wedge \neg q$
T	T	T	F	F	F	F
T	F	T	F	F	T	F
F	T	T	F	T	F	F
F	F	F	T	T	T	T

Since they are the same for every row of the table, $\neg(p \vee q) = \neg p \wedge \neg q$.

□

Name	*Equivalence*
commutativity	$p \vee q = q \vee p$ $p \wedge q = q \wedge p$
associativity	$p \vee (q \vee r) = (p \vee q) \vee r$ $p \wedge (q \wedge r) = (p \wedge q) \wedge r$
distributive	$p \wedge (q \vee r) = (p \wedge q) \vee (p \wedge r)$ $p \vee (q \wedge r) = (p \vee q) \wedge (p \vee r)$
identity	$p \vee F = p$ $p \wedge T = p$
negation	$p \vee \neg p = T$ $p \wedge \neg p = F$
domination	$p \vee T = T$ $p \wedge F = F$
idempotent	$p \vee p = p$ $p \wedge p = p$
double negation	$\neg(\neg p) = p$
DeMorgan's	$\neg(p \vee q) = \neg p \wedge \neg q$ $\neg(p \wedge q) = \neg p \vee \neg q$
absorption	$p \vee (p \wedge q) = p$ $p \wedge (p \vee q) = p$

Table 4.3: Common Logical Equivalences

⋆**Exercise 4.54.** Prove the second version of De Morgan's Law: $\neg(p \wedge q) = \neg p \vee \neg q$.

Proof ____

____

p	q	$p \wedge q$	$\neg(p \wedge q)$	$\neg p$	$\neg q$	$\neg p \vee \neg q$
T	T					
T	F					
F	T					
F	F					

____

Truth tables aren't the only way to prove that two propositions are equivalent. You can also use other equivalences. Let's see an example.

⋆**Fill in the details 4.55.** Prove the idempotent laws ($p \vee p = p$ and $p \wedge p = p$) by using the other equivalences.

Proof: We have

$$\begin{aligned} p &= p \vee F && \text{(by identity)} \\ &= p \vee (p \wedge \neg p) && \text{(by \_\_\_\_\_\_\_\_\_\_)} \\ &= (p \vee p) \wedge (p \vee \neg p) && \text{(by \_\_\_\_\_\_\_\_\_\_)} \\ &= (p \vee p) \wedge T && \text{(by negation)} \\ &= \_\_\_\_\_\_\_\_\_\_\_\_\_\_ && \text{(by identity)} \end{aligned}$$

Thus, $p \vee p = p$. Similarly,

$$\begin{aligned} p &= \_\_\_\_\_\_\_\_\_\_\_\_\_\_ && \text{(by identity)} \\ &= \_\_\_\_\_\_\_\_\_\_\_\_\_\_ && \text{(by negation)} \\ &= \_\_\_\_\_\_\_\_\_\_\_\_\_\_ && \text{(by distributive)} \\ &= \_\_\_\_\_\_\_\_\_\_\_\_\_\_ && \text{(by negation)} \\ &= p \wedge p && \text{(by \_\_\_\_\_\_\_\_\_\_)} \end{aligned}$$

Thus, ______________________. ☐

Although it is helpful to specifically state which rules are being used at every step, it isn't always required.

Example 4.56. Prove that $(p \wedge q) \vee (p \wedge \neg q) = p$.

Proof: It is not too difficult to see that

$$(p \wedge q) \vee (p \wedge \neg q) = p \wedge (q \vee \neg q) = p \wedge T = p.$$

□

⋆**Exercise 4.57.** Use the other equivalences (not a truth table) to prove the Absorption laws.

(a) Prove that $p \vee (p \wedge q) = p$.
Proof:

(b) Prove that $p \wedge (p \vee q) = p$.
Proof:

One use of propositional equivalences is to simplify logical expressions.

Example 4.58. Simplify $\neg(p \vee \neg q)$.

Solution: Using DeMorgan's Law and double negation, we can see that

$$\neg(p \vee \neg q) = \neg p \wedge \neg(\neg q) = \neg p \wedge q.$$

Of course, this also applies to simplifying conditional expressions in code.

Example 4.59. Simplify the following code as much as possible.

```
if ( !(a==null || a.size()<=0) ) {
    a.clear();
}
```

Solution: First, notice that by DeMorgan's Law, `!(a==null || a.size()<=0)` is equivalent to `!(a==null) && !(a.size()<=0)`. Simplifying a bit more, we get `a!=null && a.size()>0`. Thus, the code becomes:

```
if (a!=null && a.size()>0) {
    a.clear();
}
```

This may not look much simpler, but it is much easier to understand.

This simplification can also be done by defining p to be `a==null` and q to be `a.size()<=0`. Then the expression is $\neg(p \vee q)$. Applying De Morgan's Law, this is the same as $\neg p \wedge \neg q$, which we translate back to `!(a==null) && !(a.size()<=0)` and simplify as in the final step above.

As the previous example demonstrates, you can apply the rules to propositions in various form. Sometimes it is useful to explicitly define p and q (and sometimes r) and write expressions using formal mathematical notation, but at other times it is just as easy to apply the rules the the expressions as they are. In the previous example, we didn't gain that much by defining p and q. But with more complicated expressions it certainly can be helpful.

Note: *A common mistake is to forget to use De Morgan's law when dealing with negation. For instance, in the last example, replacing the code* `!(a==null || a.size()<=0 )` *with the code* `!(a==null) || !(a.size()<=0)` *would be incorrect. You cannot just distribute a negation among other terms. Always remember to use De Morgan's law:* $\neg(p \vee q) \neq \neg p \vee \neg q$.

⋆**Exercise 4.60.** Simplify the following code as much as possible.
Hint: Example 4.56 might be of use.

```
if ((x>0 && x<y) || (x>0 && x>=y)) {
    x=y;
}
```

⋆**Evaluate 4.61.** Simplify the following code as much as possible.

```
if (x>0) {
    if(x<y || x>0) {
        x=y;
    }
}
```

Solution: Because the second if is in the first one which is if $(x > 0)$ then $x > 0$ is duplicated but at the same time to satisfy the second one we just need to keep the second if and cut the first one. $x < y$ and $x > 0$ are independent conditions so they cannot be more simplified. So the answer is:

```
if(x<y || x>0) {
    x=y;
}
```

Evaluation ______________________________

⋆**Exercise 4.62.** Simplify the following code as much as possible.

```
if (x>0) {
    if(x<y || x>0) {
        x=y;
    }
}
```

Although some of these examples may seem a bit contrived, in some sense they *are* realistic. As code is refactored, code is added and removed in various places, conditionals are combined or separated, etc. and sometimes it leads to conditionals that are more complicated than they need to be. In addition, when working on large teams, you will often work on code written by others. Since some programmers don't have a good grasp on logic, you will certainly run into conditional statements that are way more complicated and convoluted than necessary. As I believe these examples demonstrate, simplifying conditionals is not nearly as easy as one might think. It takes great care to ensure that your simplified version is still correct.

Note: *There is an important difference between the logical operators as discussed here and how they are implemented in programming languages such as Java, C, and C++. It is something that is sometimes called* **short circuiting**. *You are probably familiar with the concept even if you haven't heard it called that before. It exploits the* **domination laws:**

$$F \wedge q = F$$

$$T \vee q = T$$

Let's see an example.

Example 4.63. Consider the statement `if(x>=0 && a[x]!=0)`. The first domination law implies that when $x < 0$, the expression in the if statement will evaluate to false regardless of the truth value of `a[x]!=0`. Therefore, many languages will simply not evaluate the second part of the expression—they will *short circuit*.

The same thing happens for statements like `if(x<0 || x>a.length)`. When $x < 0$, the expression is true regardless of the truth value of `x>a.length`. Again, many languages don't evaluate the second part of this expression if the first part is true. Of course, if the first part is false, the second part is evaluated since the truth value now depends on the truth value of the second part.

There are at least two benefits of this. First, it is more efficient since sometimes less code needs to be executed. Second, it allows the checking of one condition before checking a second condition that might cause a crash. You have probably used it in statement like the above to make sure you don't index outside the bounds of an array. Another use is to avoid attempting to access methods or fields when a variable refers to null (e.g. `if(a!=null && a.size()>0)`).

But this has at least two consequences that can cause subtle problems if you aren't careful. First, although the AND and OR operators are *commutative* (e.g. $p \vee q$ and $q \vee p$ are equivalent), that is not always the case for Boolean expressions in these languages. For instance, the statement `if(x>=0 && a[x]!=0)` is not equivalent to `if(a[x]!=0 && x>=0)` since the second one will cause a crash if $x < 0$. Second, if the second part of the expression is code that you expect will always be executed, you may spend a long time tracking down the bug that this creates.

⋆**Evaluate 4.64.** Rewrite the following segment of code so that it is as simple as possible and logically equivalent.

```
if( !(list.isEmpty() && list.get(0)>=100) && !(list.get(0)<100) )
{
    x++;
} else
{
    x--;
}
```

Solution 1: The second and third statements mean the same thing. Also, if the second is true then we got a value so we know the list is not empty, so the first statement is unnecessary. This leads to the following equivalent code:

```
if(list.get(0) >= 100) {x++;} else {x--;}
```

Evaluation ______________________________

____

Solution 2: I used DeMorgan's law to obtain:

```
if(!list.isEmpty() || list.get(0) < 100) {
    x++;
} else {
    x--;
}
```

Evaluation ______________________________

____

Solution 3: Let a be `list.isEmpty()` and b be `list.get(0)>=100`. But then $\neg b$ =`list.get(0)<100`. The original expression is $\neg(a \wedge b) \wedge \neg(\neg b)$. But

$$\begin{aligned}
\neg(a \wedge b) \wedge \neg(\neg b) &= \neg(a \wedge b) \wedge b \\
&= (\neg a \vee \neg b) \wedge b \\
&= (\neg a \wedge b) \vee (\neg b \wedge b) \\
&= (\neg a \wedge b) \vee F \\
&= \neg a \wedge b
\end{aligned}$$

So my simplified code is

```
if( !list.isEmpty() && list.get(0)>= 100 ) {
    x++;
} else {
    x--;
}
```

Evaluation ______________________________

____

⋆**Question 4.65.** In the solutions to the previous problem we said that the final solution was correct. But there might be a catch. Go back to the original code and the final solution and look closer. Is the final solution *really* equivalent to the original? Explain why or why not.

Evaluation ____________________

____

____

The previous question serves as a reinforcement of a point previously made. When dealing with logical expressions in programs, we have to be careful about our notion of equivalence. This is because of short-circuiting and the fact that expressions in programs, unlike logical statements, can crash instead of being true or false.

⋆**Exercise 4.66.** Let p be "$x > 0$", q be "$y > 0$," and r be "Exactly one of x or y is greater than 0."

(a) Express r in terms of p and q using $\oplus$ (and possibly $\neg$).

Answer ____________________

(b) Express r in terms of p and q *without using* $\oplus$.

Answer ____________________

Table 4.4 contains some important identities involving $\rightarrow$, $\leftrightarrow$, and $\oplus$. Since these operators are not always present in a programming language, identities that express them in terms of $\vee$, $\wedge$, and $\neg$ are particularly important.

$p \oplus q = (p \vee q) \wedge \neg(p \wedge q)$	$p \leftrightarrow q = (p \rightarrow q) \wedge (q \rightarrow p)$
$p \oplus q = (p \wedge \neg q) \vee (\neg p \wedge q)$	$p \leftrightarrow q = \neg p \leftrightarrow \neg q$
$\neg(p \oplus q) = p \leftrightarrow q$	$p \leftrightarrow q = (p \wedge q) \vee (\neg p \wedge \neg q)$
$p \rightarrow q = \neg q \rightarrow \neg p$	$\neg(p \leftrightarrow q) = p \leftrightarrow \neg q$
$p \rightarrow q = \neg p \vee q$	$\neg(p \leftrightarrow q) = p \oplus q$

Table 4.4: Logical equivalences involving $\rightarrow$, $\leftrightarrow$, and $\oplus$

Here is the proof of one of these.

Example 4.67. Prove that $p \oplus q = (p \wedge \neg q) \vee (\neg p \wedge q)$.

Solution: It is straightforward to see that $(p \wedge \neg q) \vee (\neg p \wedge q)$ is true if p is true and q is false, or if p is false and q is true, and false otherwise. Put another way, it is true iff p and q have different truth values. But this is the definition of $p \oplus q$. Thus, $p \oplus q = (p \wedge \neg q) \vee (\neg p \wedge q)$.

The previous example demonstrates an important general principle. When proving identities (or equations of any sort), sometimes it works best to start from the right hand side. Try to keep this in mind in the future.

⋆**Evaluate 4.68.** Show that $p \leftrightarrow q$ and $(p \wedge q) \vee (\neg p \wedge \neg q)$ are logically equivalent.

Proof 1: $p \leftrightarrow q$ is true when p and q are both true, and so is $(p \wedge q) \vee (\neg p \wedge \neg q)$. Therefore they are logically equivalent.

Evaluation ______________________________

____

____

Proof 2: They are both true when p and q are both true or both false. Therefore they are logically equivalent.

Evaluation ______________________________

____

____

Proof 3: Each of these is true precisely when p and q are both true.

Evaluation ______________________________

____

____

Proof 4: Each of these is true when p and q have the same truth value and false otherwise, so they are equivalent.

Evaluation ______________________________

____

____

In the previous example, you should have noticed that just a subtle change in wording can be the difference between a correct or incorrect proof. When writing proofs, remember to be very precise in how you word things. You may know what you mean when you wrote something, but a reader can only see what you actually wrote.

4.3 Predicates and Quantifiers

Definition 4.69. *A* **predicate** *or* **propositional function** *is a statement containing one or more variables, whose truth or falsity depends on the value(s) assigned to the variable(s).*

We have already seen predicates in previous examples. Let's revisit one.

Example 4.70. In a previous example we saw that "$x < 0$" was a contingency, "$x^2 < 0$" was a contradiction, and "$x^2 \geq 0$" was a tautology. Each of these is actually a predicate since until we assign a value to x, they are not propositions.

Sometimes it is useful to write *propositional functions* using functional notation.

Example 4.71. Let $P(x)$ be "$x < 0$". Notice that until we assign some value to x, $P(x)$ is neither true nor false.

$P(3)$ is the proposition "$3 < 0$," which is false.

If we let $Q(x)$ be "$x^2 \geq 0$," then $Q(3)$ is "$3^2 \geq 0$," which is true.

Notice that both $P(x)$ and "$x < 0$" are propositional functions. In other words, we don't have to use functional notation to represent a propositional function. Any statement that has a variable in it is a propositional function, whether we label it or not.

⋆**Exercise 4.72.** Which of the following are propositional functions?

(a) ___ $x^2 + 2x + 1 = 0$

(b) ___ $3^2 + 2 \cdot 3 + 1 = 0$

(c) ___ John Cusack was in movie M.

(d) ___ x is an even integer if and only if $x = 2k$ for some integer k.

Definition 4.73. *The symbol* $\forall$ *is the* **universal quantifier***, and it is read as "for all", "for each", "for every", etc. For instance,* $\forall x$ *means "for all* x*". When it precedes a statement, it means that the statement is true* for all values of x.

As the name suggests, the "all" refers to everything in the **universe of discourse** *(or* **domain of discourse***, or simply* **domain***), which is simply the set of objects to which the current discussion relates.*

Example 4.74. Let $P(x)$="$x < 0$". Then $P(x)$ is a propositional function, and $\forall x P(x)$ means "all values of x are negative." If the domain is $\mathbb{Z}$, $\forall x P(x)$ is false. However, if the domain is negative integers, $\forall x P(x)$ is true.

Hopefully you recall that $\mathbb{N}$ is the set of natural numbers ($\{0, 1, 2, \ldots\}$) and $\mathbb{Z}$ is the set of integers. We will use these in some of the following examples.

Example 4.75. Express each of the following English sentences using the universal quantifier. Don't worry about whether or not the statements are true. Assume the domain is real numbers.

(a) The square of every number is non-negative.

(b) All numbers are positive.

Solution:

(a) $\forall x(x^2 \geq 0)$

(b) $\forall x(x > 0)$

⋆**Exercise 4.76.** Express each of the following using the universal quantifier. Assume the domain is $\mathbb{Z}$.

(a) Two times any number is less than three times that number.

Answer ____________________

(b) $n!$ is always less than n^n.

Answer ____________________

Example 4.77. The expression $\forall x(x^2 \geq 0)$ means "for all values of x, x^2 is non-negative". But what constitutes *all* values? In other words, what is the domain? In this case the most logical possibilities are the integers or real numbers since it seems to be stating something about numbers (rather than people, for example). In most situations the context should make it clear what the domain is.

Example 4.78. The expression $\forall x \geq 0, x^3 \geq 0$ means "for all positive values of x, $x^3 \geq 0$." There are several other ways of expressing this idea, but this one is probably the most convenient. One alternative would be to restrict the domain to positive numbers and write it as $\forall x(x^3 \geq 0)$. But sometimes you don't want to or can't restrict the domain.

Another way to express it is $\forall x(x \geq 0 \rightarrow x^3 \geq 0)$.

⋆**Exercise 4.79.** Use the universal quantifier to express the fact that the square of any integer is not zero as long as the integer is not zero.

Answer ____________________

Definition 4.80. *The symbol* $\exists$ *is the* **existential quantifier**, *and it is read as "there exists", "there is", "for some", etc. For instance,* $\exists x$ *means "For some* x*". When it precedes a statement, it means that the statement is true for* at least one value of x *in the universe.*

Example 4.81. Prove that $\exists x(\sqrt{x} = 2)$ is true when the domain is the integers.

Proof. Notice that when $x = 4$, $\sqrt{x} = \sqrt{4} = 2$, proving the statement. □

⋆**Exercise 4.82.** Express the sentence "Some integers are positive" using quantifiers. You may assume the domain of the variable(s) is $\mathbb{Z}$.

Answer ______________________________

Sometimes you will see *nested quantifiers.* Let's see a few examples.

Example 4.83. Use quantifiers to express the sentence "all positive numbers have a square root," where the domain is real numbers.

Solution: We can express this as $\forall(x > 0)\exists y(\sqrt{x} = y)$.

⋆**Evaluate 4.84.** Express the sentence "Some integers are even" using quantifiers. You may assume the domain of the variable(s) is the integers.

Solution 1: $\exists x(x \text{ is even})$.

Evaluation ______________________________

____

Solution 2: $\exists x(x/2 \in \mathbb{Z})$.

Evaluation ______________________________

____

Solution 3: $\exists x \exists y(x = 2y)$.

Evaluation ______________________________

____

Example 4.85. Translate $\forall\forall\exists\exists$ into English.

Solution: It means "*for every upside-down A there exists a backwards E.*" This is a geeky math joke that might make sense if you paid attention in calculus (assuming you ever took calculus, of course). If you don't get it, don't worry about it. Move along. These aren't the droids you're looking for.

⋆**Exercise 4.86.** Express the following statement using quantifiers: "Every integer can be expressed as the sum of two squares." Assume the domain for all three variables (did you catch the hint?) is $\mathbb{Z}$.

Answer ____

⋆**Fill in the details 4.87.** Prove or disprove the statement from the previous example.

Proof: The statement is false. Let $x = 3$. We need to show that no choice of ____________ will yield $y^2 + z^2 = 3$. We can restrict y and z to ________________ since the square of a negative integer is the same as the square of its absolute value. We will do a proof by cases, considering the possible values of y.

$y \neq 0$ since 3 is not ________________.

If $y = 1$, we need ____________, which is impossible.

If $y \geq 2$, $y^2 \geq 4$, so we need ____________, ________________.

Since we have ________________________ and none of them work, the statement is false. □

Example 4.88. Prove or disprove that the following statement is true

$$\forall n \in \mathbb{N}\ \exists m \in \mathbb{N}\ \big(n > 3 \to (n+7)^2 > 49 + m\big)$$

Solution: First, you need to convince yourself that if we can always find some value of m based on a given value of $n > 3$ such that $(n+7)^2 > 49 + m$, the statement is true. Notice that $(n+7)^2 > 49 + m$ iff $n^2 + 14n > m$. So if we take m to be any number smaller than $n^2 + 14n$, for instance $m = n^2 + 14n - 1$, then the statement is true.

Example 4.89. Prove or disprove that the following statement is equivalent to the statement from the Example 4.88.

$$\exists m \in \mathbb{N}\ \forall n \in \mathbb{N}\ \big(n > 3 \to (n+7)^2 > 49 + m\big)$$

Solution: This is almost the same as the expression from the previous example, but the $\forall n \in \mathbb{N}$ and $\exists m \in \mathbb{N}$ have been reversed. Does that change the meaning?

Let's find out.

The expression in the previous example is saying something like "For any natural number n, there is some natural number $m \ldots$" In English, the statement in this example is saying something like "There exists a natural number m such that for any natural number $n \ldots$" Are these different? Indeed. The one from the previous example lets us pick a value of m based on the value of n. The one from this example requires that we pick a value of m that will work for all values of n. Can you see how that is saying something different?

Example 4.90. Prove or disprove that the following statement is true.

$$\exists m \in \mathbb{N}\ \forall n \in \mathbb{N}\ \Big(n > 3 \to (n+7)^2 > 49 + m\Big)$$

Solution: This statement is true. We need there to be some value of m such that for any $n > 3$, $n^2 + 14n > m$ (we worked this out in Example 4.88). Can we find an m such that $m < n^2 + 14n$ for all values of $n > 3$? Sure. It should be clear that $m = 3^2 + 14 \cdot 3 < n^2 + 14n$ for all values of $n > 3$.

⋆**Exercise 4.91.** Find a predicate $P(x, y)$ such that $\forall x \exists y P(x, y)$ and $\exists y \forall x P(x, y)$ have different true values. Justify your answer. (Hint: Think simple. Will something like "$x = y$" or "$x < y$" work if we choose the appropriate domain?)
Answer:

Example 4.92. Let $P(x)$="$x < 0$". Then $\neg\forall x P(x)$ means "it is not the case that all values of x are negative." Put more simply, it means "some value of x is not negative", which we can write as $\exists x \neg P(x)$.

What we saw in the last example actually holds for any propositional function.

Theorem 4.93 (DeMorgan's Laws for quantifiers)**.** *If $P(x)$ is a propositional function, then*

$$\neg\forall x P(x) = \exists x \neg P(x), \text{ and}$$

$$\neg\exists x P(x) = \forall x \neg P(x).$$

Proof: *We will prove the first statement. The proof of the other is very similar. Notice that $\neg\forall x P(x)$ is true if and only if $\forall x P(x)$ is false. $\forall x P(x)$ is false if and only if there is at least one x for which $P(x)$ is false. This is true if and only if $\neg P(x)$ is true for some x. But this is exactly the same thing as $\exists x \neg P(x)$, proving the result.* □

Example 4.94. Negate the following expression, but simplify it so it does not contain the $\neg$ symbol.

$$\forall n \exists m (2m = n)$$

Solution:

$$\begin{aligned} \neg\forall n \exists m(2m = n) &= \exists n \neg\exists m(2m = n) \\ &= \exists n \forall m \neg(2m = n) \\ &= \exists n \forall m (2m \neq n) \end{aligned}$$

⋆**Exercise 4.95.** Answer the following questions about the expression from Example 4.94, assuming the domain is $\mathbb{Z}$.

(a) Write the expression in English. You can start with a direct translation, but then smooth it out as much as possible.

Answer ______________________________

____

(b) Write the negation of the expression in English. State it as simply as possible.

Answer ______________________________

____

(c) What is the truth value of the expression? Prove it.

Answer ______________________________

____

Let's see how quantifiers connect to algorithms. If you want to determine whether or not something (e.g. $P(x)$) is true for all values in a domain (e.g., you want to determine the truth value of $\forall x P(x)$), one method is to simply loop through all of the values and test whether or not $P(x)$ is true. If it is false for any value, you know the answer is false. If you test them all and none of them were false, you know it is true.

Example 4.96. Here is how you might determine if $\forall x P(x)$ is true or false for the domain $\{0, 1, 2, \ldots, 99\}$:

```
boolean isTrueForAll() {
    for(int i=0;i<100;i++) {
        if( !P(i) ) {
            return false;
        }
    }
    return true;
}
```

Notice the negation in the code—this can trip you up if you aren't careful.

Example 4.97. Let $P(x)$ and $Q(x)$ be predicates and the domain be $\{0, 1, 2, \ldots, 99\}$. What is `isTrueForAll2()` determining?

```
boolean isTrueForAll2() {
    for(int i=0;i<100;i++) {
        if( !P(i) && !Q(i) )
            return false;
    }
    return true;
}
```

Solution: Notice that if both $P(i)$ and $Q(i)$ are false for the same value of i, it returns false, and otherwise it returns true. Put another way, it returns true if for every value of i, either $P(i)$ or $Q(i)$ is true. Thus, `isTrueForAll2` is determining the truth value of $\forall i(P(i) \vee Q(i))$.

$\star$**Exercise 4.98.** Rewrite the expression `( !P(i) && !Q(i) )` from the previous example so that it uses only one negation.

Answer:

⋆**Exercise 4.99.** Let $P(x)$ and $Q(x)$ be predicates and the domain be $\{0, 1, 2, \ldots, 99\}$. What is `isTrueForAll3()` determining?

```
boolean isTrueForAll3() {
    boolean result = true;
    for(int i=0;i<100;i++) {
        if(!P(i)) {
            result = false;
        }
    }
    if(result==true) {
        return true;
    }
    for(int i=0;i<100;i++) {
        if(!Q(i)) {
            return false;
        }
    }
    return true;
}
```

Answer ____

____

____

Example 4.100. Now we are ready for the million dollar question:[a] Are `isTrueForAll2` and `isTrueForAll3` determining the same thing?

Solution: At first glance, it looks like they might be. But we need to dig deeper, and we need to prove one way or the other. To prove it, we would need to show that these expressions evaluate to the same truth value, regardless of what P and Q are. To disprove it, we just need to find a P and a Q for which these expressions have different truth values. But let's first talk it through to see if we can figure out which answer seems to be correct.

$\forall i(P(i) \vee Q(i))$ is saying that for every value of i, either $P(i)$ or $Q(i)$ has to be true. $\forall iP(i) \vee \forall iQ(i)$ is saying that either $P(i)$ has to be true for every i, or that $Q(i)$ has to be true for every i. These sound similar, but not exactly the same, so we cannot be sure yet. In particular, we cannot jump to the conclusion that they are not equivalent because we described each with different words. There are many ways of wording the same concept.

At this point we either need to try to tweak the wording so that we can see that they are really saying the same thing, or we need to try to convince ourselves they aren't. Let's try the latter.

What if $P(i)$ is always true and $Q(i)$ is always false? Then both statements are true. But that doesn't prove that they are always both true, so this doesn't help.

Let's try something else. What if we can find a $P(i)$ and a $Q(i)$ such that for any given value of i, we can ensure that either $P(i)$ or $Q(i)$ is true, but also that there is some value j such that $P(j)$ is false and some value $k \neq j$ such that $Q(k)$ is false? Then $\forall i(P(i) \vee Q(i))$ would be true, but $\forall iP(i) \vee \forall iQ(i)$ false, so this would work. But in order to be certain, we have to know that such a P and Q exist.[b]

What if we let $P(i)$ be "i is even", $Q(i)$ be "i is odd", and the universe be $\mathbb{Z}$. Then $\forall iP(i) = \forall iQ(i) = F$, so $\forall iP(i) \vee \forall iQ(i) = F$, but $\forall i(P(i) \vee Q(i)) = T$. Now we have all of the pieces. Let's put this all together in the form of a proof.

Proof: (that $\forall i(P(i) \vee Q(i)) \neq \forall iP(i) \vee \forall iQ(i)$)
Let $P(i)$ be "i is even", $Q(i)$ be "i is odd", and the universe be $\mathbb{Z}$. Then $\forall i(P(i) \vee Q(i))$ is true since every integer is either even or odd. On the other hand, $\forall iP(i)$ is false since there are integers that are not even and $\forall iQ(i)$ is false since there are integers that are not odd. Thus, $\forall iP(i) \vee \forall iQ(i)$ is false. Since they have different truth values, $\forall i(P(i) \vee Q(i)) \neq \forall iP(i) \vee \forall iQ(i)$ $\square$

[a]There is no million dollars for answering this question. It's just an expression.

[b]Consider this: If I can find an even number that is prime but is not 2, then there would be at least 2 even primes. That's great. Unfortunately, I can't find such a number.

4.4 Normal Forms

Earlier we saw identities that express logical operators in terms of $\vee$, $\wedge$, and $\neg$. It turns out that even if there isn't an identity that does it, there is a straightforward technique to convert any logical expression into one only using $\vee$, $\wedge$, and $\neg$. That is the topic of this section.

Definition 4.101. *A* **literal** *is a boolean variable or its negation.*

Definition 4.102. *A* **conjunctive clause** *is a conjunction of one or more literals.*

Example 4.103. Let p, q, and r be boolean variables. Then p, $\neg p$, q, $\neg q$, r, and $\neg r$ are all literals. $p \wedge q \wedge r$, $\neg p \wedge r$, and $r \wedge \neg q \wedge p$ are all conjunctive clauses.

Definition 4.104. *A logical expression is in* **disjunctive normal form** *(***DNF***) (or* **sum-of-products expansion***) if it is expressed as a disjunction of conjunctive clauses.*

Example 4.105. Let p, q, and r be boolean variables. Then the following are in disjunctive normal form:

- $(p \wedge q \wedge r) \vee (\neg p \wedge r)$
- $p \vee (q \wedge \neg p) \vee (r \wedge \neg p)$
- $r \wedge \neg q \wedge p$

These are *not* in disjunctive normal form.

- $p \to q$
- $p \wedge (q \vee r)$
- $p \vee (q \wedge \neg p) \wedge (r \vee \neg q)$

Given a truth table for an expression we can create its *disjunctive normal form* as follows.

Procedure 4.106. *This will convert a boolean expression to disjunctive normal form.*

1. *Create the truth table for the expression.*
2. *Identify the rows having output T.*
3. *For each such row, create a conjunctive clause that includes all of the variables which are true on that row and the negation of all of the variables that are false.*
4. *Combine all of the conjunctive clauses by disjunctions.*

Example 4.107. Express $p \oplus q$ in disjunctive normal form.

Solution: The truth table for $p \oplus q$ is given to the right.

p	q	$p \oplus q$
T	T	F
T	F	T
F	T	T
F	F	F

The second row yields conjunctive clause $p \wedge \neg q$, and the third row yields conjunctive clause $\neg p \wedge q$. The disjunction of these is $(p \wedge \neg q) \vee (\neg p \wedge q)$. Thus, $p \oplus q = (p \wedge \neg q) \vee (\neg p \wedge q)$.

The previous example is essentially just another proof of the identity that was proven in Example 4.67.

⋆**Exercise 4.108.** Express $p \leftrightarrow q$ in disjunctive normal form.

p	q	$p \leftrightarrow q$
T	T	
T	F	
F	T	
F	F	

Example 4.109. Express Z in disjunctive normal form.

p	q	r	Z
T	T	T	T
T	T	F	T
T	F	T	F
T	F	F	F
F	T	T	F
F	T	F	T
F	F	T	F
F	F	F	T

Solution: $Z = (p \wedge q \wedge r) \vee (p \wedge q \wedge \neg r) \vee (\neg p \wedge q \wedge \neg r) \vee (\neg p \wedge \neg q \wedge \neg r)$.

The solution from the previous example can be simplified to $Z = (p \wedge q) \vee (\neg p \wedge \neg r)$. Although this can be done by applying the logical equivalences we learned about earlier, there are more sophisticated techniques that can be used to simplify expressions that are in disjunctive normal form. This is beyond our scope, but you will likely learn more about this when you take a computer organization class and discuss circuit minimization. The important point I want to make here is that computing the disjunctive normal form of an expression using the technique we describe will not always produce the most simple form of the expression. In fact, much of the time it won't be.

⋆**Exercise 4.110.** Express Y in disjunctive normal form.

p	q	r	Y
T	T	T	F
T	T	F	F
T	F	T	F
T	F	F	F
F	T	T	T
F	T	F	T
F	F	T	T
F	F	F	T

There is another important form that is very similar to disjunctive normal form.

Definition 4.111. *A* **disjunctive clause** *is a disjunction of one or more literals. A logical expression is in* **conjunctive normal form** *(***CNF***) (or* **product-of-sums expansion***) if it is expressed as a conjunction of disjunctive clauses.*

There are several methods for converting to conjunctive normal form. They generally involve using double negation, distributive, and De Morgan's laws either based on the truth table or based on the disjunctive normal form. However, we won't discuss these techniques here. The main reason to introduce you to these forms is that they each have connections to important areas of computer science. They are used in circuit design and minimization, artificial intelligence algorithms, automated theorem proving, and the study of algorithm complexity.

4.5 Bitwise Operations

In this section we will consider *bitwise operations*. But first we need to review a few concepts you are probably already familiar with.

In your programming class you learned that a *Boolean variable* is one that is either true or false. You may or may not have learned about the connection between Boolean variables and bits. Recall that a *bit* can have the value 0 or 1. A bit can be used to represent a Boolean variable by assigning 0 to false and 1 to true. Table 4.5 shows the truth tables for the various Boolean operators that are available in many languages. Notice that they are identical to the operators we discussed earlier except that we have replaced T/F with 0/1 and have used the notation from Java/C/C++ instead of the mathematical notation.

		AND	OR	XOR	IFF
p	q	$(p\&\&q)$	$(p\|\|q)$	$p! = q$	$(p == q)$
1	1	1	1	0	1
1	0	0	1	1	0
0	1	0	1	1	0
0	0	0	0	0	1

Table 4.5: Truth tables for the Boolean operators

We don't usually think about `!=` being XOR and `==` being IFF (or biconditional). We usually think of them in their more natural interpretation: 'not equal' and 'equal'.

Note: *A note of caution: Although Java is a lot like C and C++, how it deals with logical expressions is very different. Java has an explicit* boolean *type and you can only use the logical operators on boolean values. Further, conditional statements in Java require boolean values. In C and C++, the* `int` *type is used as a boolean value, where* 0 *is false, and anything else is true. This is very convenient, but can also cause some confusion.*

Example 4.112. In C/C++, `(5&&6)`, `(5||0)`, `(4!=5)` are all true. In Java the first two statements are illegal.

Now it's time to extend the concept of Boolean operators to integer data types (including `int`, `short`, `long`, `byte`, etc.).

Definition 4.113. *A* **bitwise operation** *is a boolean operation that operates on the individual bits of its argument(s).*

Definition 4.114. *The* **compliment** *or* **bitwise NOT***, usually denoted by* `~`*, just flips each bit.*

Example 4.115. Assume `10011001` is in binary. Then `~10011001=01100110`. If this were a 32-bit integer, the answer would be `11111111111111111111111101100110` since the leading 24 bits (which we assume to be 0) would be flipped.

Note: *For simplicity, the rest of the examples will assume numbers are represented with 8 bits. The concept is exactly the same regardless of how many bits are used for a particular data type.*

⋆**Fill in the details 4.116.** 255 is `11111111` in binary. `~11111111=00000000`, which is 0 in decimal. Therefore, `~255=0`.

Similarly, we can see that `~240=15` since 240 is ____________________ in binary, and

`~`____________________ = ____________________, which is ______________ in decimal.

⋆**Exercise 4.117.** `~11000110`=__________________________

Definition 4.118. *The following are the two-operator bitwise operators.*

- *The* **bitwise AND***, usually denoted by* `&`*, applies* $\wedge$ *to the corresponding bits of each argument.*
- *The* **bitwise OR***, usually denoted by* `|`*, applies* $\vee$ *to the corresponding bits of each argument.*
- *The* **bitwise XOR***, usually denoted by* `^`*, applies* $\oplus$ *to the corresponding bits of each argument.*

We will present examples in table form rather than 'code form' since it is much easier to see what is going on when the bits are lined up.

Example 4.119.

```
  01011101     01011101     01011101
& 11010100    |11010100   ^ 11010100
----------    ---------   ----------
  01010100     11011101     10001001
```

Note: *It is important to remember that* `&` *and* `&&` *are not the same thing! The same holds for* `|` *and* `||`*. It is equally important to remember that* `^` *does* not *mean exponentiation in most programming languages.*

⋆**Exercise 4.120.**

$$\begin{array}{r} 11110000 \\ \texttt{\&}\ 11001100 \\ \hline \end{array} \qquad \begin{array}{r} 11110000 \\ \texttt{|}11001100 \\ \hline \end{array} \qquad \begin{array}{r} 11110000 \\ \texttt{\^{}}\ 11001100 \\ \hline \end{array}$$

Note: *A final reminder: It is important to understand the difference between the Boolean operators and the bitwise operators.*

4.6 Problems

Problem 4.1. Draw a truth table to represent the following.

(a) $\neg p \vee q$

(b) $(p \rightarrow q) \vee \neg p$

(c) $(p \wedge \neg q) \vee r$

(d) $((p \vee q) \wedge \neg(p \vee q)) \vee r$

(e) $(p \vee \neg r) \wedge q$

(f) $(p \oplus q) \wedge (q \vee r)$

Problem 4.2. Give 2 different proofs that $[(p \vee q) \wedge \neg p] \rightarrow q$ is a tautology.

Problem 4.3. Prove $\neg(p \leftrightarrow q) = p \oplus q$ without using truth tables.

Problem 4.4. Use Procedure 4.106 to find the disjunctive normal form for each of the expressions from Problem 4.1.

Problem 4.5. Express $p \vee q \vee r$ using only $\wedge$ and $\neg$.

Problem 4.6. The *NAND* of p and q, denoted by $p|q$, is the proposition "not both p and q". The NAND of p and q is false when p and q are both true and true otherwise.

(a) Draw a truth table for $NAND$

(b) Express $p|q$ using $\vee$, $\wedge$, and/or $\neg$ (you may not need all of them).

(c) Express $p \wedge q$ using only $|$. Your answer should be as simple as possible. Give a truth table that shows they are the same.

(d) Express $\neg p \vee q$ using only $|$. Your answer should be as simple as possible. Give a truth table that shows they are the same.

Problem 4.7. The *NOR* of p and q, denoted by $p \downarrow q$, is the proposition "neither p nor q". The NOR of p and q is true when p and q are both false and false otherwise.

(a) Draw a truth table for $\downarrow$

(b) Express $p \downarrow q$ using $\vee$, $\wedge$, and/or $\neg$ (you may not need all of them).

(c) Express $p \wedge q$ using only $\downarrow$. Your answer should be as simple as possible. Give a truth table that shows they are the same.

(d) Express $\neg p \vee q$ using only $\downarrow$. Your answer should be as simple as possible. Give a truth table that shows they are the same.

Problem 4.8. A set of logical operators is *functionally complete* if any possible operator can be implemented using only operators from that set. It turns out that $\{\neg, \wedge\}$ is functionally complete. So is $\{\neg, \vee\}$. To show that a set if functionally complete, all one needs to do is show how to implement all of the operators from another functionally complete set. Given this,

(a) Show that $\{|\}$ is functionally complete. (Hint: Since $\{\neg, \wedge\}$ is functionally complete, one way is to show how to implement both $\wedge$ and $\neg$ using just $|$.)

(b) Show that $\{\downarrow\}$ is functionally complete.

Problem 4.9. Write each of the following expressions so that negations are only applied to propositional functions (and not quantifiers or connectives).

(a) $\neg\forall x\exists y\neg P(x, y)$

(b) $\neg(\forall x\exists y P(x, y) \wedge \exists x\neg\forall y P(x, y))$

(c) $\neg\forall x(\exists y P(x, y) \vee \forall y Q(x, y))$

(d) $\neg\forall x\neg\exists y(\neg\forall z P(x, z) \rightarrow \exists z Q(x, y, z))$

(e) $\neg\exists x(\neg\forall y[\exists z(P(y, x, z) \wedge P(y, z, x) \wedge P(x, y, z))] \vee \exists z Q(x, z))$

Problem 4.10. Let $P(x, y)$="x likes y", where the universe of discourse for x and y is the set of all people. Translate each of the following into English, smoothing them out as much as possible. Then give the truth value of each.

(a) $\forall x\forall y P(x, y)$

(b) $\forall x\exists y P(x, y)$

(c) $\forall y\exists x P(x, y)$

(d) $\forall x P(x, Raymond)$

(e) $\neg\forall x\forall y P(x, y)$

(f) $\forall x\neg\forall y P(x, y)$

(g) $\forall x\neg\forall y\neg P(x, y)$

Problem 4.11. Let $P(x, y, z)$="$x^2 + y^2 = z^2$", where the universe of discourse for all variables is the set of integers. What are the truth values of each of the following?

(a) $\forall x\forall y\forall z P(x, y, z)$

(b) $\exists x\exists y\forall z P(x, y, z)$

(c) $\forall x\exists y\exists z P(x, y, z)$

(d) $\forall x\forall y\exists z P(x, y, z)$

(e) $\forall x\exists y\forall z P(x, y, z)$

(f) $\exists x\exists y\exists z P(x, y, z)$

(g) $\exists z P(2, 3, z)$

(h) $\exists x\exists y P(x, y, 5)$

(i) $\exists x\exists y P(x, y, 3)$

Problem 4.12. Write each of the following sentences using quantifiers and propositional functions. Define propositional functions as necessary (e.g. Let $D(x)$ be the proposition 'x plays disc golf.')

(a) All disc golfers play ultimate Frisbee.

(b) If all students in my class do their homework, then some of the students will pass.

(c) If none of the students in my class study, then all of the students in my class will fail.

(d) Not everybody knows how to throw a Frisbee 300 feet.

(e) Some people like ice cream, and some people like cake, but everybody needs to drink water.

(f) Everybody loves somebody.

(g) Everybody is loved by somebody.

(h) Not everybody is loved by everybody.

(i) Nobody is loved by everybody.

(j) You can't please all of the people all of the time, but you can please some of the people some of the time.

(k) If only somebody would give me some money, I would buy a new house.

(l) Nobody loves me, everybody hates me, I'm going to eat some worms.

(m) Every rose has its thorn, and every night has its dawn.

(n) No one ever is to blame.

Problem 4.13. Express the following phrase using quantifiers. "There is some constant c such that $f(x)$ is no greater than $c \cdot g(x)$ for all $x \geq x_0$ for some constant x_0." Your solution should contain no English words.

Problem 4.14. Consider the following expression:

$$\forall \epsilon{>}0 \, \exists \delta{>}0 \, \forall x (0 < |x - c| < \delta \rightarrow |f(x) - L| < \epsilon).$$

(a) Express it in English. Be as concise as possible.

(b) (Difficult if you have not had calculus.) This is the definition of something. What is it?

Problem 4.15. You are helping a friend debug the code below. He tells you "The code in the if statement never executes. I have tried it for `x=2`, `x=4`, and even `x=-1`, and it never gets to the code inside the if statement."

```
if((x%2==0 && x<0) || !(x%2==0 || x<0)) {
    // Do something.
}
```

(a) Is he correct that the code inside the if statement does not execute for his chosen values? Justify your answer.

(b) Under what conditions, if any, will the code in the if statement execute? Be specific and complete.

Problem 4.16. Simplify the following code as much as possible:

```
if(x<=0 && x>0) {
    doSomething();
} else {
    doAnotherThing();
}
```

Problem 4.17. Consider the following code.

```
boolean notBothZero(int x, int y) {
    if(!(x==0 && y==0)) {
        return true;
    } else {
        return false;
    }
}
boolean unknown1(int x, int y) {
    if(x!=0 && y!=0) {
        return true;
    } else {
        return false;
    }
}
boolean unknown2(int x, int y) {
    if(x!=0 || y!=0) {
        return true;
    } else {
        return false;
    }
}
```

(a) Is `unknown1` equivalent to `notBothZero`? Prove or disprove it.

(b) Is `unknown2` equivalent to `notBothZero`? Prove or disprove it.

(c) Are `unknown1` and `unknown2` equivalent to each other? Prove or disprove it.

Problem 4.18. Simplify the following code as much as possible. (It can be simplified into a single if statement that is about as complex as the original outer if statement).

```
if ( (!x.size()<=0  && x.get(0)!=11) || x.size()>0 ) {
   if ( !(x.get(0)==11 && (x.size()>13 || x.size()<13) )
      && (x.size()>0 || x.size()==13) ) {
            // Do a few things.
   }
}
```

Problem 4.19. The following method returns true if and only if none of the entries of the array are 0:

```
boolean noZeroElements(int[] a, int n) {
    for(int i=0;i<n;i++) {
        if(a[i] == 0 )
            return false;
    }
    return true;
}
```

The two methods below implement this idea for two arrays. Assume `list1` and `list2` have the same size for both of these methods.

```
boolean unknown1(int[] list1, int[] list2, int n) {
    for(int i=0;i<n;i++) {
        if( list1[i]==0 && list2[i]==0 )
            return false;
    }
    return true;
}

boolean unknown2(int[] list1, int[] list2, int n) {
    if(noZeroElements(list1, n)) {
        return true;
    } else if(noZeroElements(list2, n) {
        return true;
    } else {
        return false;
    }
}
```

(a) What is `unknown1` determining? (Give answer in terms of `list1` and `list2` and the appropriate quantifier(s).)

(b) What is `unknown2` determining? (Give answer in terms of `list1` and `list2` and the appropriate quantifier(s).)

(c) Prove or disprove that `unknown1` and `unknown2` are determining the same thing.

Chapter 5

Sets, Functions, and Relations

5.1 Sets

Definition 5.1. *A* **set** *is an unordered collection of objects. These objects are called the* **elements** *of the set. If a belongs to the set A, then we write $a \in A$, read "a is an element of A." If a does not belong to the set A, we write $a \notin A$, read "a is not an element of A." Generally speaking, repeated elements in a set are ignored.*

Definition 5.2. *The number of elements in a set A, also known as the the* **cardinality** *of A, will be denoted by $\operatorname{card}(A)$ or $|A|$. If the set A has infinitely many elements, we write $|A| = \infty$.*

Example 5.3. Let $D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ be the set of the ten decimal digits. Then $4 \in D$ but $11 \notin D$. Also, $|D| = 10$.

Notice that the elements in a set are listed between curly braces. You should do the same when you specify the elements of a set.

$\star$**Exercise 5.4.** What is the set of prime numbers less than 10?

Answer ____

Example 5.5. The sets $\{1, 2, 3\}$, $\{3, 2, 1\}$, and $\{1, 1, 1, 2, 2, 3\}$ actually represent the same set since repeated values are ignored and the order elements are listed does not matter. The cardinality of each of these sets is 3.

Definition 5.6. *We say two sets are* **equal** *if they contain the same elements. That is $\forall x(x \in A \leftrightarrow x \in B)$. If A and B are equal sets, we write $A = B$.*

Note: *We will normally denote sets by capital letters, say $A, B, S, \mathbb{N}$, etc. Elements will be denoted by lowercase letters, say a, b, ω, r, etc.*

$\star$**Exercise 5.7.** Let $A = \{1,2,3,4,5,6\}$, $B = \{1,2,3,4,5,4,3,2,1\}$, and $C = \{6,3,4,5,1,3,2\}$.

Then $|A| =$ ______, $|B| =$ ______, and $|C| =$ ______.

Which of A, B, and C represent the same sets? ____________________

Definition 5.8. *The following notation is pretty standard, and we will follow it in this book.*

$\mathbb{N} = \{0, 1, 2, 3, \ldots\}$	*the set of* **natural numbers**.
$\mathbb{Z} = \{\ldots -2, -1, 0, 1, 2, \ldots\}$	*the set of* **integers**.
$\mathbb{Z}^+ = \{1, 2, 3, \ldots\}$	*the set of* **positive integers**.
$\mathbb{Z}^- = \{-1, -2, -3, \ldots\}$	*the set of* **negative integers**.
$\mathbb{Q}$	*the* **rational numbers**.
$\mathbb{R}$	*the* **real numbers**.
$\mathbb{C}$	*the* **complex numbers**.
$\varnothing = \{\}$	*the* **empty set** *or* **null set**.

Note: *There is no universal agreement of the definition of $\mathbb{N}$. Although here it is defined as $\{0, 1, 2, 3, \ldots\}$, it is sometimes defined as $\mathbb{N} = \mathbb{Z}^+$. The only difference is whether or not 0 is included. I prefer the definition given here because then we have a notation for the positive integers ($\mathbb{Z}^+$) as well as the non-negative integers ($\mathbb{N}$).*

Example 5.9. Notice that $|\mathbb{N}| = |\mathbb{Z}| = |\mathbb{R}| = \infty$. But this may be a bit misleading. Do all of these sets have the same number of elements? Believe it or not, it turns out that $\mathbb{N}$ and $\mathbb{Z}$ do, but that $\mathbb{R}$ has many more elements than both of these. If it seems strange to talk about whether or not two infinite sets have the same number of elements, don't worry too much about it. We probably won't bring it up again.

$\star$**Exercise 5.10.** (a) $|\mathbb{C}| =$ ________, (b) $|\mathbb{Z}^+| =$ ________, (c) $|\varnothing| =$ ________

Example 5.11. Let S be the set of the squares of integers. We can express this as $S = \{n^2 | n \in \mathbb{Z}\}$ or $S = \{n^2 : n \in \mathbb{Z}\}$. We call this *set builder notation*. We read the : or | as "such that." Thus, S is the set containing "numbers of the form n^2 such that n is an integer."

Example 5.12. Use set builder notation to express $\mathbb{C}$, the set of complex numbers.

Solution: $\mathbb{C} = \{a + bi : a, b \in \mathbb{R}\}$.

⋆**Exercise 5.13.** Use set builder notation to express the set of even integers.

Answer ____________________

⋆**Exercise 5.14.** Use set builder notation to express $\mathbb{Q}$, the set of all rational numbers.

Answer ____________________

Definition 5.15. *If every element in A is also in B, we say that A is a* **subset** *of B and we write this as $A \subseteq B$. If $A \subseteq B$ and there is some $x \in B$ such that $x \notin A$, then we say A is a* **proper subset** *of B, denoting it by $A \subset B$.*

If there is some $x \in A$ such that $x \notin B$, then A is not a subset of B, which we write as $A \not\subseteq B$.

Note: *Some authors use $\subset$ to mean subset without necessarily implying it is a proper subset. Sometimes you will need to consider the context in order to interpret it correctly.*

Example 5.16. Let $S = \{1, 2, \ldots, 20\}$, that is, the set of integers between 1 and 20, inclusive. Let $E = \{2, 4, 6, \ldots, 20\}$, the set of all even integers between 2 and 20, inclusive. Notice that $E \subseteq S$. Let $P = \{2, 3, 5, 7, 11, 13, 17, 19\}$, the set of primes less than 20. Then $P \subseteq S$.

⋆**Exercise 5.17.** Let $S = \{n^2 | n \in \mathbb{Z}\}$ and $A = \{1, 4, 9, 16\}$. Answer each of the following, including a brief justification.

(a) Is $A \subseteq S$? ____________________

(b) Is $A \subset S$? ____________________

(c) Is $S \subseteq S$? ____________________

(d) Is $S \subset S$? ____________________

(e) Is $S \subset A$? ____________________

⋆**Exercise 5.18.** Let A be the set of integers divisible by 6, B be the set of integers divisible by 2, and C be the set of integers divisible by 3. Answer each of the following, giving a brief justification.

(a) Is $A \subseteq B$?____

(b) Is $A \subseteq C$?____

(c) Is $B \subseteq A$?____

(d) Is $B \subseteq C$?____

(e) Is $C \subseteq A$?____

(f) Is $C \subseteq B$?____

Example 5.19. The set

$$S = \{\text{Roxan}, \text{Jacquelin}, \text{Sean}, \text{Fatimah}, \text{Wakeelah}, \text{Ashley}, \text{Ruben}, \text{Leslie}, \text{Madeline}\}$$

is the set of students in a particular course. This set can be split into two subsets: the set $F = \{\text{Roxan}, \text{Jacquelin}, \text{Fatimah}, \text{Wakeelah}, \text{Ashley}, \text{Madeline}\}$ of females in the class, and the set $M = \{\text{Sean}, \text{Ruben}, \text{Leslie}\}$ of males in the class. Thus we have $F \subseteq S$ and $M \subseteq S$. Notice that it is *not true* that $F \subseteq M$ or that $M \subseteq F$.

Example 5.20. Find all the subsets of $\{a, b, c\}$.

Solution: They are $\varnothing$, $\{a\}$, $\{b\}$, $\{c\}$, $\{a, b\}$, $\{b, c\}$, $\{a, c\}$, and $\{a, b, c\}$.

Notice that there are 8 subsets. Also notice that $8 = 2^3$. As we will see shortly, that is not a coincidence.

Notice that we wrote $\varnothing$ and not $\{\varnothing\}$ in the previous example. It turns out that $\varnothing \neq \{\varnothing\}$. $\varnothing$ is the empty set–that is, the set that has no elements. $\{\varnothing\}$ is the set containing the empty set. Thus, $\{\varnothing\}$ is a set containing the single element $\varnothing$. You can use either $\varnothing$ or $\{\}$ to denote the empty set, but not $\{\varnothing\}$.

⋆**Exercise 5.21.** Find all the subsets of $\{a, b, c, d\}$.

Definition 5.22. *The* power set *of a set is the set of all subsets of a set. The power set of a set A is denoted by $P(A)$.*

Example 5.23. If $A = \{a, b, c\}$, example 5.20 implies that $P(A) = \{\varnothing, \{a\}, \{b\}, \{c\}, \{a, b\}, \{b, c\}, \{a, c\}, \{a, b, c\}\}$. Notice that the solution is a set, the elements of which are also sets.

An *incorrect answer* would be $\{\varnothing, a, b, c, \{a, b\}, \{b, c\}, \{a, c\}, \{a, b, c\}\}$. This is incorrect because a is not the same thing as $\{a\}$ (the set containing a). $\{a\} \in P(A)$, but $a \notin P(A)$. This is a subtle but important distinction.

⋆**Exercise 5.24.** Find $P(\{a, b, c, d\})$.

We will prove the following theorem in the next section after we have developed the appropriate notation to do so.

Theorem 5.25. *Let A be a set with n elements. Then $|P(A)| = 2^n$.*

⋆**Exercise 5.26.** Let A be a set with 4 elements.

(a) $|P(A)| =$ ________________________.

(b) $|P(P(A))| =$ ________________________.

(c) $|P(P(P(A)))| =$ ________________________.

⋆**Exercise 5.27.** If one element is added to a finite set A, how much larger is the power set of A after the element is added (relative to the size of the power set before it is added)? Explain your answer.

Answer ____

____

____

5.2 Set Operations

We can obtain new sets by performing operations on other sets. In this section we discuss the common set operations. *Venn diagrams* are often used as a pictorial representation of the relationships between sets. We provide Venn diagrams to help visualize the set operations. In our Venn diagrams, the region(s) in the darker color represent the elements of the set of interest.

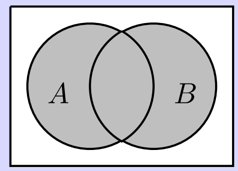
Definition 5.28.

The **union** *of two sets A and B is the set containing elements from either A or B. More formally,*

$$A \cup B = \{x : x \in A \text{ or } x \in B\}.$$

Notice that in this case the **or** *is an* **inclusive or**. *That is, x can be in A, or it can be in B, or it can be in both.*

$A \cup B$



Example 5.29. Let $A = \{1, 2, 3, 4, 5, 6\}$, and $B = \{1, 3, 5, 7\}$. Then $A \cup B = \{1, 2, 3, 4, 5, 6, 7\}$.

⋆**Exercise 5.30.** Let A be the set of even integers and B be the set of odd integers. Then

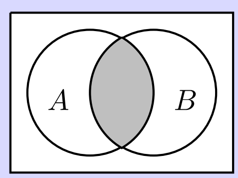
$A \cup B=$ ____

Definition 5.31.

The **intersection** *of two sets A and B is the set containing elements that are in both A and B. More formally,*

$$A \cap B = \{x : x \in A \text{ and } x \in B\}.$$

$A \cap B$



Example 5.32. Let $A = \{1, 2, 3, 4, 5, 6\}$, and $B = \{1, 3, 5, 7, 9\}$. Then $A \cap B = \{1, 3, 5\}$.

⋆**Exercise 5.33.** Let A be the set of even integers and B be the set of odd integers. Then

$A \cap B=$ ____

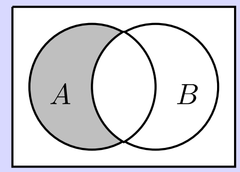
Definition 5.34.

The **difference** *(or* **set-difference***) of sets A and B is the set containing elements from A that are not in B. More formally,*

$$A \setminus B = \{x : x \in A \text{ and } x \notin B\}.$$

The set difference of A and B is sometimes denoted by $A - B$.

$A \setminus B$



Example 5.35. Let $A = \{1, 2, 3, 4, 5, 6\}$, and $B = \{1, 3, 5, 7, 9\}$. Then $A \setminus B = \{2, 4, 6\}$ and $B \setminus A = \{7, 9\}$.

⋆**Exercise 5.36.** Let A be the set of even integers and B be the set of odd integers. Then

$A \setminus B=$ __________________________ and $B \setminus A=$ __________________________.

We can now prove Theorem 5.25.

Example 5.37. Let A be a set with n elements. Then $|P(A)| = 2^n$.

Proof: We use induction[a] and the idea from the solution to Exercise 5.21. Clearly if $|A| = 1$, A has $2^1 = 2$ subsets: $\varnothing$ and A itself.
Assume every set with $n-1$ elements has 2^{n-1} subsets. Let A be a set with n elements. Choose some $x \in A$. Every subset of A either contains x or it doesn't. Those that do not contain x are subsets of $A \setminus \{x\}$. Since $A \setminus \{x\}$ has $n-1$ elements, the induction hypothesis implies that it has 2^{n-1} subsets. Every subset that does contain x corresponds to one of the subsets of $A \setminus \{x\}$ with the element x added. That is, for each subset $S \subseteq A \setminus \{x\}$, $S \cup \{x\}$ is a subset of A containing x. Clearly there are 2^{n-1} such new subsets. Since this accounts for all subsets of A, A has $2^{n-1} + 2^{n-1} = 2^n$ subsets. $\square$

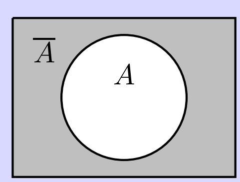
[a] We will cover induction more fully and formally later. But since this use of induction is pretty intuitive, especially in light of Example 5.21, it serves as a useful foreshadowing of things to come.

Definition 5.38.

Let $A \subseteq U$. The **complement** *of A with respect to U is just the set difference $U \setminus A$. More formally,*

$$\overline{A} = \{x \in U : x \notin A\} = U \setminus A.$$

In words, $\overline{A}$ is the set of everything not in A. Other common notations for set complement include A^c and A'.



U

Note: *Often the set U, which is called the* **universe** *or* **universal set***, is implied and we just use $\overline{A}$ to denote the complement. We usually follow this convention here. Further, when talking about several sets, we will usually assume they have the same universal set.*

Example 5.39. Let $U = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ be the universal set of decimal digits and $A = \{0, 2, 4, 6, 8\} \subset U$ be the set of even digits. Then $\overline{A} = \{1, 3, 5, 7, 9\}$ is the set of odd digits.

⋆**Exercise 5.40.** Let A be the set of even integers and B be the set of odd integers, and let the universal set be $U = \mathbb{Z}$. Then $\overline{A}=$ __________________ and $\overline{B}=$ __________________.

It should not be too difficult to convince yourself that the following theorem is true.

Theorem 5.41. *Let A be a subset of some universal set U. Then*

$$\begin{aligned} \overline{A} \cap A &= \emptyset, \text{ and} \\ \overline{A} \cup A &= U. \end{aligned}$$

The various intersecting regions for two and three sets can be seen in Figures 5.1 and 5.2.

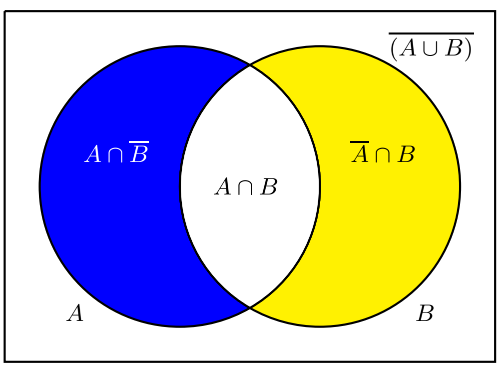


Figure 5.1: Venn diagram for two sets.

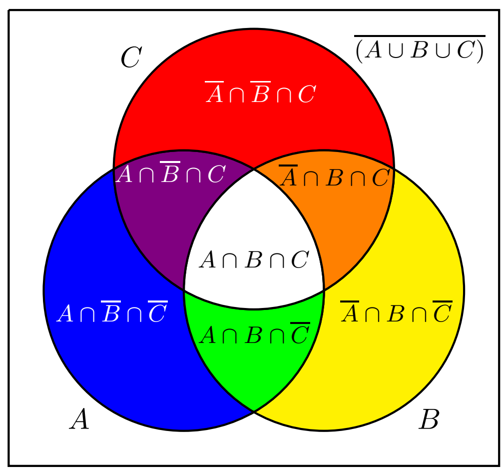


Figure 5.2: Venn diagram for three sets.

Definition 5.42. *Two sets A and B are* **disjoint** *or* **mutually exclusive** *if $A \cap B = \emptyset$. That is, they have no elements in common.*

Example 5.43. Let A be the set of prime numbers, B be the set of perfect squares, and C be the set of even numbers. Then A and B are clearly disjoint since if a number is a perfect square, it cannot possibly be prime (although 0 and 1 are not prime for different reasons than the rest of the elements of B). On the other hand, A and C are not disjoint since they both contain 2, and B and C are not disjoint because they both contain 4.

⋆**Exercise 5.44.** Let A be the set of even integers and B be the set of odd integers. Are A and B disjoint? Explain.

Answer ______________________________

____

Set identities can be used to show that two sets are the same. Table 5.1 gives some of the most common set identities. In these identities, U is the universal set. We won't provide proofs for most of these, but we will present a few examples and a technique that will allow you to verify that they are correct.

Name	*Identity*
commutativity	$A \cup B = B \cup A$ $A \cap B = B \cap A$
associativity	$A \cup (B \cup C) = (A \cup B) \cup C$ $A \cap (B \cap C) = (A \cap B) \cap C$
distributive	$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
identity	$A \cup \varnothing = A$ $A \cap U = A$
complement	$A \cup \overline{A} = U$ $A \cap \overline{A} = \varnothing$
domination	$A \cup U = U$ $A \cap \varnothing = \varnothing$
idempotent	$A \cup A = A$ $A \cap A = A$
complementation	$\overline{(\overline{A})} = A$
DeMorgan's	$\overline{A \cup B} = \overline{A} \cap \overline{B}$ $\overline{A \cap B} = \overline{A} \cup \overline{B}$
absorption	$A \cup (A \cap B) = A$ $A \cap (A \cup B) = A$

Table 5.1: Set Identities

These identities may look somewhat familiar. They are essentially the same as the logical equivalences presented in Table 4.3. In fact, if we equate T to U, F to $\varnothing$, $\vee$ to $\cup$, $\wedge$ to $\cap$, and $\neg$ to $\bar{\ }$, the laws are identical. This is because logic operations and sets are both what we call *Boolean algebras*. We won't go into detail about this connection, but in case you run into the concept in the future, you heard it here first!

The following theorem can be used to prove set identities.

Theorem 5.45. *Two sets A and B are equal if and only if $A \subseteq B$ and $B \subseteq A$.*

Let's see this theorem in action.

Example 5.46. Prove that $A \setminus B = A \cap \overline{B}$.

Proof: Let $x \in A \setminus B$. Then by definition of difference, $x \in A$ and $x \notin B$. But if $x \notin B$, then $x \in \overline{B}$ by definition of complement. Since $x \in A$ and $x \in \overline{B}$, $x \in A \cap \overline{B}$ by definition of intersection. Since whenever $A \setminus B$, $x \in A \cap \overline{B}$, we have shown that $A \setminus B \subseteq A \cap \overline{B}$.

Now assume that $x \in A \cap \overline{B}$. Then $x \in A$ and $x \in \overline{B}$ by definition of intersection. By definition of complement, $x \notin B$. But if $x \in A$ and $x \notin B$, then $x \in A \setminus B$ by definition of difference. Since whenever $x \in A \cap \overline{B}$, $x \in A \setminus B$, we have that $A \cap \overline{B} \subseteq A \setminus B$.

Since we have shown that $A \setminus B \subseteq A \cap \overline{B}$ and that $A \cap \overline{B} \subseteq A \setminus B$, by Theorem 5.45 $A \setminus B = A \cap \overline{B}$. □

That was the long, drawn-out version of the proof. The purpose of all of the detail is to make the technique clear. Here is a proof without any extraneous details.

Proof: We will prove this by showing set containment both ways.

Let $x \in A \setminus B$. Then $x \in A$ and $x \notin B$. This implies that $x \in \overline{B}$. Therefore $x \in A \cap \overline{B}$. Since $A \setminus B$ implies $x \in A \cap \overline{B}$, $A \setminus B \subseteq A \cap \overline{B}$.

Now assume that $x \in A \cap \overline{B}$. Then $x \in A$ and $x \in \overline{B}$. Then $x \notin B$, and therefore $x \in A \setminus B$. Since $x \in A \cap \overline{B}$ implies $x \in A \setminus B$, $A \cap \overline{B} \subseteq A \setminus B$. □

The proofs in the previous example are called *set containment proofs* since we showed set containment both ways. The technique is pretty straightforward: Theorem 5.45 tells us that if $X \subseteq Y$ and $Y \subseteq X$, then $X = Y$. Thus, to prove $X = Y$, we just need to show that $X \subseteq Y$ and $Y \subseteq X$. But how do we show that one set is a subset of another? This is easy: To show that $X \subseteq Y$, we show that every element from X is also in Y. In other words, we assume that $x \in X$ and use definitions and logic to show that $x \in Y$. Assuming we do not use any special properties about x other than the fact that $x \in X$, then x is an arbitrary element from X, so this shows that $X \subseteq Y$. Showing that $Y \subseteq X$ uses exactly the same technique.

Note: *Be careful. To prove that $X = Y$, you generally need to prove two things: $X \subseteq Y$ and $Y \subseteq X$. Do not forget to do both. On the other hand, if you are asked to prove that $X \subseteq Y$, you do not need to (and should not) show that $Y \subseteq X$.*

Let's see another example of this type of proof. This proof will provide a few more details than necessary in order to further explain the technique.

Theorem 5.47. *Prove the first De Morgan's Laws: Given sets A and B, $\overline{(A \cup B)} = \overline{A} \cap \overline{B}$.*

Proof: *Let $x \in \overline{(A \cup B)}$. Then $x \notin A \cup B$ (by definition of complement). Thus $x \notin A$ and $x \notin B$ (by definition of union), which is the same thing as $x \in \overline{A}$ and $x \in \overline{B}$ (by definition of complement). But then we have that $x \in \overline{A} \cap \overline{B}$ (by definition of intersection). Notice that x was an arbitrary element from $\overline{(A \cup B)}$, and we showed that $x \in \overline{A} \cap \overline{B}$. Therefore, every element in $\overline{(A \cup B)}$ is also in $\overline{A} \cap \overline{B}$. In other words, $\overline{(A \cup B)} \subseteq \overline{A} \cap \overline{B}$.*

Now, let $x \in \overline{A} \cap \overline{B}$. Then $x \in \overline{A}$ and $x \in \overline{B}$. This means that $x \notin A$ and $x \notin B$ which is the same as $x \notin A \cup B$. But this last statement asserts that $x \in \overline{(A \cup B)}$. Hence $\overline{A} \cap \overline{B} \subseteq \overline{(A \cup B)}$.

Since we have shown that the two sets contain each other, they are equal by Theorem 5.45. □

You have already seen a few correct ways to prove that $A \setminus B = A \cap \overline{B}$. Can you spot the problem(s) in the following 'proofs' of this? These proofs use the alternative notation of $A - B$ for set difference.

⋆**Evaluate 5.48.** Use a set containment proof to prove that if A and B are sets, then $A - B = A \cap \overline{B}$.

Proof 1: Assume $x \in \{A - B\}$ so $x \in A$ and x is not $\in B$. This means $x \in A$ and $\overline{B}$. Therefore $x \in A \cap \overline{B}$. Thus $A - B = A \cap \overline{B}$.

Evaluation ____

Proof 2: $\overline{B}$ is the other part of the universal that does not contain any part of B. $A \cup \overline{B}$ means all intersection part of A and the universal that does not contain any part of B. Therefore it returns all elements that are in A but not in B which are $A - B$. Thus, $A - B = A \cap \overline{B}$.

Evaluation ____

Proof 3: To prove that $A - B = A \cap \overline{B}$, first let $x \in A - B$. By definition of the difference of sets, this means that x is an element of A that is not in B, or in other words, $x \in A$ and $x \notin B$. This is the same as $x \in A \cap \overline{B}$, thus proving that $A - B \subseteq A \cap \overline{B}$.
Now let $x \in A \cap \overline{B}$. This means that $x \in A$ and $x \notin B$, so it is in A, but not in B, which is what we just proved in the previous statement, thus proving that $A - B = A \cap \overline{B}$.

Evaluation ____

Sometimes we can do a set containment proof in one step instead of two. This only works if every step of the proof is reversible. We illustrate this idea next.

Example 5.49. Prove that $A \setminus (B \cup C) = (A \setminus B) \cap (A \setminus C)$.

Proof: We have

$$\begin{aligned} x \in A \setminus (B \cup C) &\leftrightarrow x \in A \wedge x \notin (B \vee C) \\ &\leftrightarrow (x \in A) \ \wedge \ ((x \notin B) \ \wedge \ (x \notin C)) \\ &\leftrightarrow (x \in A \ \wedge \ x \notin B) \ \wedge \ (x \in A \ \wedge \ x \notin C) \\ &\leftrightarrow (x \in A \setminus B) \ \wedge \ (x \in A \setminus C) \\ &\leftrightarrow x \in (A \setminus B) \cap (A \setminus C). \end{aligned}$$

$\square$

Note: *The proof in the previous example works because every step is reversible. You can only write something like '$\alpha \leftrightarrow \beta$' in a proof if $\alpha \to \beta$ and $\beta \to \alpha$ are both true. When attempting to shortcut proofs with this technique, make sure each step truly is reversible.*

⋆**Fill in the details 5.50.** Use a set containment proof to show that

$$(A \cup B) \cap C = (A \cap C) \cup (B \cap C).$$

Solution: We have,

$x \in (A \cup B) \cap C$

$\leftrightarrow$ $x \in (A \cup B) \wedge$ __________________ by definition of intersection

$\leftrightarrow$ $(x \in A \vee$ __________________ $) \wedge x \in C$ by __________________

$\leftrightarrow$ $(x \in A \wedge x \in C) \vee$ ________________________ by __________________

$\leftrightarrow$ ________________________ $\vee (x \in B \cap C)$ by __________________

$\leftrightarrow$ $x \in (A \cap C) \cup (B \cap C)$. by __________________

Example 5.51. In Java, the `TreeSet` class is one implementation of a *set* that has several methods with perhaps unfamiliar names, but they do what should be familiar things. Let's discuss a few of them.[a] Let `A` and `B` be `TreeSet`s.

(a) The method `retainAll(TreeSet other)` "*retains only the elements in this TreeSet that are contained in the* `other` *TreeSet. In other words, removes from this TreeSet all of its elements that are not contained in* `other`." It is not too difficult to see that `A.retainAll(B)` is computing $A \cap B$.[b]

(b) The method `boolean containsAll(TreeSet other)` "*returns true if this set contains all of the elements of* `other` *(and false otherwise)*." Thus, `A.containsAll(B)` returns true iff $B \subseteq A$.

(c) Even without documentation, it seems likely that `A.size()` is determining $|A|$.

(d) It is also seems likely that `A.isEmpty()` is determining if $A = \emptyset$.

---

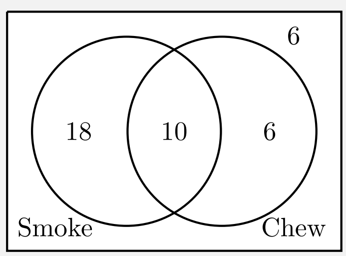
[a]The method signatures and documentation have been modified from the official definition so we can focus on the point at hand.

[b]Technically it is doing more than that. It is storing the result in A. So it is like it is doing $A = A \cap B$, where $=$ here means assignment, not equals.

Sometimes you need to find the number of elements in the union of several sets. This is easy if the sets do not intersect. If they do intersect, more care is needed to make sure no elements are missed or counted more than once. In the following examples we will use Venn diagrams to help us do this correctly. Later, we will learn about a more powerful tool to do this—*inclusion-exclusion*.

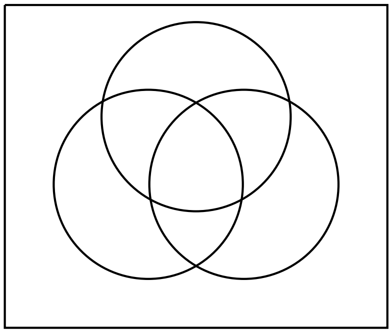
Example 5.52. Of 40 people, 28 smoke and 16 chew tobacco. It is also known that 10 both smoke and chew. How many among the 40 neither smoke nor chew?

Solution: We fill up the Venn diagram below as follows. Since $|Smoke \cap Chew| = 10$, we put a 10 in the intersection. Then we put $28 - 10 = 18$ in the part that *Smoke* does not overlap *Chew* and $16 - 10 = 6$ in the part of *Chew* that does not overlap *Smoke*. We have accounted for $10 + 18 + 6 = 34$ people that are in at least one of the sets. The remaining $40 - 34 = 6$ people outside these sets don't smoke or chew (and probably don't date girls who do).



We should note that we truly hope that these numbers are not representative of the number of people who smoke and/or chew in real life. It's bad for you. Don't do it. Really.

⋆**Exercise 5.53.** In a group of 30 people, 8 speak English, 12 speak Spanish and 10 speak French. It is known that 5 speak English and Spanish, 7 Spanish and French, and 5 English and French. The number of people speaking all three languages is 3. How many people speak at least one of these languages?



Definition 5.54. *The* **Cartesian product** *of sets A and B is the set $A \times B = \{(a, b) | a \in A \wedge b \in B\}$. In other words, it is the set of all ordered pairs of elements from A and B.*

Example 5.55. If $A = \{1, 2, 3\}$ and $B = \{a, b\}$, then

$$A \times B = \{(1, a), (1, b), (2, a), (2, b), (3, a), (3, b)\}, \text{ and}$$

$$B \times A = \{(a, 1), (a, 2), (a, 3), (b, 1), (b, 2), (b, 3)\}.$$

Notice that $A \times B \neq B \times A$. If $A \neq B$, this is always the case.

⋆**Exercise 5.56.** Let $A = \{1, 2, 3, 4\}$, and $B = \{3\}$. Compute $A \times B$.

$A \times B =$ ____

Definition 5.57. *If A is a set, then $A^2 = A \times A$, and $A^n = A \times A^{n-1}$.*

Example 5.58. If $B = \{a, b\}$then

$$B^2 = \{(a, a), (a, b), (b, a), (b, b)\}, \text{ and}$$

$$B^3 = \{(a, a, a), (a, b, a), (b, a, a), (b, b, a), (a, a, b), (a, b, b), (b, a, b), (b, b, b)\}$$

⋆**Exercise 5.59.** Let $A = \{0, 1\}$. Find A^2 and A^3.

$A^2 =$ ______________________________

$A^3 =$ ______________________________

____

It shouldn't be too difficult to convince yourself of the following.

Theorem 5.60. *If A and B are finite sets with $|A| = n$ and $|B| = m$, then $|A \times B| = n \cdot m$.*

Example 5.61. Let A and B be finite sets with $|A| = 100$ and $|B| = 5$. Then $|A \times B| = 100 * 5 = 500$, $|A^2| = 100 * 100 = 10,000$, and $|B^4| = 5^4 = 625$.

⋆**Exercise 5.62.** Let A, B, and C be sets with $|A| = 10$, $|B| = 50$, and $|C| = 20$. Determine the following

(a) $|A \times B| =$ ______________________________

(b) $|A \times C| =$ ______________________________

(c) $|A^2| =$ ______________________________

(d) $|B^3| =$ ______________________________

(e) $|A \times B \times C| =$ ______________________________

⋆**Evaluate 5.63.** If $A \times B = \emptyset$, what can we conclude about A and B?

Solution 1: Assume A and B are not empty. We know the Cartesian product of A and B, denoted by $A \times B$, is the set of all ordered pairs (a, b), where $a \in A$ and $b \in B$. Therefore, we can conclude that our assumption was incorrect because if each set is not empty, (a, b) is in the cross product, but $A \times B = \emptyset$, so at least one of the sets must be empty.

Evaluation ____

____

Solution 2: Notice that if $A = \emptyset$ and $B = \emptyset$, $A \times B = \emptyset$. Therefore, if $A \times B = \emptyset$, then $A = \emptyset$ and $B = \emptyset$.

Evaluation ____

____

Solution 3: We can conclude that both A and B are empty. I'll prove it by contradiction. Assume that $A \times B = \emptyset$, but that it is not the case that both A and B are empty. Then neither A nor B is empty. But then there is some $a \in A$ and some $b \in B$, and $(a, b) \in A \times B$, which implies that $A \times B \neq \emptyset$. This contradicts our assumption. Therefore both A and B are empty.

Evaluation ____

____

Solution 4: At least one of A or B is empty by contradiction. Assume that $A \times B = \emptyset$, but that it is not the case that at least one of A or B is empty. Then neither A nor B is empty. Then there is some $a \in A$ and some $b \in B$. But then $(a, b) \in A \times B$, which implies that $A \times B \neq \emptyset$. This contradicts our assumption. Therefore at least one of A or B is empty.

Evaluation ____

____

5.3 Functions

This section is meant as a review of what you hopefully already learned in an earlier course, probably in high school. Thus, it is pretty brief. But we do try to cover all of the important material and provide enough examples to illustrate the concepts.

Definition 5.64. *Let A and B be sets. Then a* **function** *f from A to B assigns to each element of A exactly one element from B. We write $f : A \to B$ if f is a function from A to B. If $a \in A$ and f assigns to a the value $b \in B$, we write $f(a) = b$. We also say that f* **maps** *a to b.*

If $A = B$, we sometimes say f is a function **on** *A.*

Example 5.65. If $A = B = \mathbb{N}$, we can define a function $f : A \to B$ by $f(x) = x^2$. Then $f(1) = 1$, $f(2) = 4$, $f(3) = 9$, etc. Although $f(x)$ is defined for all $x \in A$, not every $b \in B$ is mapped to by f. For instance, there is no $a \in A$ for which $f(a) = 5$.

Example 5.66. Notice that we can define $f(x) = \sqrt{x}$ on the positive real numbers, but we *cannot* define it on the positive integers since $\sqrt{2}$ is not an integer. Similarly, since $\sqrt{-1} = i \notin \mathbb{R}$, we cannot define it on the real numbers. We *can* let it be a function from $\mathbb{R}$ to $\mathbb{C}$, though. But we won't because this course is complex enough even without complex numbers.

Definition 5.67. *Let f be a function from A to B.*

1. *We call A the* **domain** *of f.*
2. *We call B the* **codomain** *of f.*
3. *The* **range** *of f is the set $\{b | f(a) = b \text{ for some } a \in A\}$. In other words the range is the subset of B that are actually mapped to by f.*

Example 5.68. Let $A = B = \mathbb{N}$ and $f : A \to B$ be defined by $f(x) = x^2$. Then the domain and codomain of f are both $\mathbb{N}$, and the range is $\{a^2 | a \in \mathbb{N}\}$, which is a proper subset of the codomain.

Figure 5.3 gives a pictorial representation of a function. Notice that in this example every element in A has precisely one arrow going from it. So if I ask "what is $f(x)$?", there is always an answer and it is always unique. On the other hand, there is a point in B that has two arrows going to it and several points that have no arrows going to them. This is fine.

Figure 5.4 does not represent a function since there are several points in A which have two arrows going from them and several with no arrows at all. The problem here is that if I ask "what is $f(x)$?", sometimes there is no answer and sometimes there are multiple answers. Thus, f would not represent a function.

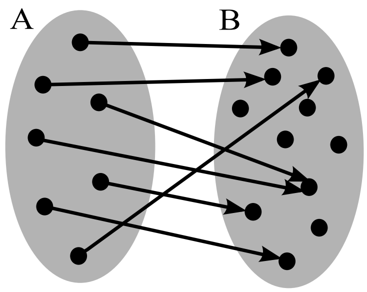


Figure 5.3: A pictorial representation of a function from A to B.

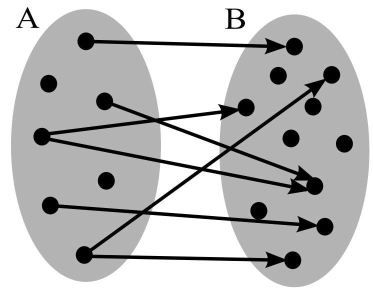


Figure 5.4: This picture does *not* represent a function.

Note: *In figures 5.3 and 5.4, the dots represent all of the elements of the sets A and B and the gray ovals are mainly there to help identify which dots are in which set. However, in these sorts of diagrams it is more common for the dots to represent only* some *of the elements. You need to let the context help you determine how to properly interpret these diagrams.*

Example 5.69. Give a formal definition of a function that assigns to an age the number of complete decades someone of that age has lived. For instance, $f(34) = 3$ and $f(5) = 0$. Be sure to indicate what the domain and codomain are.

Solution: It isn't hard to see that the domain and codomain are both $\mathbb{N}$. Thus we want a function $f : \mathbb{N} \to \mathbb{N}$. One way to define f is by $f(x) = \lfloor x/10 \rfloor$.

⋆**Exercise 5.70.** Give a formal definition of a function that returns the parity of an integer. That is, it returns 0 for even numbers and 1 for odd numbers. Be sure to indicate what the domain and codomain are.

Answer ____

____

Definition 5.71. *Let $f : A \to B$ be a function.*

- *f is said to be* **injective** *or* **one-to-one** *if and only if $f(a) = f(b)$ implies that $a = b$. In other words, f maps every element of A to a* different *element of B.*
- *f is said to be* **surjective** *or* **onto** *if and only if for every $b \in B$, there exists some $a \in A$ such that $f(a) = b$. In other words, every element in B gets mapped to by some element in A.*
- *f is said to be* **bijective** *or a* **one-to-one correspondence** *if it is both injective and surjective.*

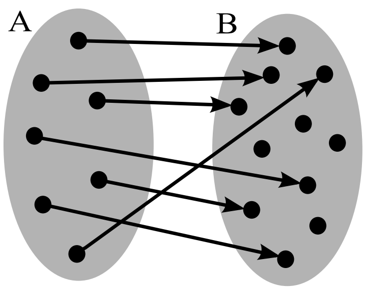


Figure 5.5: A pictorial representation of a *one-to-one* function.

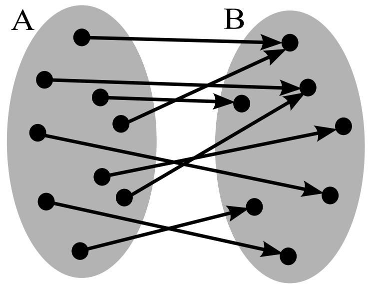


Figure 5.6: A pictorial representation of an *onto* function.

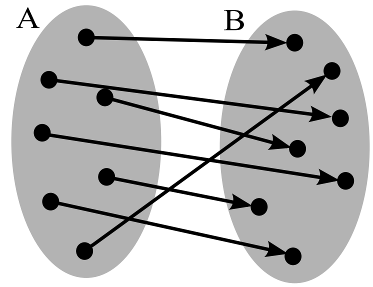


Figure 5.7: A pictorial representation of an *bijective* function.

Procedure 5.72. *To show that a function f is one-to-one, you just need to show that whenever $f(a) = f(b)$, then $a = b$.*

Example 5.73. Let $f(x) = 2x - 3$ be a function on the integers. Show that f is one-to-one.

Solution: Let $a, b \in \mathbb{Z}$ and assume that $f(a) = f(b)$. Then $2a - 3 = 2b - 3$. Adding 3 to both sides, we get $2a = 2b$. Dividing both sides by two, we obtain $a = b$. Therefore, $f(x) = 2x - 3$ is one-to-one.

⋆**Question 5.74.** Previously we mentioned that 'working both sides' was not an appropriate proof technique. Why is it O.K. in the previous example?

Answer ______________________________

⋆**Exercise 5.75.** Prove that $f(x) = 5x$ is one-to-one over the real numbers.

Proof ______________________________

Procedure 5.76. *To show that a function f is* **not** *one-to-one, we simply need to find two values $a \neq b$ in the domain such that $f(a) = f(b)$. That is, we just need to show that there are two different numbers in the domain that are mapped to the same value in the codomain.*

Example 5.77. Let $f(x) = x^2$ be a function on the integers. Show that f is not one-to-one.

Solution: Notice that $f(-1) = f(1) = 1$. Thus, $f(x)$ is not one-to-one.

⋆**Exercise 5.78.** Let $f(x) = \lfloor x \rfloor$ be a function on $\mathbb{R}$. Prove that f is not one-to-one.

Proof ____

____

Procedure 5.79. *To show that a function f is onto, we need to show that for an arbitrary $b \in B$, there is some $a \in A$ such that $f(a) = b$. That is, show that every value in B is mapped to by f.*

Example 5.80. Let $f(x) = x^3$ be a function on the real numbers. Show that f is onto.

Solution: Let $b \in \mathbb{R}$. Then $f\left(\sqrt[3]{b}\right) = \left(\sqrt[3]{b}\right)^3 = b^{3/3} = b$. Since every $b \in \mathbb{R}$ is mapped to (from $\sqrt[3]{b}$), f is onto.

⋆**Exercise 5.81.** Let $f(x) = 2x + 1$ be a function on $\mathbb{R}$. Show that f is onto.

Proof ____

____

____

____

Procedure 5.82. *To show that a function f is* **not** *onto, we just need to find some $b \in B$ such that there is no $a \in A$ with $f(a) = b$. In other words, we just need to find one value that isn't mapped to by f.*

Example 5.83. Let $f(x) = x^3$ be a function on the integers. Show that f is not onto.

Solution: There is no integer a such that $a^3 = 2$. In other words, 2 is not mapped to. Thus, $f(x)$ is not onto.

⋆**Exercise 5.84.** Let $f(x) = \lfloor x \rfloor$ be a function on $\mathbb{R}$. Prove that f is not onto.

Proof ______________________________

It is important to remember that whether or not a function is one-to-one or onto might depend on the domain/codomain over which the function is defined. For instance, notice that in the last two examples we used the same function but on different domains/codomains. In one case the function was onto, and in the other case it wasn't.

⋆**Exercise 5.85.** Consider the function $f(x) = x^2$.

(a) Prove or disprove that $f(x) = x^2$ is one-to-one on $\mathbb{Z}$.

Answer ______________________________

(b) Prove or disprove that $f(x) = x^2$ is one-to-one on $\mathbb{R}$.

Answer ______________________________

(c) Prove or disprove that $f(x) = x^2$ is one-to-one on $\mathbb{N}$.

Answer ______________________________

⋆**Exercise 5.86.** Determine which of the following functions from $\mathbb{Z}$ to $\mathbb{Z}$ is one-to-one and/or onto. Prove your answers.

(a) $f(x) = x + 2$

Answer ______________________________

(b) $g(x) = x^2$

Answer ______________________________

(c) $h(x) = 2x$

Answer ______________________________

(d) $r(x) = \lfloor x/2 \rfloor$

Answer ______________________________

The functions in the previous exercise were specifically chosen to demonstrate that all four possibilities of being or not being one-to-one and onto (one-to-one and onto, one-to-one and not onto, not one-to-one but onto, and not one-to-one or onto) are possible.

The following theorem should come as no surprise if you take a few minutes to think about it (and you *should* take a few minutes to think about it until you are convinced it is correct).

Theorem 5.87. *Let $f : A \to B$ be a function, and let A and B be finite.*

1. *If f is one-to-one, then $|A| \leq |B|$.*
2. *If f is onto, then $|A| \geq |B|$.*
3. *If f is bijective, then $|A| = |B|$.*

⋆**Exercise 5.88.** Let's test your understanding of the material so far. Answer each of the following true/false questions, giving a very brief justification/counterexample.

(a) ___If $f : A \to B$ is onto, then the domain and range are not only the same size, but they are the same set.

(b) ___If $f : A \to A$, then f must be one-to-one and onto.

(c) ___If $f : A \to B$ is both one-to-one and onto, then A and B have the same number of elements.

(d) ___Let $f(1) = 2$ and $f(1) = 3$. Then f is a valid function.

(e) ___Let $f : \mathbb{R} \to \mathbb{R}$ be defined by $f(x) = x^3$. Then f is one-to-one and onto.

(f) ___Let $f : \mathbb{R}^+ \to \mathbb{R}$ be defined by $f(x) = \sqrt{x}$. Then f is a function that is neither one-to-one nor onto.

(g) ___The range of a function is always a subset of the codomain.

(h) ___A function that is one-to-one is guaranteed to be onto.

(i) ___Let $a, b \in \mathbb{Z}$, with $a \neq 0$, and define $f : \mathbb{Z} \to \mathbb{Z}$ by $f(x) = ax + b$. Then f is one-to-one and onto.

(j) ___Let $a, b \in \mathbb{Z}$, with $a \neq 0$, and define $f : \mathbb{N} \to \mathbb{N}$ by $f(x) = ax+b$. Then f is one-to-one and onto.

(k) ___Let $a, b \in \mathbb{R}$, with $a \neq 0$, and define $f : \mathbb{R} \to \mathbb{R}$ by $f(x) = ax+b$. Then f is one-to-one and onto.

Definition 5.89. *Let f be a one-to-one correspondence from A to B. The* **inverse** *of f, denoted by f^{-1}, is the function such that $f^{-1}(b) = a$ whenever $f(a) = b$.*

A function that has an inverse is called **invertible**. *Said another way, a function is* **invertible** *if and only if it is one-to-one and onto.*

Note: *It is important to note that the function f^{-1} is not the same thing as $1/f$. This is an unfortunate case when a notation can be interpreted in two different ways. That is, in some contexts, a^{-1} means the inverse function and in other contexts it means $1/a$. Usually the context will help you determine which one is the correct interpretation.*

Procedure 5.90. *One method of finding the inverse of a function is to replace $f(x)$ (or whatever the name of the function is) with y and solve for x (or whatever the variable is). Finally, replace y with x and you have the inverse.*

Example 5.91. Let $f : \mathbb{Z} \to \mathbb{Z}$ be defined by $f(x) = x + 2$. Notice that f is a one-to-one correspondence, so it has an inverse. We let $y = x+2$. Solving for x, we get $x = y-2$. Thus, $f^{-1}(x) = x - 2$.

Example 5.92. Let $f : \mathbb{R} \to \mathbb{R}$ be defined by $f(x) = x^2$. Then f does not have an inverse since it is not one-to-one.

Example 5.93. Let $f : \mathbb{R} \to \mathbb{R}$ be defined by $f(x) = x^3$. We leave it to the reader to prove that f is one-to-one and onto. Given that, we can find it's inverse.

Let $y = x^3$. Taking the third root of both sides, we obtain $\sqrt[3]{y} = \sqrt[3]{x^3} = x$. Or $x = \sqrt[3]{y}$. Thus, the inverse of f is given by $f^{-1}(x) = \sqrt[3]{x}$.

⋆**Exercise 5.94.** Let $f(x) = 3x - 5$ be a function over $\mathbb{R}$. Prove that f has an inverse and then find it.

Definition 5.95. *Let g be a function from A to B and f a function from B to C. The* **composition** *of f and g, denoted by $f \circ g$, is defined as $(f \circ g)(x) = f(g(x))$ for any $x \in A$.*

In other words, to compose f with g, we *first* compute $g(x)$. Then we plug in $g(x)$ into the formula for f.

Note: *Look closely at the notation. $f \circ g$ has f before g, so it might seem like it should be $g(f(x))$–in other words, apply f first, then then g. But that is* not *how it is defined.*

Also notice that to compose f with g, it is necessary that the range of g is a subset of the domain of f since otherwise it would be impossible to compute.

Example 5.96. Let f and g be functions on $\mathbb{Z}$ defined by $f(x) = x^2$ and $g(x) = 2x - 5$. Compute $f \circ g$ and $g \circ f$, simplifying your answers.

Solution:

$$\begin{aligned}(f \circ g)(x) &= f(g(x)) = f(2x-5) = (2x-5)^2 = 4x^2 - 20x + 25. \\ (g \circ f)(x) &= g(f(x)) = g(x^2) = 2x^2 - 5.\end{aligned}$$

Notice that in the previous example, $f \circ g \neq g \circ f$. In other words, the order in which we compose functions matters since the result is not always the same (although occasionally it is).

⋆**Exercise 5.97.** Let f and g be functions on $\mathbb{R}$ defined by $f(x) = \lfloor x \rfloor$ and $g(x) = x/2$. Compute $f \circ g$ and $g \circ f$, simplifying your answers.

$(f \circ g)(x) =$ ______________________________

$(g \circ f)(x) =$ ______________________________

Example 5.98. Let f be a function from B to C, and g be a function from A to B. If both f and g are one-to-one, prove that $f \circ g$ is one-to-one.

Direct Proof:
For any distinct elements $x, y \in A$, $g(x) \neq g(y)$, since g is one-to-one. Since f is also one-to-one, then $f(g(x)) \neq f(g(y))$, which is the same as $(f \circ g)(x) \neq (f \circ g)(y)$. Therefore $f \circ g$ is one-to-one. □

Proof by Contradiction:
Assume $f \circ g$ is not one-to-one. Then there exist distinct elements $x, y \in A$ such that $(f \circ g)(x) = (f \circ g)(y)$. This is equivalent $f(g(x)) = f(g(y))$. Since f is one-to-one, it must be the case that $g(x) = g(y)$. But $x \neq y$, and g is one-to-one, so $g(x) \neq g(y)$. This is a contradiction. Therefore $f \circ g$ is one-to-one. □

Definition 5.99. *We define the* **identity function**, $\iota_A : A \to A$, *by* $\iota_A(x) = x$. *The subscript can be omitted if the domain/codomain is clear.*

Theorem 5.100. *Let f be an invertible function from A to B. Then $f \circ f^{-1} = \iota_B$ and $f^{-1} \circ f = \iota_A$.*

Proof: *Let $a \in A$ and define $b = f(a)$. Then by definition, $f^{-1}(b) = a$, so $(f^{-1} \circ f)(a) = f^{-1}(f(a)) = f^{-1}(b) = a$. Thus, $f^{-1} \circ f = \iota_A$.*

Conversely, if $b \in B$ and we define $a = f^{-1}(b)$, then $(f \circ f^{-1})(b) = f(f^{-1}(b)) = f(a) = b$. Thus, $f \circ f^{-1} = \iota_B$. □

Example 5.101. Prove or disprove that $f(x) = 2x + 1$ and $g(x) = 2x - 1$, defined over the real numbers, are inverses.

Solution: Notice that $(f \circ g)(x) = f(2x - 1) = 2(2x - 1) + 1 = 4x - 1 \neq x$. According to Theorem 5.100, this implies that f and g are not inverses.

⋆**Exercise 5.102.** Let's test your understanding of the material so far. Answer each of the following true/false questions, giving a very brief justification/counterexample.

(a) ____Let $a, b \in \mathbb{Z}$ and define $f : \mathbb{Z} \to \mathbb{Z}$ be defined by $f(x) = ax + b$. Then f is invertible.

(b) ____Let $a, b \in \mathbb{Z}$ and define $f : \mathbb{N} \to \mathbb{N}$ be defined by $f(x) = ax + b$. Then f is invertible.

(c) ____Let $a, b \in \mathbb{R}$ and define $f : \mathbb{R} \to \mathbb{R}$ be defined by $f(x) = ax + b$. Then f is invertible.

(d) ___If $f(x) = x^2$, then $f^{-1}(x) = 1/x^2$.

(e) ___Let n be a positive integer. Then the function $\sqrt[n]{x}$ is invertible on $\mathbb{R}$.

(f) ___Let n be a positive integer. Then the function $\sqrt[n]{x}$ is invertible on $\mathbb{N}$.

(g) ___Let n be a positive integer. Then the function $\sqrt[n]{x}$ is invertible on $\mathbb{R}^+$ (the positive real numbers).

(h) ___Let f and g be functions on $\mathbb{Z}^+$ defined by $f(x) = x^2$ and $g(x) = 1/x$. Then $f \circ g = g \circ f$.

(i) ___Let f and g be functions on $\mathbb{Z}$ defined by $f(x) = (x+1)^2$ and $g(x) = x + 1$. Then $f \circ g = g \circ f$.

(j) ___Let $f(x) = \lfloor x \rfloor$ and $g(x) = \lceil x \rceil$ be defined on the real numbers. Then $f \circ g = g \circ f$.

(k) ___Let $f(x) = \lfloor x \rfloor$ and $g(x) = \lceil x \rceil$ be defined on the real numbers. Then f and g are inverses of each other.

(l) ___Let $f(x) = x^2$ and $g(x) = \sqrt{x}$ be defined over the positive real numbers. Then f and g are inverses of each other.

5.4 Partitions and Equivalence Relations

Partitions and equivalence relations are useful in computer science in several contexts. One of the most obvious is software testing. When creating test cases, you always want to ensure that you are covering 'all of the cases'. But what does that mean? It means you are thinking about how to *partition* all of the possible inputs into several sets, where the elements in one set are somehow different from those in another set, and are quite a lot like the other elements in the set. Let's see an example.

Example 5.103. Consider the following function that returns $n!$ if $n \geq 0$, and returns -1 if $n < 0$ ($n!$ is undefined for negative values of n, but we have to return something, so why not a negative number?)

```
int factorial(int n) {
    if(n<0)       { return -1; }
    else if(n==0) { return 1; }
    else {
        int fact = 1;
        for(int i=1;i<=n;i++) {
            fact = fact*i;
        }
        return fact;
    }
}
```

What values of n should we use to test `factorial`?

Solution: There seems to be three different types of values based on the structure of the code: 0, numbers less than 0, and numbers greater than 0. So we should test at least one number from each of these sets. Since boundaries can sometimes cause problems, we should include those. In light of this, we might test 0, -1, -2, -10, 1, 2, and 8. Since these cover all of the cases, they should provide pretty good evidence of whether or not `factorial` is implemented properly.[a]

[a] But remember, testing never *proves* that code is correct!

Definition 5.104. *Let $S \neq \varnothing$ be a set. A* **partition** *of S is a collection of non-empty, pairwise disjoint subsets of S whose union is S.*

Example 5.105. Define $\mathbb{E} = \{2k : k \in \mathbb{Z}\}$ and $\mathbb{O} = \{2k+1 : k \in \mathbb{Z}\}$. Clearly $\mathbb{E}$ is the set of even integers and $\mathbb{O}$ is the set of odd integers. Since $\mathbb{E} \cap \mathbb{O} = \varnothing$ and $\mathbb{E} \cup \mathbb{O} = \mathbb{Z}$, $\{\mathbb{E}, \mathbb{O}\}$ is a partition of $\mathbb{Z}$. Put another way, we can partition the integers based on parity.

Example 5.106. We can partition the socks in our sock drawer by color. In other words, we put all of the black socks in one set, the white ones in another, the green ones in another, etc. For simplicity, we can put all of the multi-color socks in a single set.

Example 5.107. We can partition the set of all humans by putting each person into a set based on the first letter of their first name. So *Adam* and *Adele* go into set A and *Zeek* goes into set Z, for instance. The sets in the partition are $A, B, \ldots Z$.[a]

[a] For simplicity, we assume everyone's name is written using the Roman alphabet.

Example 5.108. Let $A = \{1,5,8\}$, $B = \{2,3\}$, $C = \{4\}$, $D = \{6,9\}$, and $E = \{7,10,11,12\}$. Then the sets A, B, C, D, and E form a partition of the set $\{1,2,3,4,5,6,7,8,9,10,11,12\}$.

Example 5.109. When choosing test cases for the `factorial` method in Example 5.103, we thought about 3 subsets of $\mathbb{Z}$: $\{0\}$, $\mathbb{Z}^+$, and $\mathbb{Z}^-$. These cases form a partition of $\mathbb{Z}$ since they are disjoint and $\mathbb{Z} = \{0\} \cup \mathbb{Z}^+ \cup \mathbb{Z}^-$. This is good since it means we covered at least one value of the different types, and we didn't 'overtest' any of the cases by unknowingly duplicating values from the same case.

⋆**Exercise 5.110.** You need to decide on test cases for a method `int maximum(int a,int b)` that returns the maximum of its arguments. How would you partition the possible inputs into sets such that if it is correct for one (or a few) tests of cases from that set, it is probably correct for the rest of the cases in that set? Notice that the set of inputs is $\mathbb{Z} \times \mathbb{Z}$.

Answer ______________________________

____

____

Most of the partitions we talk about will be based on some meaningful characteristic of the elements of a set–like parity, color, or sign. But this is not inherent in the definition. For instance, the sets in the partition from Example 5.108 do not seem to have any significant meaning. Some, like the one in Example 5.105, will have a precise mathematical definition. Others, like the one in Examples 5.106 will not.

⋆**Exercise 5.111.** Define a partition on $\mathbb{Z}$ that contains more than one subset.

Answer ______________________________

____

____

Example 5.112. Let $3\mathbb{Z} = \{3k : k \in \mathbb{Z}\}$, $3\mathbb{Z}+1 = \{3k+1 : k \in \mathbb{Z}\}$, and $3\mathbb{Z}+2 = \{3k+2 : k \in \mathbb{Z}\}$.[a] Since

$$(3\mathbb{Z}) \cup (3\mathbb{Z}+1) \cup (3\mathbb{Z}+2) = \mathbb{Z} \text{ and}$$

$$(3\mathbb{Z}) \cap (3\mathbb{Z}+1) = \varnothing,\ (3\mathbb{Z}) \cap (3\mathbb{Z}+2) = \varnothing, (3\mathbb{Z}+1) \cap (3\mathbb{Z}+2) = \varnothing,$$

$\{3\mathbb{Z}, 3\mathbb{Z}+1, 3\mathbb{Z}+2\}$ is a partition of $\mathbb{Z}$.

[a] "The notation in this example may seem a bid odd at first. How are you supposed to interpret "$3\mathbb{Z}+1$"? Is this 3 times the set $\mathbb{Z}$ plus 1? What does it mean to do algebra with sets and numbers? I won't get into all of the technical details, but here is a short answer. You can think of "$3\mathbb{Z}+1$" as just a name. Sure, it may seem like an odd name, but why can't we name a set whatever we want? Some people name their kids *Jon Blake Cusack 2.0* and get away with it. You can also think of "$3\mathbb{Z}+1$" as describing how to create the set—by taking every element from $\mathbb{Z}$, multiplying it by 3, and then adding 1. Thus, you can think of "$3\mathbb{Z}+1$" as being both an algebraic expression and a name.

⋆**Exercise 5.113.** Let $\mathbb{I} = \mathbb{R} \setminus \mathbb{Q}$ (the set of irrational numbers). Prove that $\{\mathbb{Q}, \mathbb{I}\}$ is a partition of $\mathbb{R}$.

Proof ______________________________

____

____

Recall that when a list of number is given between parentheses (e.g. $(1,2,3)$), it typically denotes an ordered list. That is, the order that the element are listed matters. So, for instance, $(1,2)$ and $(2,1)$ are not the same thing.

Next we will develop an alternative way of thinking about partitions: equivalence relations. After defining some terms and providing a few examples, we will make the connection between partitions and equivalence relations more clear.

Definition 5.114. *Let A, B be sets. A* **relation** *(or* **binary relation***) from A to B is a subset of the Cartesian product $A \times B$.*

Given a relation R, we say that x is **related to** *y if $(x,y) \in R$. We sometimes write this as xRy. An alternative notation is $x \sim y$.*

If R is a relation from A to A, we sometimes say R is a relation **on** *A.*

Example 5.115. Let A be the set of all students at this school and B be the set of all courses at this school. We can define a relation R by saying that xRy if student x has taken course y. Said another way, we can define R by saying that $(x,y) \in R$ if student x has taken course y.

Example 5.116. We can define a relation $R = \{(a, a^2) : a \in \mathbb{Z}\}$. That is, x is related to y if $y = x^2$.

Example 5.117. We can define a relation on $\mathbb{Z}$ by saying that x is related to y if they have the same parity. Thus, $(2,0)$, $(234,-342)$, $(3,17)$ are all in R, but $(2,127)$ is not.

⋆**Question 5.118.** Define $R = \{(a, b) : a, b \in \mathbb{Z} \text{ and } a < b\}$. Is R a relation? Explain.

Answer ____

____

⋆**Question 5.119.** Is $\{(1, 2), (345, 7), (43, 8675309), (11, 11)\}$ a relation on $\mathbb{Z}^+$? Explain.

Answer ____

____

Definition 5.120. *A relation R on set A is said to be* **reflexive** *if for all $x \in A$, xRx (or $(x, x) \in R$).*

⋆**Exercise 5.121.** Let P be the set of all people. Which of the following relations on P are reflexive? Explain why or why not.

(a) $T = \{(a, b) : a, b \in P \text{ and } a \text{ is taller than } b\}$

(b) N is the relation with a related to b iff a's name starts with the same letter as b's name.

(c) C is the relation defined by $(a, b) \in C$ if a and b have been to the same city.

(d) $K = \{(a, b) : a, b \in P \text{ and } a \text{ does not know who } b \text{ is}\}$

(e) $R = \{(\text{Barack Obama}, \text{George W. Bush})\}$.

(a) T: ____

____

(b) N: ____

____

(c) C: ____

____

(d) K: ______________________________

____

(e) R: ______________________________

____

Definition 5.122. *A relation R on set A is said to be* **symmetric** *if for all $x, y \in A$, xRy implies yRx (or $(x, y) \in R$ implies $(y, x) \in R$).*

⋆**Exercise 5.123.** Which of the relations from Example 5.121 are symmetric? Explain why or why not.

(a) T: ______________________________

____

(b) N: ______________________________

____

(c) C: ______________________________

____

(d) K: ______________________________

____

(e) R: ______________________________

____

Definition 5.124. *A relation R on set A is said to be* **anti-symmetric** *if for all $x, y \in A$, xRy and yRx implies $x = y$ (or $(x, y) \in R$ and $(y, x) \in R$ implies $x = y$).*

⋆**Question 5.125.** Let R be a relation on $\mathbb{Z}$.

(a) If $(1,1) \in R$, can you tell whether or not R is anti-symmetric? Explain.

Answer ______________________________

____

(b) What if $(1,2)$ and $(2,1)$ are both in R? Can you tell whether or not R is anti-symmetric?

Answer ______________________________

____

⋆**Question 5.126.** An alternative definition of *anti-symmetric* is that if $x \neq y$, then (x,y) and (y,x) are not both in the relation. Why is this definition equivalent?

Answer ______________________________

____

Note: *This definition is sometimes misunderstood. Let's call elements of the form* (x,x) **diagonal** *elements and elements of the form* (x,y) *where* $x \neq y$ **off-diagonal** *elements.*[a] *Then the definition of anti-symmetric is only dealing with off-diagonal elements. It is saying nothing about the diagonal elements. In other words, it is* **not** *saying that* $(x,x) \in R$ *for any, let alone all, values of* x. *But it also isn't saying* $(x,x) \notin R$. *It is simply saying that the* only *way for both* (x,y) *and* (y,x) *to be in* R *is if* $x = y$.

The alternative definition given in the previous question may help a little. Notice that the definition there starts with 'if $x \neq y \ldots$*' So what does the definition say about the case* $x = y$*? Nothing. It never mentions it.*

You could redefine it as follows: R *is anti-symmetric if for all non-diagonal elements* $(x,y) \in R$, $(y,x) \notin R$. *But that can be problematic if you forget that* $x \neq y$ *is required.*

[a] These terms come from thinking about the elements of a relation as elements in a matrix indexed by the members of the set. If this doesn't make sense, don't worry too much about it.

⋆**Exercise 5.127.** Which of the relations from Example 5.121 are anti-symmetric? Explain why or why not.

(a) T: ______________________________

____

(b) N: ______________________________

____

(c) C: ______________________________

____

(d) K: ______________________________

____

(e) R: ______________________________

____

⋆**Question 5.128.** Answer each of the following. Include a brief justification/example.

(a) If a relation is not symmetric, is it anti-symmetric?

Answer ______________________________

____

(b) If a relation is not anti-symmetric, is it symmetric?

Answer ______________________________

____

(c) Can a relation be both symmetric and anti-symmetric?

Answer ______________________________

____

⋆**Exercise 5.129.** Give an example of a relation on any set of your choice that is both symmetric and anti-symmetric. Justify your answer.

Answer ______________________________

____

Definition 5.130. *A relation R on set A is said to be* **transitive** *if for all $x, y, z \in A$, xRy and yRz implies xRz (or $((x, y) \in R$ and $(y, z) \in R)$ implies $(x, z) \in R$).*

⋆**Exercise 5.131.** Which of the relations from Example 5.121 are transitive? Explain why or why not.

(a) T: ______________________________

____

(b) N: ______________________________

____

(c) C: ______________________________

____

(d) K: ______________________________

____

(e) R: ______________________________

____

Definition 5.132. *A relation which is reflexive, symmetric and transitive is called an* **equivalence relation**.

Example 5.133. Let $S =$ {All Human Beings}, and define the the relation M by $(a, b) \in M$ if a has the same (biological) mother[a] as b. Show that M is an equivalence relation.

Proof: (**Reflexive**) a has the same mother as a, so $(a, a) \in M$ and M is reflexive.
(**Symmetric**) If a has the same mother as b, then b clearly has the same mother as a. Thus, $(a, b) \in M$ implies $(b, a) \in M$, so M is symmetric.
(**Transitive**) If a has the same mother as b, and b has the same mother as c, then clearly a has the same mother as c. In other words, $(a, b) \in M$ and $(b, c) \in M$ implies that $(a, c) \in M$, so M is transitive.
Since M is reflexive, symmetric, and transitive, it is an equivalence relation. ☐

[a] The important assumption we are making is that each person has exactly one mother.

⋆**Exercise 5.134.** Which of the relations from Example 5.121 are equivalence relations? Explain why or why not.

(a) T: ______

______

(b) N: ______

______

(c) C: ______

______

(d) K: ______

______

(e) R: ______

______

Definition 5.135. *A relation which is reflexive, anti-symmetric and transitive is called a* **partial order***.*

⋆**Exercise 5.136.** Which of the relations from Example 5.121 are partial orders? Explain why or why not.

(a) T: ______

______

(b) N: ______

______

(c) C: ______________________________

____

(d) K: ______________________________

____

(e) R: ______________________________

____

⋆**Exercise 5.137.** Let X be a collection of sets. Let R be the relation on X such that A is related to B if $A \subseteq B$. Prove that R is a partial order on X.

Proof: (**Reflexive**) ______________________________

____

(**Anti-symmetric**) ______________________________

____

(**Transitive**) ______________________________

____

____

□

Labeling the lines of these proofs with what property we are proving isn't strictly necessary. However, it does make the proofs a little easier to read.

⋆**Exercise 5.138.** Consider the relation $R = \{(1,2),(1,3),(1,5),(2,2),(3,5),(5,5)\}$ on the set $\{1,2,3,4,5\}$. Prove or disprove each of the following.

(a) R is reflexive

Answer ______________________________

____

(b) R is symmetric

Answer ______________________________

____

(c) R is anti-symmetric

Answer ______________________________

____

(d) R is transitive

Answer ______________________________

____

(e) R is an equivalence relation

Answer ______________________________

____

(f) R is a partial order

Answer ______________________________

____

It turns out that congruence modulo n is an equivalence relation. (See Definition 3.13 if necessary).

Theorem 5.139. *Let n be a positive integer. Then $R = \{(a, b) : a \equiv b \pmod{n}\}$ is a relation on the set of integers. Show that R is an equivalence relation.*

Proof: *We need to show that R is reflexive, symmetric, and transitive.*
*(**Reflexive**) Clearly $a - a = 0 \cdot n$, so $a \equiv a \pmod{n}$. Thus, R is reflexive.*
*(**Symmetric**) Assume $(a, b) \in R$. Then $a \equiv b \pmod{n}$, which implies $a - b = kn$ for some integer k. So $b - a = (-k)n$, and since $-k$ is an integer, $b \equiv a \pmod{n}$. Therefore, $(b, a) \in R$. Thus, R is symmetric.*
*(**Transitive**) Assume $(a, b), (b, c) \in R$. Then $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$. Thus, $a - b = kn$ for some integer k and $b - c = ln$ for some integer l. Given these, we can see that*

$$a - c = (a - b) + (b - c) = kn + ln = (k + l)n.$$

Since $k + l$ is an integer, $a \equiv c \pmod{n}$. Thus $(a, c) \in R$, so R is transitive. $\square$

Notice that if we let $n = 2$ in the previous theorem, we essentially have the relation from Example 5.117.

⋆**Fill in the details 5.140.** Let R be the relation on the set of ordered pairs of positive integers (that is, $\mathbb{Z}^+ \times \mathbb{Z}^+$) such that $((a,b),(c,d)) \in R$ if and only if $ad = bc$. Show that R is an equivalence relation.[a]

Proof: We need to show that R is reflexive, symmetric, and transitive.

(**Reflexive**) Since $ab = ba$ for all positive integers, ____________________ $\in R$ for all (a,b). Thus R is reflexive.

(**Symmetric**) Assume $((a,b),(c,d)) \in R$. Then we know that $ad =$ _______.

We can rearrange this as $cb =$ _______. Thus, ______________ $\in R$, so R is

____________________.

(**Transitive**) Assume that $((a,b),(c,d)) \in R$ and $((c,d),(e,f)) \in R$. Then

we know that ____________________ and ____________________. Solving the

second for c, we get $c =$ ______________. Plugging it into the first we get

$ad =$ ______________. Multiplying both sides by f, and canceling the d on both

sides yields ____________________. Thus, ____________________ $\in R$, so R is transitive. □

[a] In this example, R is a relation on a set of ordered pairs. Thus, the elements of R are ordered pairs of ordered pairs. Don't let this confuse you. The elements of a relation are always ordered pairs. What each part of the pair is depends on the underlying set. If it is the set of animals, then the elements of the relation are ordered pairs of animals. If it is $\mathbb{Z}$, then the elements of the relation are ordered pairs of integers. And if it is $\mathbb{Z}^+ \times \mathbb{Z}^+$, then the elements of the relation are ordered pairs of ordered pairs of positive integers.

Definition 5.141. *Let R be an equivalence relation on a set S. Then the* **equivalence class of a**, *denoted by* $[a]$, *is the subset of S containing all of the elements that are related to a. More formally,*

$$[a] = \{x \in S : xRa\}.$$

If $x \in [a]$, we say that x is a **representative** *of the equivalence class $[a]$. Note that* any *element of an equivalence class can serve as a representative.*

Example 5.142. The equivalence class of 3 modulo 8 is $[3] = \{8k+3 : k \in \mathbb{Z}\}$. Notice that $[11] = \{8k+11 : k \in \mathbb{Z}\} = \{8k+3 : k \in \mathbb{Z}\} = [3]$. In fact, $[3] = [8l+3]$ for all integers l. In other words, any element of the form $8l+3$, where l is an integer, can serve as a representative of $[3]$. Further, we can call this class $[3]$, $[11]$, $[19]$, etc. It doesn't really matter since they all represent the same set of integers. Of course, $[3]$ is the most logical choice.

Example 5.143. Notice that if our relation is congruence modulo 3, we can define three equivalence classes:

$$\begin{aligned}[0] &= \{3k : k \in \mathbb{Z}\}, \\ [1] &= \{3k+1 : k \in \mathbb{Z}\}, \text{ and} \\ [2] &= \{3k+2 : k \in \mathbb{Z}\}.\end{aligned}$$

It isn't too difficult to see that $\mathbb{Z} = [1] \cup [2] \cup [3]$, and that these three sets are disjoint. In other words, the equivalence classes $\{[1], [2], [3]\}$ form a partition of $\mathbb{Z}$. As we will see shortly, this is not a coincidence.

Lemma 5.144. *Let R be an equivalence relation on a set S. Then two equivalence classes are either identical or disjoint.*

> **Proof:** *Let $a, b \in S$, and assume $[a] \cap [b] \neq \varnothing$. We need to show that $[a] = [b]$. First, let $x \in [a] \cap [b]$ (which exists since $[a] \cap [b] \neq \varnothing$). Then xRa and xRb, so by symmetry aRx and by transitivity aRb.*
>
> *Now let $y \in [a]$. Then yRa. Since we just showed that aRb, then yRb by transitivity. Thus $y \in [b]$. Therefore $[a] \subseteq [b]$.*
>
> *A symmetric argument proves that $[b] \subseteq [a]$. Therefore, $[a] = [b]$.* □

Let's bring together some of the examples of partitions with examples of equivalence relations and classes.

Example 5.145. We just saw that congruence modulo 3 is an equivalence relation with three equivalence classes, $\{3k : k \in \mathbb{Z}\}$, $\{3k+1 : k \in \mathbb{Z}\}$, and $\{3k+2 : k \in \mathbb{Z}\}$. In Example 5.112, we defined a partition of $\mathbb{Z}$ using these same three subsets.

Example 5.146. In Example 5.117 we defined a relation on $\mathbb{Z}$ based on parity. It is not difficult to see that the equivalence classes of that relation are $[0] = \mathbb{E}$ and $[1] = \mathbb{O}$. Notice these are the same subsets we used to partition $\mathbb{Z}$ in Example 5.105.

Example 5.147. In Example 5.107 we defined a partition of people according to the first letter of their first name. The sets in the partition were $A, B, \ldots, Z$.

We can define an equivalence relation on the set of all people by saying a is related to b if a's name starts with the same letter of the alphabet as b's name. In a series of previous exercises, you proved that this defines an equivalence relation. Notice that the equivalence classes are the sets $A, B, \ldots, Z$ (which we can think of as, for instance $[Adam], [Betty], \ldots, [Zeek]$). Again, these are the same sets that we used to partition people into in Example 5.107.

In these examples, there seems to be a connection between the equivalence classes of the relation and the sets in a partition. As the next theorem illustrates, this is no coincidence.

Theorem 5.148. *Let $S \neq \varnothing$ be a set. Every equivalence relation on S induces a partition of S and vice-verse.*

Proof: *By Lemma 5.144, if R is an equivalence relation on S then*

$$S = \bigcup_{a \in S} [a],$$

and $[a] \cap [b] = \varnothing$ if a is not related to b. This proves the first half of the theorem. Conversely, let

$$S = \bigcup_{\alpha} S_\alpha, \quad S_\alpha \cap S_\beta = \varnothing \text{ if } \alpha \neq \beta,$$

be a partition of S. We define the relation R on S by letting aRb if and only if they belong to the same S_α. Since the S_α are mutually disjoint, it is clear that R is an equivalence relation on S and that for $a \in S_\alpha$, we have $[a] = S_\alpha$. $\square$

Put in simple terms, equivalence classes of an equivalence relation and partitions of sets are essentially the same thing. The main difference is in how we are looking at it. When thinking about equivalence relations/classes, we are focused on what it means for two things to be related. When thinking about partitions, we are focused on what it means for an element to be in a particular subset of the partition.

Example 5.149. In light of Theorem 5.148, we can say that the relation defined by congruence modulo 4 partitions the set of integers into precisely 4 equivalence classes: [0], [1], [2], and [3]. That is, given any integer, it is contained in one (and only one) of these classes.

More generally, if $n > 2$, $\mathbb{Z}$ can be partitioned into n sets, $[0], [1], \ldots, [n-1]$, each of which is an equivalence class of the relation defined by congruence modulo n.

When we think about the partition, we are focused on the concept that each number x goes into one of the n subsets based on the value $x \bmod n$. On the other hand, when we think about the relation of congruence modulo n, we are focused on the idea that x and y are in the same equivalence class iff $x \equiv y \pmod{n}$.

5.5 Problems

Problem 5.1. Draw a Venn diagram showing $A \cap (B \cup C)$, where A, B, and C are sets.

Problem 5.2. Assume A, B, and C are sets. Prove each of the following using a set containment proof.

(a) $(A \cap B \cap C) \subseteq (A \cap B)$.

(b) $A \cap B \subseteq A \cup B$.

(c) $(A \cup B) \setminus (A \cap B) = (A \setminus B) \cup (B \setminus A)$.

(d) $(A - B) \setminus C \subseteq A \setminus C$.

(e) $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$.

Problem 5.3. Prove each of the following set identities using a set containment proof based on the basic definitions of $\cap$, $\cup$, etc. (see examples 5.46, 5.49, and 5.50).

(a) $A \cup (A \cap B) = A$.

(b) $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$

(c) $(A \setminus B) \setminus C = (A \setminus C) \setminus (B \setminus C)$.

(d) $\overline{A \cup (B \cap C)} = (\overline{C} \cup \overline{B}) \cap \overline{A}$. (This one is a little tricky.)

Problem 5.4. Rusty has 20 marbles of different colors: black, blue, green, and yellow. Seventeen of the marbles are not green, five are black, and 12 are not yellow. How many blue marbles does he have?

Problem 5.5. Let `A` and `B` be `TreeSet`s (See Example 5.51).

(a) The method `addAll(TreeSet other)` *adds all of the elements in* `other` *to this set if they're not already present.* What is the result of `A.addAll(B)` (in terms of `A` and `B` and set operators)?

(b) The method `removeAll(TreeSet other)` *removes from this set all of its elements that are contained in* `other`. What is the result of `A.removeAll(B)` (in terms of `A` and `B` and set operators)?

(c) Write `A.contains(x)` using set notation, where x is an element that can be stored in a `TreeSet`.

Problem 5.6. You need to settle an argument between your boss (who can fire you) and your professor (who can fail you). They are trying to decide who to invite to the Young Accountants Volleyball League. They want to invite freshmen who are studying accounting and are at least 6 feet tall. They have a list of all students.

(a) Your boss says they should make a list of all freshmen, a list of all accounting majors, and a list of everyone at least 6 feet tall. They should then combine the lists (removing duplicates) and invite those on the combined list. Is he correct? Explain. If he is not correct, describe in the simplest possible terms who ends up on his guest list.

(b) Your professor says they should make a list of everyone who is not a freshman, a list of everyone who does not do accounting, and a list of everyone who is under 6 feet tall. They should make a fourth list that contains everyone who is on all three of the prior lists. Finally, they should remove from the original list everyone on this fourth list, and invite the remaining students. Is he correct? Explain. If he is not correct, describe in the simplest possible terms who ends up on his guest list.

(c) Give a simple description of how the guest list should be created.

Problem 5.7. Let $a, b \in \mathbb{R}$, $a \neq 0$, and define $f : \mathbb{R} \to \mathbb{R}$ by $f(x) = ax + b$. Prove that f is one-to-one and onto.

Problem 5.8. Let a and b be real numbers with $a \neq 0$. Show that the function $f(x) = a\,x + b$ is invertible.

Problem 5.9. Prove or disprove: if a, b, and c are real numbers with $a \neq 0$, then the function $f(x) = a\,x^2 + b\,x + c$ is invertible.

Problem 5.10. Prove that if f and g are onto, then $f \circ g$ is also onto.

Problem 5.11. Let $f(x) = x + \lfloor x \rfloor$ be a function on $\mathbb{R}$. (This one is a little tricky.)

(a) Prove or disprove that f is one-to-one.

(b) Prove or disprove that f is onto.

(c) Prove or disprove that f is invertible.

Problem 5.12. Find the inverse of the function $f(x) = x^3 + 1$ over the real numbers.

Problem 5.13. Let f be the function on $\mathbb{Z}^+$ that maps x to the number of bits required to represent x in binary. For instance, $f(1) = 1$, $f(2) = 2$, $f(3) = 2$, $f(4) = 3$, $f(10) = 4$, etc. Hint: The number 2^n requires $n+1$ bits to represent (a single 1 followed by n zeros). You may be able to use this fact in one of your proofs.

(a) Prove or disprove that f is one-to-one.

(b) Prove or disprove that f is onto.

(c) Prove or disprove that f is invertible.

Problem 5.14.

Consider the relation $R = \{(1,2),(1,3),(3,5),(2,2),(5,5),(5,3),(2,1),(3,1)\}$ on the set $\{1,2,3,4,5\}$. Is R reflexive? symmetric? anti-symmetric? transitive? an equivalence relation? a partial order?

Problem 5.15. Let X be the set of all people. Which of the following are equivalence relations? Prove it.

(a) $R_1 = \{(a,b) \in X^2 | a \text{ and } b \text{ are the same height}\}$

(b) $R_2 = \{(a,b) \in X^2 | a \text{ is taller than } b\}$

(c) $R_3 = \{(a,b) \in X^2 | a \text{ is at least as tall as } b\}$

(d) $R_4 = \{(a,b) \in X^2 | a \text{ and } b \text{ have the same last name}\}$

(e) $R_5 = \{(a, b) \in X^2 | a \text{ has the same kind of pet as } b\}$

Problem 5.16. Repeat the previous problem, but which are partial orders? Prove it.

Problem 5.17. Define three different equivalence relations on the set of all TV shows. For each, give examples of the equivalence classes, including one representative from each. Prove that each is an equivalence relation.

Problem 5.18. Define a relation on the set of all Movies that is *not* an equivalence relation.

Problem 5.19. Let $A = \{1, 2, \ldots, n\}$. Let R be the relation on $P(A)$ (the power set of A) such that $a, b \in P(A)$ are related iff $|a| = |b|$. Prove that R is an equivalence relation. What are the equivalence classes of R?

Problem 5.20. The class `Relation` is a partial implementation of a relation on a set A. It has a list of `Element` objects.

- An `Element` stores an ordered pair from A. `Element` has methods `getFrom()` and `getTo()` (using the language of the directed graph representation). So if an `Element` is storing (a, b), `getFrom()` returns a and `getTo()` returns b. The constructor `Element(Object a, Object b)` creates an element (a, b).
- The `Relation` class has methods like `areRelated(Object a,Object b)`, `getElements( )`, and `getUniverse( )`.
- Methods in the `Relation` class can use `for(Element e : getElements())` to iterate over elements of the relation.
- Similarly, the loop `for(Object a : getUniverse())` iterates over the elements of A.

Given all of this, implement the following methods in the `Relation` class:

(a) `isReflexive()`

(b) `isSymmetric()`

(c) `isAntiSymmetric()`

Chapter 6

Sequences and Summations

6.1 Sequences

Definition 6.1. *A* **sequence** *of real numbers is a function whose domain is the set of natural numbers and whose output is a subset of the real numbers. We usually denote a sequence by one of the notations*

$$a_0, a_1, a_2, \dots$$

or

$$\{a_n\}_{n=0}^{+\infty}$$

or

$$\{a_n\}.$$

The last notation is just a shorthand for the second notation.

Note: *Since sequences are functions, sometimes function notation is used. That is, $a(n)$ instead of a_n.*

We will be mostly interested in two types of sequences. The first type are sequences that have an explicit formula for their n-th term. They are said to be in *closed form.*

Example 6.2. Let $a_n = 1 - \frac{1}{2^n}, n = 0, 1, \dots$. Then $\{a_n\}_{n=0}^{+\infty}$ is a sequence for which we have an explicit formula for the n-th term. The first five terms are

$$\begin{array}{lclclcl}
a_0 & = & 1 - \frac{1}{2^0} & = & 1 - 1 & = & 0, \\
a_1 & = & 1 - \frac{1}{2^1} & = & 1 - \frac{1}{2} & = & \frac{1}{2}, \\
a_2 & = & 1 - \frac{1}{2^2} & = & 1 - \frac{1}{4} & = & \frac{3}{4}, \\
a_3 & = & 1 - \frac{1}{2^3} & = & 1 - \frac{1}{8} & = & \frac{7}{8}, \\
a_4 & = & 1 - \frac{1}{2^4} & = & 1 - \frac{1}{16} & = & \frac{15}{16}.
\end{array}$$

Note: *Sometimes we may not start at $n = 0$. In that case we may write*

$$a_m, a_{m+1}, a_{m+2}, \dots,$$

or

$$\{a_n\}_{n=m}^{+\infty},$$

where m is a non-negative integer. Most sequences we will deal with will start with $m = 0$ or $m = 1$.

⋆**Exercise 6.3.** Let $\{x_n\}$ be the sequence defined by $x_n = 1 + (-2)^n, n = 0, 1, 2, \dots$. Find the first five terms of $\{x_n\}$.

(a) $x_0 =$ ____________________

(b) $x_1 =$ ____________________

(c) $x_2 =$ ____________________

(d) $x_3 =$ ____________________

(e) $x_4 =$ ____________________

⋆**Exercise 6.4.** Find the first five terms of the following sequences.

(a) $x_n = 1 + \left(-\frac{1}{2}\right)^n, n = 0, 1, 2, \dots$

$x_0 =$ ______________ $x_1 =$ ______________ $x_2 =$ ______________

$x_3 =$ ______________ $x_4 =$ ______________

(b) $x_n = n! + 1, n = 0, 1, 2, \dots$

$x_0 =$ ______________ $x_1 =$ ______________ $x_2 =$ ______________

$x_3 =$ ______________ $x_4 =$ ______________

(c) $x_n = \dfrac{1}{n! + (-1)^n}, n = 2, 3, 4, \ldots$

$x_2 =$ ______________________ $x_3 =$ ______________________ $x_4 =$ ______________________

$x_5 =$ ______________________ $x_6 =$ ______________________

(d) $x_n = \left(1 + \dfrac{1}{n}\right)^n, n = 1, 2, \ldots$

$x_1 =$ ______________________ $x_2 =$ ______________________ $x_3 =$ ______________________

$x_4 =$ ______________________ $x_5 =$ ______________________

The second type of sequence are defined recursively. That is, each term is based on previous term(s). We call these *recurrence relations*.

Example 6.5. Let

$$x_0 = 1, \quad x_n = \left(1 + \frac{1}{n}\right) x_{n-1}, \text{ for } n = 1, 2, \ldots.$$

Then $\{x_n\}_{n=0}^{+\infty}$ is a recursively defined sequence. The terms $x_1, x_2, \ldots, x_5$ are

$$\begin{array}{ccccccccc}
x_1 & = & \left(1+\frac{1}{1}\right) x_0 & = & \left(1+\frac{1}{1}\right) 1 & = & 1+1 & = & 2. \\
x_2 & = & \left(1+\frac{1}{2}\right) x_1 & = & \left(1+\frac{1}{2}\right) 2 & = & 2+1 & = & 3. \\
x_3 & = & \left(1+\frac{1}{3}\right) x_2 & = & \left(1+\frac{1}{3}\right) 3 & = & 3+1 & = & 4. \\
x_4 & = & \left(1+\frac{1}{4}\right) x_3 & = & \left(1+\frac{1}{4}\right) 4 & = & 4+1 & = & 5. \\
x_5 & = & \left(1+\frac{1}{5}\right) x_4 & = & \left(1+\frac{1}{5}\right) 5 & = & 5+1 & = & 6.
\end{array}$$

Notice that in the previous example, we gave an explicit definition of x_0. This is called an *initial condition*. Every recurrence relation needs one or more initial conditions. Without them, we have an abstract definition of a sequence, but cannot compute any values since there is no "starting point."

When we find an explicit formula (or closed formula) for a recurrence relation, we say we have *solved* the recurrence relation.

Example 6.6. Given the values we computed in Example 6.5, it seems relatively clear that $x_n = n + 1$ is a solution for that recurrence relation.

Note: *It is important to be careful about jumping to conclusions too quickly when solving recurrence relations.*[a] *Although it turns out that in the previous example, $x_n = n+1$ is the correct closed form (we will prove it shortly), just because it works for the first 5 terms does not necessarily imply that the pattern continues.*

[a] These comments also apply to other problems that involve seeing a pattern and finding an explicit formula.

⋆**Exercise 6.7.** Let $\{x_n\}$ be the sequence defined by

$$x_0 = 1, x_n = 5 \cdot x_{n-1}, \text{ for } n = 1, 2, \ldots.$$

Find a closed form for x_n. (Hint: Start by computing x_1, x_2, x_3, etc. until you see the pattern.)

⋆**Exercise 6.8.** Let $\{x_n\}$ be the sequence defined by

$$x_0 = 1, x_n = n \cdot x_{n-1}, \text{ for } n = 1, 2, \ldots.$$

Find a closed form for x_n.

⋆**Evaluate 6.9.** Define $\{a_n\}$ by $a(0) = 1$, $a(1) = 2$, and

$$a_n = \left\lfloor \frac{1+\sqrt{5}}{2} \times a_{n-1} \right\rfloor + a_{n-2}$$

for $n \geq 2$. Find a closed form for a_n.

Solution: We can see that

$$\begin{aligned}
a_2 &= \left\lfloor \tfrac{1+\sqrt{5}}{2} \times a_1 \right\rfloor + a_0 &= \left\lfloor \tfrac{1+\sqrt{5}}{2} \times 2 \right\rfloor + 1 &= 4 \\
a_3 &= \left\lfloor \tfrac{1+\sqrt{5}}{2} \times a_2 \right\rfloor + a_1 &= \left\lfloor \tfrac{1+\sqrt{5}}{2} \times 4 \right\rfloor + 2 &= 8 \\
a_4 &= \left\lfloor \tfrac{1+\sqrt{5}}{2} \times a_3 \right\rfloor + a_2 &= \left\lfloor \tfrac{1+\sqrt{5}}{2} \times 8 \right\rfloor + 4 &= 16
\end{aligned}$$

(You can verify these with a calculator). At this point it seems relatively clear that $a_n = 2^n$.

Evaluation ____

____

Did you catch what happened in the previous Evaluate exercise? The 'obvious' solution wasn't correct. If you missed this, go back and read the solution.

Generally speaking, you need to *prove* that the closed form is correct. One way to do this is to plug it back into the recursive definition. If we can plug it into the right hand side of the recursive definition and are able to simplify it to the left hand side, then it must be a solution. We also have to verify that it works for the initial condition(s).

As an analogy, how do you know that $x = -1$ is a solution to the equation $x^2 + 2x + 1 = 0$? You plug it in to get $(-1)^2 + 2(-1) + 1 = 1 - 2 + 1 = 0$. Since we got 0, $x = -1$ is a solution. We do something similar for recurrence relations, except that what we are plugging in is a formula instead of just a number.

Example 6.10. Prove that $x_n = n + 1$ is a solution to the recurrence relation given by

$$x_0 = 1, \quad x_n = \left(1 + \frac{1}{n}\right) x_{n-1}, \quad n = 1, 2, \ldots.$$

Proof: To prove that $x_n = n + 1$ is a solution for $n \geq 0$, we need to show two things. First, that it works for the initial condition. Since $x_0 = 1 = 0+1$, it works for the initial condition. Second, that if we plug it into the right hand side of the recursive definition, that we can simplify it to x_n. Doing so, we get

$$\begin{aligned}
\left(1 + \frac{1}{n}\right) x_{n-1} &= \left(1 + \frac{1}{n}\right)((n-1)+1) \\
&= \left(\frac{n+1}{n}\right) n \\
&= n + 1 \\
&= x_n
\end{aligned}$$

Since plugging the solution back in verifies the recurrence relation, $x_n = n + 1$ is a solution to the recurrence relation.

If you are confused by the first step of algebra, remember that we are assuming that $x_n = n + 1$ for $n \geq 0$. Thus, $x_{n-1} = (n - 1) + 1 = n$, since we are just plugging in $n - 1$ instead of n. □

⋆**Exercise 6.11.** Prove that your solution to Exercise 6.7 is correct.

⋆**Exercise 6.12.** Prove that your solution to Exercise 6.8 is correct.

⋆**Evaluate 6.13.** Determine what `ferzle(n)` (below) returns for $n = 0, 1, 2, 3, 4$ and then re-write `ferzle` without using recursion, making it as efficient as possible.[a]

```
int ferzle(int n) {
    if(n<=0) {
        return 3;
    } else {
        return ferzle(n-1) + 2;
    }
}
```

Solution: First, we can see that ferzle(0) returns 3 since it executes the code in the if statement. ferzle(1) returns ferzle(0)+2, which is $3+2=5$. ferzle(2) returns ferzle(1)+2, which is $5+2=7$. ferzle(3) returns ferzle(2)+2, which is $7+2=9$. ferzle(4) returns ferzle(3)+2, which is $9+2=11$. Notice that $11 = 2*4+3$, $9 = 2*3+3$, $7 = 2*2+3$, $5 = 2*1+3$, and $3 = 2*0+3$. From this, it is pretty clear that ferzle(n) returns $2n+3$. Thus, my simplified function is as follows:

```
int ferzle(int n) {
    return 2*n+3;
}
```

Evaluation ____

____

____

[a] Although we have not formally covered recursion yet, we expect that you have seen it before and know enough to follow this example.

⋆**Exercise 6.14.** Fix the code from the solution given in Evaluate 6.13 so that it still uses the closed form, but works correctly for all values of n.

```
int ferzle(int n) {



}
```

A more complete discussion of solving recurrences appears in Chapter 8.

The following is a famous example of a recursively defined sequence that we will revisit several times.

Example 6.15. The *Fibonacci sequence* is a sequence of numbers that is of interest in various mathematical and computing applications. They are defined using the following recurrence relation:[a]

$$f_n = \begin{cases} 0 & \text{if } n{=}0 \\ 1 & \text{if } n{=}1 \\ f_{n-1} + f_{n-2} & \text{if } n > 1 \end{cases}$$

In words, each Fibonacci number (beyond the first two) is the sum of the previous two. The first few are $f_0 = 0$, $f_1 = 1$,

$$\begin{aligned} f_2 &= f_1 + f_0 = 1 + 0 = 1, \\ f_3 &= f_2 + f_1 = 1 + 1 = 2, \\ f_4 &= f_3 + f_2 = 2 + 1 = 3, \\ f_5 &= f_4 + f_3 = 3 + 2 = 5, \\ f_6 &= f_5 + f_4 = 5 + 3 = 8, \\ f_7 &= f_6 + f_5 = 8 + 5 = 13. \end{aligned}$$

Later we will see the closed form for the Fibonacci sequence. If you are really adventurous, you might consider trying to determine it yourself. But be warned: It is not a simple formula that you will come up with by just looking at some of the Fibonacci numbers.

[a] In the remainder of the book, when you see f_k, you should assume it refers to the k-th Fibonacci number unless otherwise specified.

Definition 6.16. *A sequence* $\{a_n\}_{n=0}^{+\infty}$ *is said to be*

- **increasing** *if* $a_n \leq a_{n+1}$ $\forall n \in \mathbb{N}$
- **strictly increasing** *if* $a_n < a_{n+1}$ $\forall n \in \mathbb{N}$
- **decreasing** *if* $a_n \geq a_{n+1}$ $\forall n \in \mathbb{N}$
- **strictly decreasing** *if* $a_n > a_{n+1}$ $\forall n \in \mathbb{N}$

Some people call these sequences **non-decreasing**, **increasing**, **non-increasing**, *and* **decreasing**, *respectively.*

A sequence is called **monotonic** *if it is any of these, and* **non-monotonic** *if it is none of these.*

Example 6.17. Recall that $0! = 1$, $1! = 1$, $2! = 1 \cdot 2 = 2$, $3! = 1 \cdot 2 \cdot 3 = 6$, etc. Prove that the sequence $x_n = n!, n = 0, 1, 2, \ldots$ is strictly increasing for $n \geq 1$.

Proof: For $n > 1$ we have

$$x_n = n! = n(n-1)! = nx_{n-1} > x_{n-1},$$

since $n > 1$. This proves that the sequence is strictly increasing. $\square$

⋆**Question 6.18.** Notice in this first example we concluded that the sequence is strictly increasing since we showed that $x_n > x_{n-1}$. But according to the definition we need to show that $x_n < x_{n+1}$. So did we do something wrong? Explain.

Answer ____

____

____

Example 6.19. Prove that the sequence $x_n = 2 + \frac{1}{2^n}$, $n = 0, 1, 2, \ldots$ is strictly decreasing.

Proof: We have

$$\begin{aligned} x_{n+1} - x_n &= \left(2 + \frac{1}{2^{n+1}}\right) - \left(2 + \frac{1}{2^n}\right) \\ &= \frac{1}{2^{n+1}} - \frac{1}{2^n} \\ &= -\frac{1}{2^{n+1}} \\ &< 0. \end{aligned}$$

Thus, $x_{n+1} - x_n < 0$, so $x_n > x_n+1$, i.e., the sequence is strictly decreasing. □

⋆**Exercise 6.20.** Prove that the sequence $x_n = \frac{n^2+1}{n}$, $n = 1, 2, \ldots$ is strictly increasing.

⋆**Exercise 6.21.** Decide whether the following sequences are increasing, strictly increasing, decreasing, strictly decreasing, or non-monotonic. You do not need to prove your answer, but give a brief justification.

(a) $x_n = n$, $n = 0, 1, 2, \ldots$

Answer ______

______

(b) $x_n = (-1)^n n$, $n = 0, 1, 2, \ldots$

Answer ______

______

(c) $x_n = \dfrac{1}{n!}$, $n = 0, 1, 2, \ldots$

Answer ______

______

(d) $x_n = \dfrac{n}{n+1}$, $n = 0, 1, 2, \ldots$

Answer ______

______

(e) $x_n = n^2 - n$, $n = 1, 2, \ldots$

Answer ______

______

(f) $x_n = n^2 - n$, $n = 0, 1, 2, \ldots$

Answer ______

______

(g) $x_n = (-1)^n$, $n = 0, 1, 2, \ldots$

Answer ______

______

(h) $x_n = 1 - \frac{1}{2^n}$, $n = 0, 1, 2, \ldots$

Answer ____

(i) $x_n = 1 + \frac{1}{2^n}$, $n = 0, 1, 2, \ldots$

Answer ____

There are two types of sequences that come up often. We will briefly discuss each.

Definition 6.22. *A* **geometric progression** *is a sequence of the form*

$$a,\ ar, ar^2,\ ar^3,\ ar^4, \ldots,$$

where a (the **initial term***) and r (the* **common ratio***) are real numbers. That is, a geometric progression is a sequence in which every term is produced from the preceding one by multiplying it by a fixed number.*

Notice that the n-th term is ar^{n-1}. If $a = 0$ then every term is 0. If $ar \neq 0$, we can find r by dividing any term by the previous term.

Example 6.23. Find the 35-th term of the geometric progression

$$\frac{1}{\sqrt{2}},\ -2,\ \frac{8}{\sqrt{2}}, \ldots.$$

Solution: $a = \frac{1}{\sqrt{2}}$, and the common ratio is $r = -2/\frac{1}{\sqrt{2}} = -2\sqrt{2}$. Thus, the n-th term is $\frac{1}{\sqrt{2}}\left(-2\sqrt{2}\right)^{n-1}$. Hence the 35-th term is $\frac{1}{\sqrt{2}}\left(-2\sqrt{2}\right)^{34} = \frac{2^{51}}{\sqrt{2}} = 1125899906842624\sqrt{2}$.

⋆**Exercise 6.24.** Find the 17-th term of the geometric progression

$$-\frac{2}{3^{17}},\ \frac{2}{3^{16}},\ -\frac{2}{3^{15}}, \cdots.$$

Example 6.25. The fourth term of a geometric progression is 24 and its seventh term is 192. Find its second term.

Solution: We are given that $ar^3 = 24$ and $ar^6 = 192$, for some a and r. Clearly, $ar \neq 0$, and so we find

$$\frac{ar^6}{ar^3} = r^3 = \frac{192}{24} = 8.$$

Thus, $r = 2$. Now, $a(2)^3 = 24$, giving $a = 3$. The second term is thus $ar = 6$.

⋆**Exercise 6.26.** The 6-th term of a geometric progression is 20 and the 10-th is 320. Find the absolute value of its third term.

Definition 6.27. *An* **arithmetic progression** *is a sequence of the form*

$$a,\ a + d, a + 2d,\ a + 3d,\ a + 4d, \ldots,$$

where a (the **initial term***) and d (the* **common difference***) are real numbers. That is, an arithmetic progression is a sequence in which every term is produced from the preceding one by adding a fixed number.*

Example 6.28. If $s_n = 3n - 7$, then $\{s_n\}$ is an arithmetic progression with $a = -7$ and $d = 3$ (assuming we begin with s_0).

Note: *Notice that geometric progressions are essentially a discrete version of an exponential function and arithmetic progressions are a discrete version of a linear function. One consequence of this is that a sequence cannot be both of these unless it is the sequence $a, a, a, \ldots$ for some a.*

Example 6.29. Consider the sequence

$$4, 7, 10, 13, 16, 19, 22, \ldots.$$

Assuming the pattern continues, is this a geometric progression? Is it an arithmetic progression?

Solution: It is easy to see that each term is 3 more than the previous term. Thus, this is an arithmetic progression with $a = 4$ and $d = 3$. Clearly it is therefore not geometric.

⋆**Question 6.30.** Tests like the SAT and ACT often have questions such as the following.

23. Given the sequence of numbers 2, 9, 16, 23, what will the 8th term of the sequence be? (a) 60 (b) 58 (c) 49 (d) 51 (e) 56

(a) What is the 'correct' answer to this question?

Answer ______________________________

____

(b) Why did I put 'correct' in quotes in the previous question?

Answer ______________________________

____

⋆**Question 6.31.** Determine whether or not the following sequences are geometric and/or arithmetic. Explain your answer.

(a) The sequence from Example 6.7.

Answer ______________________________

____

(b) The sequence from Example 6.8.

Answer ______________________________

____

(c) The sequence generated by `ferzle(n)` in Evaluate 6.13 on the non-negative inputs.

Answer ______________________________

____

6.2 Sums and Products

When there is a need to add or multiply terms from a sequence, *summation notation* (or *sum notation*) and *product notation* come in handy. We first introduce sum notation.

Definition 6.32. *Let $\{a_n\}$ be a sequence. Then for $1 \le m \le n$, where m and n are integers, we define*

$$\sum_{k=m}^{n} a_k = a_m + a_{m+1} + \cdots + a_n.$$

We call k the **index of summation** *and m and n the* **limits** *of the summation. More specifically, m is the* **lower limit** *and n is the* **upper limit***. Each a_k is a* **term** *of the sum.*

Note: *We often use i, j, and k as index variables for sums, although any letters can be used.*

Example 6.33. We can express the sum $1 + 3 + 3^2 + 3^3 + \cdots + 3^{49}$ as

$$\sum_{i=0}^{49} 3^i.$$

(Recall that $3^0 = 1$, so the first term fits the pattern.)

⋆**Exercise 6.34.** Write the following sum using sum notation.

$$1 + y + y^2 + y^3 + \cdots + y^{100}$$

Example 6.35. Write the following sum using sum notation.

$$1 - y + y^2 - y^3 + y^4 - y^5 + \cdots - y^{99} + y^{100}$$

Solution: This is a lot like the previous exercise, except that every other term is negative. So how do we get those terms to be negative? The standard trick relies on the fact that $(-1)^i$ is 1 if i is even and -1 if i is odd. Thus, we can multiple each term by $(-1)^i$ for an appropriate choice of i. Since the odd powers are the negative ones, this is easy:

$$\sum_{i=0}^{100} (-1)^i y^i \qquad \text{or} \qquad \sum_{i=0}^{100} (-y)^i$$

Note: *You might be tempted to give the following solution to the previous problem:*

$$\sum_{i=0}^{100} -y^i.$$

As we will see shortly, this is the same as

$$-\sum_{i=0}^{100} y^i,$$

which is not the correct answer. The bottom line: Always use parentheses in the appropriate locations, especially when negative numbers are involved!

⋆**Exercise 6.36.** Write the following sum using sum notation.

$$1 + y^2 + y^4 + y^6 + \cdots + y^{100}$$

Note: *If you struggled understanding the two solutions to the previous example, it might be time to review the basic algebra rules involving exponents. We will just give a few of them here. You can find more extensive lists in an algebra book or various reputable online sources. We have already used the fact that if $x \neq 0$, then $x^0 = 1$. In addition, if $x, a, b \in \mathbb{R}$ with $x > 0$, then*

$$(x^a)^b = x^{ab}, \quad x^a x^b = x^{a+b}, \quad (x^{-a}) = \frac{1}{x^a}, \quad \textit{and} \quad x^{\frac{a}{b}} = \sqrt[b]{x^a} = \left(\sqrt[b]{x}\right)^a.$$

As with sequences, we are often interested in obtaining *closed forms* for sums. We will present several important formulas, along with a few techniques to find closed forms for sums.

Example 6.37. It should not be too difficult to see that

$$\sum_{k=1}^{20} 1 = 20$$

since this sum is adding 20 terms, each of which is 1. But notice that

$$\sum_{k=0}^{19} 1 = \sum_{k=200}^{219} 1 = 20$$

since both of these sums are also adding 20 terms, each of which is 1. In other words, if the variable of summation (the k) does not appear in the sum, then the only thing that matters is how many terms the sum involves.

⋆**Exercise 6.38.** Find each of the following.

(a) $\displaystyle\sum_{k=5}^{6} 1 =$ ____________________

(b) $\displaystyle\sum_{k=20}^{30} 1 =$ ____________________

(c) $\displaystyle\sum_{k=1}^{100} 1 =$ ____________________

(d) $\displaystyle\sum_{k=0}^{100} 1 =$ ____________________

Hopefully you noticed that the previous example and exercise can be generalized as follows.

Theorem 6.39. *If $a, b \in \mathbb{Z}$, then*

$$\sum_{k=a}^{b} 1 = (b - a + 1).$$

Proof: *This sum has $b-a+1$ terms since there are that many number between a and b, inclusive. Since each of the terms is 1, the sum is obviously $b-a+1$.* □

Example 6.40. If we apply the previous theorem to the sums in Example 6.37, we would obtain $20 - 1 + 1 = 20$, $19 - 0 + 1 = 20$, and $219 - 200 + 1 = 20$.

Next is a simple theorem based on the distributive law that you learned in grade school.

Theorem 6.41. *If $\{x_n\}$ is a sequence and a is a real number, then*

$$\sum_{k=m}^{n} a \cdot x_k = a \sum_{k=m}^{n} x_k.$$

Example 6.42. Using Theorems 6.39 and 6.41, we can see that

$$\sum_{k=5}^{17} 4 = 4 \sum_{k=5}^{17} 1 = 4 \cdot (17 - 5 + 1) = 4 \cdot 13 = 52.$$

⋆**Exercise 6.43.** Find each of the following.

(a) $\displaystyle\sum_{k=5}^{6} 5 =$ ____

(b) $\displaystyle\sum_{k=20}^{30} 200 =$ ____

We can combine Theorems 6.39 and 6.41 to obtain the following.

Theorem 6.44. *If $a, b \in \mathbb{Z}$ and $c \in \mathbb{R}$, then*

$$\sum_{k=a}^{b} c = (b - a + 1)c.$$

Proof: *Using Theorem 6.41, we have*

$$\sum_{k=a}^{b} c = c \sum_{k=a}^{b} 1 = (b - a + 1)c.$$

□

Example 6.45. We can compute the sum from Example 6.42 by using Theorem 6.44 to obtain

$$\sum_{k=5}^{17} 4 = (17 - 5 + 1)4 = 52.$$

Both ways of computing this sum are valid, so feel free to use whichever you prefer.

⋆**Exercise 6.46.** Find each of the following.

(a) $\displaystyle\sum_{k=5}^{6} 5 =$ ______________________

(b) $\displaystyle\sum_{k=20}^{30} 200 =$ ______________________

(c) $\displaystyle\sum_{k=1}^{100} 9 =$ ______________________

(d) $\displaystyle\sum_{k=0}^{100} 9 =$ ______________________

⋆**Evaluate 6.47.** Compute $\sum_{k=25}^{75} 10$.

Solution: This is just $10(75-25) = 10 * 50 = 500$.

Evaluation ____________________

____

The following sum comes up often and should be committed to memory. The proof involves a nice technique that adds the terms in the sum twice, in a different order, and then divides the result by two. This is known as Gauss' trick.

Theorem 6.48. *If n is a positive integer, then*

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}.$$

Proof: *Let $S = \sum_{k=1}^{n} k$ for shorthand. Then we can see that*

$$S = 1 + 2 + 3 + \cdots + n$$

and by reordering the terms,

$$S = n + (n-1) + \cdots + 1.$$

Adding these two quantities,

$$\begin{array}{rcccccccc} S & = & 1 & + & 2 & + & \cdots & + & n \\ S & = & n & + & (n-1) & + & \cdots & + & 1 \\ \hline 2S & = & (n+1) & + & (n+1) & + & \cdots & + & (n+1) \\ & = & n(n+1), & & & & & & \end{array}$$

since there are n terms. Dividing by 2, we obtain $S = \frac{n(n+1)}{2}$, as was to be proved. □

Example 6.49.

$$\sum_{k=1}^{10} k = \frac{10(10+1)}{2} = \frac{10 \cdot 11}{2} = 55.$$

⋆**Exercise 6.50.** Compute each of the following.

(a) $\sum_{k=1}^{20} k =$ ______________________

(b) $\sum_{k=1}^{100} k =$ ______________________

(c) $\sum_{k=1}^{1000} k =$ ______________________

⋆**Evaluate 6.51.** Compute the following.

$$\sum_{k=1}^{30} k.$$

Solution 1: $\sum_{k=1}^{30} k = 29 * 30/2 = 435.$

Evaluation ______________________

Solution 2: $\sum_{k=1}^{30} k = k \sum_{k=1}^{30} 1 = k(30 - 1 + 1) = 30k.$

Evaluation ______________________

Note: *A common error is to think that the sum of the first n integers is $n(n-1)/2$ instead of $n(n+1)/2$. Whenever I use the formula, I double check my memory by computing $1+2+3$. In this case, $n = 3$. So is the correct answer $3 \cdot 2/2 = 3$ or $3 \cdot 4/2 = 6$? Clearly it is the latter. Then I know that the correct formula is $n(n+1)/2$. You can use any positive value of n to check the formula. I use 3 out of habit.*

⋆**Question 6.52.** Is it true that $\sum_{k=0}^{n} k = \sum_{k=1}^{n} k = \frac{n(n+1)}{2}$? Explain.

Answer ______________________

Theorem 6.53. *If $\{x_k\}$ and $\{y_k\}$ are sequences, then for any $n \in \mathbb{Z}^+$,*

$$\sum_{i=1}^{n} x_i + y_i = \sum_{i=1}^{n} x_i + \sum_{i=1}^{n} y_i.$$

Proof: *This follows from the commutative property of addition.* □

Example 6.54.

$$\sum_{i=1}^{20} i + 5 = \sum_{i=1}^{20} i + \sum_{i=1}^{20} 5 = \frac{20 \cdot 21}{2} + 5 \cdot 20 = 210 + 100 = 310.$$

⋆**Exercise 6.55.** Compute the following sum

$\sum_{i=1}^{100} 2 - i =$ ____.

Example 6.56. Let $\{a_k\}$ be a sequence of real numbers. Show that $\sum_{i=1}^{n}(a_i - a_{i-1}) = a_n - a_0$.

Proof: We can see that

$$\begin{aligned}
\sum_{i=1}^{n}(a_i - a_{i-1}) &= \left(\sum_{i=1}^{n} a_i\right) - \left(\sum_{i=1}^{n} a_{i-1}\right) \\
&= (a_1 + a_2 + \cdots + a_{n-1} + a_n) - (a_0 + a_1 + a_2 + \cdots + a_{n-1}) \\
&= a_1 + a_2 + \cdots + a_{n-1} + a_n - a_0 - a_1 - a_2 - \cdots - a_{n-1} \\
&= (a_1 - a_1) + (a_2 - a_2) + \cdots + (a_{n-1} - a_{n-1}) + a_n - a_0 \\
&= a_n - a_0.
\end{aligned}$$

□

⋆**Exercise 6.57.** Prove that the sum of the first n odd integers is n^2.

Example 6.58. Given what we know so far, how can we compute the following:

$$\sum_{k=50}^{100} k = ?$$

It turns out that this is not that hard. Notice that it is *almost* a sum we know. We know how to compute $\sum_{k=1}^{100} k$, but that has too many terms. Can we just subtract those terms to get the answer? What terms don't we want? Well, we don't want terms 1 through 49. But that is just $\sum_{k=1}^{49} k$. In other words,

$$\begin{aligned}\sum_{k=50}^{100} k &= \sum_{k=1}^{100} k - \sum_{k=1}^{49} k \\ &= \frac{100 \cdot 101}{2} - \frac{49 \cdot 50}{2} \\ &= 5050 - 1225 = 3825\end{aligned}$$

⋆**Exercise 6.59.** Compute each of the following.

(a) $\sum_{k=10}^{20} k =$ ______________________

(b) $\sum_{k=21}^{40} k =$ ______________________

⋆**Evaluate 6.60.** Compute the following.

$$\sum_{k=30}^{100} k.$$

Solution 1:

$$\sum_{k=30}^{100} k = \sum_{k=1}^{100} k - \sum_{k=1}^{30} k = 100 \cdot 101/2 - 30 \cdot 31/2 = 5050 - 465 = 4585$$

Evaluation ______________________

____

Solution 2:

$$\sum_{k=30}^{100} k = \sum_{k=1}^{100} k - \sum_{k=1}^{30} k = 99 \cdot 100/2 - 29 \cdot 30/2 = 4950 - 435 = 4515$$

Evaluation ______________________________

Solution 3:

$$\sum_{k=30}^{100} k = \sum_{k=1}^{100} k - \sum_{k=1}^{29} k = 100 \cdot 101/2 - 29 \cdot 30/2 = 5050 - 435 = 4615$$

Evaluation ______________________________

⋆**Question 6.61.** Explain why the following computation is incorrect. Then explain why the answer is correct even with the error(s).

$$\sum_{k=30}^{100} k = \sum_{k=1}^{100} k - \sum_{k=1}^{30} k = 100 \cdot 101/2 - 29 \cdot 30/2 = 5050 - 435 = 4615$$

Answer ______________________________

Theorem 6.62. *Let $n \in \mathbb{Z}^+$. Then the following hold.*

$$\begin{aligned}\sum_{k=1}^{n} k^2 &= \frac{n(n+1)(2n+1)}{6}\\ \sum_{k=1}^{n} k^3 &= \frac{n^2(n+1)^2}{4}\\ \sum_{k=2}^{n} \frac{1}{(k-1)k} &= \frac{1}{1\cdot 2} + \frac{1}{2\cdot 3} + \frac{1}{3\cdot 4} + \cdots + \frac{1}{(n-1)\cdot n} = \frac{n-1}{n}\end{aligned}$$

We will prove Theorem 6.62 in the chapter on mathematical induction since that is perhaps the easiest way to prove them. It is probably a good idea to attempt to commit the first two of these sums to memory since they come up on occasion.

⋆**Question 6.63.** Why does the third formula from Theorem 6.62 have a lower index of 2 (instead of 1 or 0, for instance)?

Answer ______________________________

____

⋆**Exercise 6.64.** Compute the following sum, simplifying as much as possible.

$$\sum_{k=1}^{n} k^3 + k =$$

Sometimes double sums are necessary to express a summation. As a general rule, these should be evaluated from the inside out.

Example 6.65. Evaluate the double sum $\displaystyle\sum_{i=1}^{n}\sum_{j=1}^{n} 1$.

Solution: We have $\displaystyle\sum_{i=1}^{n}\sum_{j=1}^{n} 1 = \sum_{i=1}^{n} n = n \cdot n = n^2$.

⋆**Exercise 6.66.** Evaluate the following double sums

(a) $\displaystyle\sum_{i=1}^{n}\sum_{j=1}^{i} 1 =$

(b) $\displaystyle\sum_{i=1}^{n}\sum_{j=1}^{i} j =$

(c) $\displaystyle\sum_{i=1}^{n}\sum_{j=1}^{n} ij =$

There is a formula for the sum of a geometric sequence, sometimes referred to as a *geometric series*. It is given in the next theorem.

Theorem 6.67. *Let $x \neq 1$. Then*

$$\sum_{k=0}^{n} x^k = \frac{1-x^{n+1}}{1-x} \qquad \left(\textit{or} \quad \frac{x^{n+1}-1}{x-1} \quad \textit{if you prefer}\right).$$

Proof: *First, let* $S = \sum_{k=0}^{n} x^k$. *Then*

$$xS = x\sum_{k=0}^{n} x^k = \sum_{k=0}^{n} x^{k+1} = \sum_{k=1}^{n+1} x^k.$$

So

$$\begin{aligned} xS - S &= \sum_{k=1}^{n+1} x^k - \sum_{k=0}^{n} x^k \\ &= (x_1 + x_2 + \ldots + x_n + x_{n+1}) - (x_0 + x_1 + \ldots + x_n) \\ &= x^{n+1} - x^0 = x^{n+1} - 1. \end{aligned}$$

So we have $(x-1)S = x^{n+1} - 1$, *so* $S = \frac{x^{n+1}-1}{x-1}$, *since* $x \neq 1$. □

Example 6.68.

$$\sum_{k=0}^{n} 3^k = \frac{1-3^{n+1}}{1-3} = \frac{1-3^{n+1}}{-2} = \frac{3^{n+1}-1}{2}.$$

⋆**Exercise 6.69.** Find the sum of the following geometric series. For (b)-(d), assume $y \neq 1$.

(a) $1 + 3 + 3^2 + 3^3 + \cdots + 3^{49} =$

(b) $1 + y + y^2 + y^3 + \cdots + y^{100} =$

(c) $1 - y + y^2 - y^3 + y^4 - y^5 + \cdots - y^{99} + y^{100} =$

(d) $1 + y^2 + y^4 + y^6 + \cdots + y^{100} =$

Corollary 6.70. *Let $N \geq 2$ be an integer. Then*

$$x^N - 1 = (x-1)(x^{N-1} + x^{N-2} + \cdots + x + 1).$$

Proof: *Plugging $N = n+1$ in the formula from Theorem 6.67 and doing a little algebra yields the formula.* $\square$

Example 6.71. We can see that

$$\begin{aligned} x^2 - 1 &= (x-1)(x+1) \\ x^3 - 1 &= (x-1)(x^2+x+1), \text{ and} \\ x^4 - 1 &= (x-1)(x^3+x^2+x+1). \end{aligned}$$

⋆**Exercise 6.72.** Factor $x^5 - 1$.

$x^5 - 1 =$ ________________________________

Let's use the technique from the proof of Theorem 6.67 in the special case where $x = 2$.

⋆**Fill in the details 6.73.** Find the sum

$$2^0 + 2^1 + 2^2 + 2^3 + 2^4 + \cdots + 2^n.$$

Solution: We could just use the formula from Theorem 6.67, but that would be boring. Instead, let's work it out. Let

$$S = 2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2^n.$$

Then $2S =$ ________________________. Notice S and $2S$ have most of the same terms, except S has ____________ that $2S$ doesn't have and $2S$ has

______________ that S doesn't have. Therefore,

$$\begin{array}{rcl} S = 2S - S & = & \begin{array}{ccccccccccccc} & & (2^1 & + & 2^2 & + & 2^3 & + & \cdots & + & 2^n & + & 2^{n+1}) \\ -(2^0 & + & 2^1 & + & 2^2 & + & 2^3 & + & \cdots & + & 2^n) & & \end{array} \\ & & \\ & = & \rule{4cm}{0.4pt} \\ & = & 2^{n+1} - 1. \end{array}$$

Thus,

$$\sum_{k=0}^{n} 2^k = 2^{n+1} - 1.$$

Since powers of 2 are very prominent in computer science, you should definitely commit the formula from the previous example to memory.

Together, Theorems 6.41 and 6.67 imply the following:

Theorem 6.74. *Let $r \neq 1$. Then*

$$\sum_{k=0}^{n} ar^k = \frac{a - ar^{n+1}}{1 - r}.$$

⋆**Fill in the details 6.75.** Use Theorems 6.41 and 6.67 to prove Theorem 6.74.

Proof: It is easy to see that

$$\begin{aligned} \sum_{k=0}^{n} ar^k &= \\ \\ &= \\ \\ &= \frac{a - ar^{n+1}}{1 - r}. \end{aligned}$$

□

⋆**Exercise 6.76.** Prove Theorem 6.74 *without using Theorems 6.41 and 6.67.* In other words, mimic the proof of Theorem 6.67.

Notice that if $|r| < 1$ then r^n gets closer to 0 the larger n gets. More formally, if $|r| < 1$, $\lim_{n\to\infty} r^n = 0$. This implies the following (which we will not formally prove beyond what we have already said here).

Theorem 6.77. *Let* $|r| < 1$. *Then*

$$\sum_{k=0}^{\infty} ar^k = \frac{a}{1-r}.$$

Example 6.78. A fly starts at the origin and goes 1 unit up, 1/2 unit right, 1/4 unit down, 1/8 unit left, 1/16 unit up, etc., *ad infinitum.* In what coordinates does it end up?

Solution: Its x coordinate is

$$\frac{1}{2}-\frac{1}{8}+\frac{1}{32}-\cdots=\frac{1}{2}\left(-\frac{1}{4}\right)^0+\frac{1}{2}\left(-\frac{1}{4}\right)^1+\frac{1}{2}\left(-\frac{1}{4}\right)^2+\cdots=\frac{\frac{1}{2}}{1-\frac{-1}{4}}=\frac{2}{5}.$$

Its y coordinate is

$$1-\frac{1}{4}+\frac{1}{16}-\cdots=\left(-\frac{1}{4}\right)^0+\left(-\frac{1}{4}\right)^1+\left(-\frac{1}{4}\right)^2+\cdots=\frac{1}{1-\frac{-1}{4}}=\frac{4}{5}.$$

Therefore, the fly ends up in $\left(\frac{2}{5},\frac{4}{5}\right)$.

The following infinite sums are sometimes useful.

Theorem 6.79. *Let $x \in \mathbb{R}$. The following expansions hold:*

$$\begin{aligned}
\sin x &= \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!} &&= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots + (-1)^n \frac{x^{2n+1}}{(2n+1)!} + \cdots \\
\cos x &= \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!} &&= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \cdots + (-1)^n \frac{x^{2n}}{(2n)!} + \cdots \\
e^x &= \sum_{n=0}^{\infty} \frac{x^n}{n!} &&= 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!} + \cdots \\
\frac{1}{1-x} &= \sum_{n=0}^{\infty} x^n &&= 1 + x + x^2 + x^3 + \cdots, \ \textit{if } |x| < 1
\end{aligned}$$

Product notation is very similar to sum notation, except we multiply the terms instead of adding them.

Definition 6.80. *Let $\{a_n\}$ be a sequence. Then for $1 \leq m \leq n$, where m and n are integers, we define*

$$\prod_{k=m}^{n} a_k = a_m a_{m+1} \cdots a_n.$$

As with sums, we call k the **index** *and m and n the* **lower limit** *and* **upper limit**, *respectively.*

Example 6.81. Notice that $n! = \prod_{k=1}^{n} k$.

Note: *An alternative way to express the variable and limits of sums and products is*

$$\sum_{m \leq k \leq n} a_k \qquad \textit{and} \qquad \prod_{m \leq k \leq n} a_k$$

6.3 Problems

Problem 6.1. Find at least three *different* sequences that begin with 1, 3, 7 whose terms are generated by a simple formula or rule. By different, I mean none of the sequences can have exactly the same terms. In other words, your answer cannot simply be three different ways to generate the same sequence.

Problem 6.2. Let $q_n = 2q_{n-1} + 2n + 5$, and $q_0 = 0$. Compute q_1, q_2, q_3 and q_4.

Problem 6.3. Let $a_n = a_{n-2} + n$, $a_0 = 0$, and $a_1 = 1$. Compute a_2, a_3, a_4 and a_5.

Problem 6.4. Let $a_n = n \times a_{n-1} + 5$, and $a_0 = 1$. Compute a_1, a_2, a_3, a_4 and a_5.

Problem 6.5. Define a sequence $\{x_n\}$ by $x_0 = 1$, and $x_n = 2x_{n-1} + 1$ if $n \geq 1$. Find a closed form for the nth term of this sequence. *Prove that your solution is correct.*

Problem 6.6. Compute each of the following:

(a) $\displaystyle\sum_{k=5}^{40} k$

(b) $\displaystyle\sum_{j=5}^{22} (2^{j+1} - 2^j)$

(c) $\displaystyle\sum_{k=0}^{n} 5k$

(d) $\displaystyle\sum_{i=1}^{3}\sum_{j=1}^{4} j$

(e) $\displaystyle\sum_{k=1}^{n} k(k-1)$

(f) $\displaystyle\sum_{j=1}^{n} 5^j$

(g) $\displaystyle\sum_{j=0}^{\log_2 n} 2^j$

(h) $\displaystyle\sum_{i=0}^{\log_2 n} \left(\frac{n}{2^i}\right)$

(i) $\displaystyle\sum_{i=1}^{n}\sum_{j=1}^{i}\sum_{k=1}^{j} 1$

Problem 6.7. Here is a standard interview question for prospective computer programmers: You are given a list of $1,000,001$ positive integers from the set $\{1, 2, \ldots, 1,000,000\}$. In your list, every member of $\{1, 2, \ldots, 1,000,000\}$ is listed once, except for x, which is listed twice. How do you find what x is without doing a $1,000,000$ step search?

Problem 6.8. Find a closed formula for

$$T_n = 1^2 - 2^2 + 3^2 - 4^2 + \cdots + (-1)^{n-1}n^2.$$

Problem 6.9. Show that when $n \geq 1$,

$$1 + 3 + 5 + \cdots + (2n-1) = n^2.$$

Problem 6.10. Show that when $n \geq 1$,

$$\sum_{k=1}^{n} \frac{k}{k^4 + k^2 + 1} = \frac{1}{2} \cdot \frac{n^2 + n}{n^2 + n + 1}.$$

Problem 6.11. Legend says that the inventor of the game of chess, Sissa ben Dahir, asked the King Shirham of India to place a grain of wheat on the first square of the chessboard, 2 on the second square, 4 on the third square, 8 on the fourth square, etc..

(a) How many grains of wheat are to be put on the last (64-th) square?

(b) How many grains, total, are needed in order to satisfy the greedy inventor?

(c) Given that 15 grains of wheat weigh approximately one gram, what is the approximate weight, in kg, of the wheat needed?

(d) Given that the annual production of wheat is 350 million tonnes, how many years, approximately, are needed in order to satisfy the inventor (assume that production of wheat stays constant)?

Problem 6.12. It is easy to see that we can define $n!$ recursively by defining $0! = 1$, and if $n > 0$, $n! = n \cdot (n-1)!$. Does the following method correctly compute $n!$? If not, state what is wrong with it and fix it.

```
int factorial(int n) {
    return n * factorial(n-1);
}
```

Problem 6.13. Obtain a closed formula for $\sum_{k=1}^{n} k \cdot k!$. (Hint: What is $(k+1)! - k!$, and why does it matter?)

Problem 6.14. A student turned in the code below (which does as its name suggests). I gave them a 'C' on the assignment because although it works, it is very inefficient.

```
int sumFromOneToN(int n) {
    int sum = 0;
    for(int i=1;i<=n;i++) {
        sum = sum + i;
    }
return sum;
}
```

(a) Write the 'A' version of the algorithm (in other words, a more efficient version). You can assume that $n \geq 1$.

(b) Compute `sumFromOneToN(30)` based on your algorithm.

Problem 6.15. A student turned in the code below (which does as its name suggests). I gave them a 'C' on the assignment because although it works, it is very inefficient.

```
int sumFromMToN(int m, int n) {
    int sum = 0;
    for(int i=1;i<=n;i++) {
        sum = sum + i;
    }
    for(int i=1;i<m;i++) {
        sum = sum - i;
    }
    return sum;
}
```

(a) Write the 'A' version of the algorithm (in other words, a more efficient version). You can assume that $1 \leq m \leq n$.

(b) Compute `sumFromMToN(10,50)` based on your algorithm.

Chapter 7

Algorithm Analysis

In this chapter we take a look at the analysis of algorithms. The analysis of algorithms is a very important topic in computer science. It allows us to determine and express how efficient an algorithm is, and it is one of the tools that allows us to compare multiple algorithms that solve the same problem.

Before we dive into that topic, we first discuss one of the most important tools used in algorithm analysis—*asymptotic notation*. We will define several important notations, discuss some of the useful properties of the notations, and provide many examples of two common ways of proving things related to the notations. We will then discuss the relative growth rates of several common functions, focusing on those that are relevant to the topic of algorithm analysis. We then move on to the most important topic of the chapter in which we apply all of this material to the analysis of algorithms, providing numerous examples of determining the computational complexity of various algorithms. Finally, we discuss some of the most common time complexities that occur in the study of algorithms.

7.1 Asymptotic Notation

Asymptotic notation is used to express and compare the growth rate of functions. In our case, the functions will typically represent the running time of algorithms. We will define the asymptotic notations in terms of nonnegative functions. You will find more general definitions of these notations in other books, but they are more complicated, more difficult to understand, and harder to work with. These added difficulties are a result of the possibility of the functions involved being negative. But the main reason for our use of the notations is to express the running time of algorithms. Since the running time of an algorithm is always nonnegative, there is really no good reason to use the more cumbersome definitions. We will focus on the notations most commonly used in the analysis of algorithms.

Asymptotic notation allows us to express the behavior of a function as the input approaches infinity. In other words, it is concerned about what happens to $f(n)$ as n gets larger, and is not concerned about the value of $f(n)$ for small values of n.

We will define four of the most commonly used notations (an allude to the definition of a fifth), providing a few brief examples of each. We will then discuss some of the most important and useful properties of these notations. Finally, we will present many more detailed examples.

7.1.1 The Notations

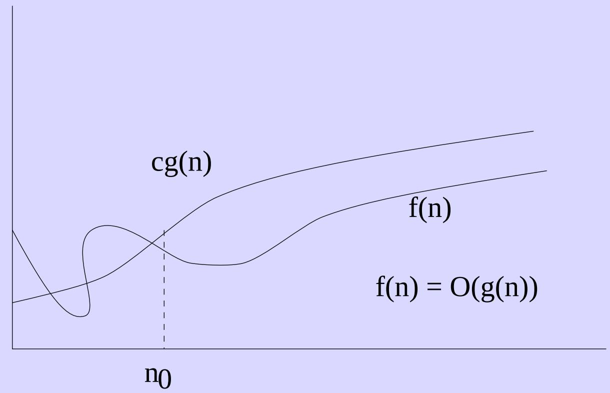
We begin with the most commonly used of the notations.

Definition 7.1 (Big-O)**.** *Let f be a nonnegative function.*

We say that $f(n)$ is **Big-O** *of $g(n)$, written as $f(n) = O(g(n))$, iff there are positive constants c and n_0 such that*

$$f(n) \le c\,g(n) \text{ for all } n \ge n_0.$$

If $f(n) = O(g(n))$, $f(n)$ grows no faster than $g(n)$. In other words, $g(n)$ is an **asymptotic upper bound** *(or just* **upper bound***) on $f(n)$.*



Note: *The "=" in the statement "$f(n) = O(g(n))$" should be read and thought of as "is", not "equals." You can think of it as a one-way equals. So saying $f(n) = O(g(n))$ is* not *the same thing as saying $O(g(n)) = f(n)$, for instance (with the latter statement not really making sense).*

An alternative notation is to write $f(n) \in O(g(n))$ instead of $f(n) = O(g(n))$. It turns out that $O(g(n))$ is actually the set of all functions that grow no faster than $g(n)$, so the set notation is actually in some sense more correct. The "=" notation is used because it comes in handy when doing algebra. You can essentially think of these as being two different notations ($=$ and $\in$) for the same thing. Similar statements are true for the other asymptotic notations.

Example 7.2. Prove that $n^2 + n = O(n^3)$.

Solution: Here, we have $f(n) = n^2 + n$, and $g(n) = n^3$. Notice that if $n \ge 1$, $n \le n^3$ and $n^2 \le n^3$. Therefore,

$$n^2 + n \le n^3 + n^3 = 2n^3$$

Thus,

$$n^2 + n \le 2n^3 \text{ for all } n \ge 1.$$

Thus, we have shown that $n^2 + n = O(n^3)$ by definition of Big-O, with $n_0 = 1$, and $c = 2$.

The following fact is a generalization of what was used in the previous example. It is used often in proofs involving asymptotic notation.

Theorem 7.3. *If a and b are real numbers with $a \le b$, then $n^a \le n^b$ whenever $n \ge 1$.*

Proof: *We will not provide a proof, but it should be fairly clear intuitively that this is true. If you cannot see why this is true, you should work out a few examples to convince yourself.* $\square$

Sometimes the easiest way to prove that $f(n) = O(g(n))$ is to take c to be the sum of the *positive* coefficients of $f(n)$, although this trick doesn't always work. We can usually easily eliminate the lower order terms with negative coefficients if we make the appropriate assumption. Let's see how to do this in the next few examples.

Example 7.4. Prove that $3n^3 - 2n^2 + 13n - 15 = O(n^3)$.

Solution: First, notice that if $n \geq 0$, then $-2n^2 - 15 \leq 0$, so

$$3n^3 - 2n^2 + 13n - 15 \leq 3n^3 + 13n.$$

Next, if $n \geq 1$, then $13n \leq 13n^3$. Therefore if $n \geq 1$,

$$3n^3 + 13n \leq 3n^3 + 13n^3 = 16n^3.$$

Also notice that if $n \geq 1$, then $n \geq 0$. Thus, our first step is still valid if we assume $n \geq 1$ since $n \geq 1$ is a stronger condition than $n \geq 0$. Putting this all together, if we assume $n \geq 1$, then

$$\begin{aligned} 3n^3 - 2n^2 + 13n - 15 &\leq 3n^3 + 13n \\ &\leq 3n^3 + 13n^3 \\ &= 16n^3. \end{aligned}$$

Since we have shown that $3n^3 - 2n^2 + 13n - 15 \leq 16n^3$ for all $n \geq 1$, we have proven that $3n^3 - 2n^2 + 13n - 15 = O(n^3)$.

We used $n_0 = 1$ and $c = 16$ in our proof. It is not necessary to explicitly point this out in our proof, though. We only do so to help you see the connection between the proof and the definition of Big-O.

Example 7.5. Prove that $5n^2 - 3n + 20 = O(n^2)$.

Solution: If $n \geq 1$,

$$\begin{aligned} 5n^2 - 3n + 20 &\leq 5n^2 + 20 && (7.1) \\ &\leq 5n^2 + 20n^2 && (7.2) \\ &= 25n^2. && (7.3) \end{aligned}$$

Since $5n^2 - 3n + 20 \leq 25n^2$ for all $n \geq 1$, $5n^2 - 3n + 20 = O(n^2)$.

In this proof we used $c = 25$ and $n_0 = 1$.

⋆**Question 7.6.** Answer the following questions related to Example 7.5.

(a) What allowed us to eliminate the $-3n$ term in step 7.1? ______________________

____

(b) What is the justification for step 7.2? ____________________

____

⋆**Evaluate 7.7.** Prove that $4n^2 - 12n + 10 = O(n^2)$.

Solution: If $n \geq 1$, $4n^2 - 12n + 10 \leq 4n^2 - 12n^2 + 10n^2 = 2n^2$. Therefore, $4n^2 - 12n + 10 = O(n^2)$.

Evaluation ____________________

____

Note: *The values of the constants used in the proofs do not need to be the best possible. For instance, if you can show that $f(n) \leq 345\,g(n)$ for all $n \geq 712$, then $f(n) = O(g(n))$. It doesn't matter whether or not it is actually true that $f(n) \leq 3\,g(n)$ for all $n \geq 5$.*

⋆**Question 7.8.** Answer each of the following questions related to Example 7.5. Include a brief justification.

(a) Could we have used $c = 50$ in the proof?

Answer ____________________

____

(b) Could we have used $c = 2$ in the proof?

Answer ____________________

____

(c) Could we have used $n_0 = 100$ in the proof?

Answer ____________________

____

(d) Could we have used $n_0 = 0$ in the proof?

Answer ____________________

____

⋆**Exercise 7.9.** Prove that $5n^5 - 4n^4 + 3n^3 - 2n^2 + n = O(n^5)$. (Hint: Use the same techniques you saw in Example 7.5.)

⋆**Question 7.10.** What values did you use for n_0 and c in your solution to Exercise 7.9?

$n_0 =$ __________, $c =$ __________

Things are not always so easy. How would you show that $(\sqrt{2})^{\log n} + \log^2 n + n^4 = O(2^n)$? Or that $n^2 = O(n^2 - 13n + 23)$? In general, we simply (or in some cases with much effort) find values c and n_0 that work. This gets easier with practice.

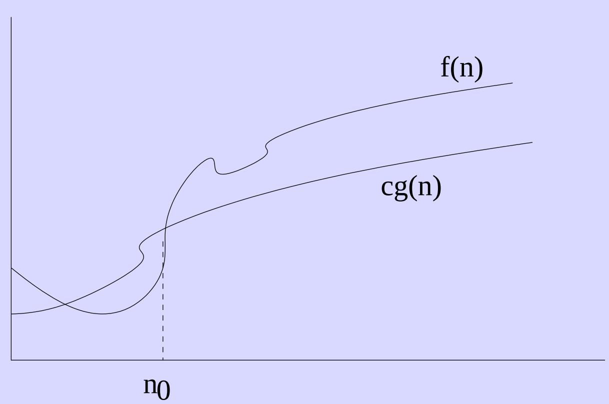
Big-O is a notation to express the idea that one function is an upper bound for another function. The next notation allows us to express the opposite idea—that one function is a *lower bound* for another function.

Definition 7.11 (Big-Omega)**.** *Let f and g be nonnegative functions.*

We say that $f(n)$ is **Big-Omega** *of $g(n)$, written as $f(n) = \Omega(g(n))$, iff there are positive constants c and n_0 such that*

$$c\,g(n) \leq f(n) \text{ for all } n \geq n_0$$

If $f(n) = \Omega(g(n))$, $f(n)$ grows no slower than $g(n)$. In other words, $g(n)$ is an **asymptotic lower bound** *(or just* **lower bound***) on $f(n)$.*



Example 7.12. Prove that $n^3 + 4\,n^2 = \Omega(n^2)$.

Proof: Here, we have $f(n) = n^3 + 4\,n^2$, and $g(n) = n^2$. It is not too hard to see that if $n \geq 1$,

$$n^2 \leq n^3 \leq n^3 + 4\,n^2$$

Therefore,

$$1\,n^2 \leq n^3 + 4\,n^2 \text{ for all } n \geq 1$$

so $n^3 + 4\,n^2 = \Omega(n^2)$ by definition of Ω, with $n_0 = 1$, and $c = 1$. □

⋆**Exercise 7.13.** Prove that $4n^2 + n + 1 = \Omega(n^2)$. (This one should be really easy—follow the technique from the previous example and don't over think it.)

⋆**Question 7.14.** What values did you use for n_0 and c in your solution to Exercise 7.13?

$n_0 =$ __________, $c =$ __________

Proving that $f(n) = \Omega(g(n))$ often requires more thought than proving that $f(n) = O(g(n))$. Although the lower-order terms with positive coefficients can be easily dealt with, those with negative coefficients make things a bit more complicated. Often, we have to pick $c < 1$. A good strategy is to pick a value of c that you think will work, and determine which value of n_0 is needed. Being able to do some algebra helps. As it turns out, we won't have to worry a whole lot about this, though. We will see a different technique to prove bounds shortly that, when it works, makes things much easier.

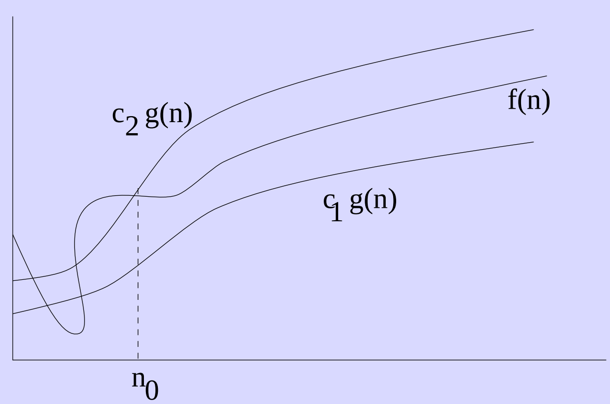
Our third notation allows us to express the idea that two functions grow at the same rate.

Definition 7.15 (Big-Theta)**.** *Let f and g be nonnegative functions.*

We say that $f(n)$ is **Big-Theta** *of $g(n)$, written as $f(n) = \Theta(g(n))$, iff there are positive constants c_1, c_2 and n_0 such that*

$$c_1\,g(n) \leq f(n) \leq c_2\,g(n) \text{ for all } n \geq n_0$$

If $f(n) = \Theta(g(n))$, $f(n)$ grows at the same rate as $g(n)$. In other words, $g(n)$ is an **asymptotically tight bound** *(or just* **tight bound***) on $f(n)$.*



Example 7.16. Prove that $n^2 + 5n + 7 = \Theta(n^2)$

Proof: When $n \geq 1$, $n^2 + 5\,n + 7 \leq n^2 + 5\,n^2 + 7n^2 \leq 13n^2$.
When $n \geq 0$, $n^2 \leq n^2 + 5\,n + 7$.
Combining these, we can see that when $n \geq 1$,

$$n^2 \leq n^2 + 5\,n + 7 \leq 13\,n^2,$$

so $n^2{+}5\,n{+}7 = \Theta(n^2)$ by definition of Θ, with $n_0{=}1$, $c_1{=}1$, and $c_2{=}13$. □

⋆**Question 7.17.** In the previous example, we combined two inequalities. One of them assumed $n \geq 0$, the other assumed that $n \geq 1$. In the combined inequality, we said it held if $n \geq 1$. Is that really O.K., or did we make a subtle error?

Answer ____

____

Using the definition of Θ can be inconvenient since it involves a double inequality. Luckily, the following theorem provides us with an easier approach.

Theorem 7.18. *If f and g are nonnegative functions, then $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.*

Proof: *The result follows almost immediately from the definitions. We leave the details to the reader.* □

This theorem implies that no new strategies are necessary for Θ proofs since they can be split into two proofs—a Big-O proof and a Ω proof. Let's see an example of this approach.

Example 7.19. Show that $\frac{1}{2}n^2 + 3\,n = \Theta(n^2)$

Proof: Notice that if $n \geq 1$,

$$\frac{1}{2}n^2 + 3\,n \leq \frac{1}{2}n^2 + 3\,n^2 = \frac{7}{2}n^2,$$

so $\dfrac{1}{2}n^2 + 3\,n = O(n^2)$. Also, when $n \geq 0$,

$$\frac{1}{2}n^2 \leq \frac{1}{2}n^2 + 3n,$$

so $\dfrac{1}{2}n^2 + 3\,n = \Omega(n^2)$. Since $\frac{1}{2}n^2 + 3\,n = O(n^2)$ and $\frac{1}{2}n^2 + 3\,n = \Omega(n^2)$, then by Theorem 7.18, $\dfrac{1}{2}n^2 + 3\,n = \Theta(n^2)$ □

How do you use asymptotic notation to express that $f(n)$ grows slower than $g(n)$? Saying $f(n) = O(g(n))$ doesn't work, because that only tells us that $f(n)$ grows *no faster than* $g(n)$. It might grow slower, but it also might grow at the same rate. With the notation we have, the best way to express this idea is to say that $f(n) = O(g(n))$ and $f(n) \neq \Theta(g(n))$. But that is awkward. Let's learn a new notation for this instead. For technical reasons that we won't get into, this notation has to be defined somewhat differently than the others.

Definition 7.20. *Let f and g be nonnegative functions, with g being eventually non-zero. We say that $f(n)$ is* **little-o** *of $g(n)$, written $f(n) = o(g(n))$ iff*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

If $f(n) = o(g(n))$, $f(n)$ grows **asymptotically slower** *than $g(n)$.*

Example 7.21. You should be able to convince yourself that $3n+2 = o(n^2)$, but $3n+2 \neq o(n)$. Similarly, $n^2+n+4 = o(n^3)$ and $n^2+n+4 = o(n^4)$, but $n^2+n+4 \neq o(n^2)$ and $n^2+n+4 \neq o(n)$.

If you are not comfortable with limits you can still convince yourself of these statements by thinking of the informal definition. For instance, $n^2 + n + 4$ grows slower than n^3 so $n^2 + n + 4 = o(n^3)$. On the other hand, $n^2 + n + 4$ grows at the same rate (so *not* slower than) n^2, so $n^2 + n + 4 \neq o(n^2)$.

$\star$**Question 7.22.** Why do we require that $g(n)$ be eventually non-zero in the definition of little-o?

Answer ____

____

Little-omega (ω) can be defined similarly to little-o, but the value of the limit is ∞ instead of 0. We won't use ω very often.

$\star$**Question 7.23.** Big-O notation is analogous to $\leq$ in certain ways. If so, what would be the similar analogies for o and ω?

Answer ____

Note:

- *It is important to remember that a O-bound is only an* **upper bound**, *and that it may or may not be a tight bound. So if $f(n) = O(n^2)$, it is possible that $f(n) = 3n^2 + 4$, $f(n) = \log n$, or any other function that grows no faster than n^2. But we also know that $f(n) \neq n^3$ or any other function that grows faster than n^2.*
- *Conversely, a Ω-bound is only a* **lower bound**. *Thus, if $f(n) = \Omega(n \log n)$, it might be the case that $f(n) = 2^n$, but we know that $f(n) \neq 3n$, for instance.*
- *Unlike the other bounds, Θ-bounds are precise. So, if $f(n) = \Theta(n^2)$, then we know that f has a quadratic growth rate. It might be that $f(n) = 3n^2$, $2n^2 - 43n - 4$, or even $n^2 + n \log n$. But we are certain that the fastest growing term of f is cn^2 for some*

constant c.

⋆**Question 7.24.** Answer the following questions about the asymptotic notations.

(a) If $f(n) = \Theta(g(n))$, is it possible that $f(n) = o(g(n))$? Explain.

(b) If $f(n) = O(g(n))$, is it possible that $f(n) = o(g(n))$? Explain.

(c) If $f(n) = O(g(n))$, is it *certain* that $f(n) = o(g(n))$? Explain.

(d) If $f(n) = o(g(n))$, is it possible that $f(n) = O(g(n))$? Explain.

⋆**Evaluate 7.25.** Let $a_0, \ldots, a_k \in \mathbb{R}$, where $a_k > 0$. Prove that $a_k n^k + a_{k-1}n^{k-1} + \cdots + a_1 n + a_0 = O(n^k)$.

Solution 1: We can first eliminate all of the constants since they become irrelevant as n grows large enough. This leaves us with $n^k + n^{k-1} + \cdots + n = O(n^k)$. Next we can eliminate all terms growing slower than n^k, since they also become irrelevant as n grows. This leaves us with $n^k = O(n^k)$, and since they are the same, they are effectively theta of each other, and by definition, anything that is theta of something is also omega and O, so we can correctly say that $n^k = O(n^k)$, thus proving that $a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 = O(n^k)$.

Evaluation

Solution 2: Let $c = \sum_{i=0}^{k} |a_i|$. Then if $n \geq 1$,

$$\begin{aligned} a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 &\leq |a_k| n^k + |a_{k-1}| n^{k-1} + \cdots + |a_1| n + |a_0| \\ &\leq |a_k| n^k + |a_{k-1}| n^k + \cdots + |a_1| n^k + |a_0| n^k \\ &\leq \sum_{i=0}^{k} |a_i| n^k = cn^k. \end{aligned}$$

Therefore, $a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 = O(n^k)$.

Evaluation ______________________________

____

____

⋆**Exercise 7.26.** Assume that $f(n) = O(n^2)$ and $g(n) = O(n^3)$. What can you say about the relative growth rates of $f(n)$ and $g(n)$? In particular, does $g(n)$ grow faster than $f(n)$?

Answer ______________________________

____

____

Keep in mind that asymptotic notation only allows you to compare the asymptotic behavior of functions. Except for Θ-notation, it only provides a bound on the growth rate. For instance, knowing that $f(n) = O(g(n))$ only tells you that $f(n)$ grows no faster than $g(n)$. It is possible that $f(n)$ grows a lot slower than $g(n)$.

⋆**Exercise 7.27.** Let's test your understanding of the material so far. Answer each of the following true/false questions, giving a very brief justification/counterexample. Justifications can appeal to a definition and/or theorem. For counterexamples, use simple functions. For instance, $f(n) = n$ and $g(n) = n^2$.

(a) ____If $f(n) = O(g(n))$, then $f(n)$ grows faster than $g(n)$

(b) ____If $f(n) = \Theta(g(n))$, then $f(n)$ grows faster than $g(n)$

(c) ____If $f(n) = O(g(n))$, then $f(n)$ grows at the same rate as $g(n)$

(d) ___If $f(n) = \Omega(g(n))$, then $f(n)$ grows faster than $g(n)$

(e) ___If $f(n) = O(g(n))$, then $f(n) = \Omega(g(n))$

(f) ___If $f(n) = \Theta(g(n))$, then $f(n) = O(g(n))$

(g) ___If $f(n) = O(g(n))$, then $f(n) = \Theta(g(n))$

(h) ___If $f(n) = O(g(n))$, then $g(n) = O(f(n))$

7.1.2 Properties of the Notations

There are a lot of properties that hold for Big-O, Θ and Ω notation (and o and ω as well, but we won't focus on those ones in this section). We will only present a few of the most important ones. We provide proofs for some of the results. The rest can be proven without too much difficulty using the definitions of the notations.

Before we present the properties, it might be useful to think about the properties of things you are already familiar with. For instance, given real numbers x, y and z, you know that if $x \leq y$ and $y \leq z$, then $x \leq z$. This is just the transitive property of $\leq$. Similarly, you know that if $x \leq y$, then $a\,x \leq a\,y$ for any positive constant a. You can think of Big-O notation as being like $\leq$, Θ notation as being like $=$, and Ω notation as being like $\geq$. Many of the properties of $\leq$, $=$ and $\geq$ that you are already familiar with have an analog with Big-O, Θ, and Ω notation. But you need to be careful because the analogies are not exact. For instance, constants cannot be ignored with inequalities but can be ignored when using asymptotic notation.

Theorem 7.28. *The transitive property holds for Big-O, Θ, and Ω. That is,*

- *If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$*
- *If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $f(n) = \Theta(h(n))$*
- *If $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$, then $f(n) = \Omega(h(n))$*

Proof: *You will prove the transitive property of Big-O in Exercise 7.49. The proofs of the other two are very similar.* □

Theorem 7.28 is pretty intuitive. For instance, when applied to Big-O notation, Theorem 7.28 is essentially stating that if $g(n)$ is an upper bound on $f(n)$ and $h(n)$ is an upper bound on $g(n)$, then $h(n)$ is an upper bound for $f(n)$. Put another way, if $f(n)$ grows no faster than $g(n)$ and

$g(n)$ grows no faster than $h(n)$, then $f(n)$ grows no faster than $h(n)$. This makes perfect sense if you think about it for a few minutes.

Example 7.29. Let's take it for granted that $4n^2 + 3n + 17 = O(n^3)$ and $n^3 = O(n^4)$ (both of which you should be able to easily prove at this point). According to Theorem 7.28, we can conclude that $4n^2 + 3n + 17 = O(n^4)$.

Theorem 7.30. *Scaling by a constant factor*
If $f(n) = O(g(n))$, then for any $k > 0$, $kf(n) = O(g(n))$.
Similarly for Θ and Ω.

> **Proof:** *We will give the proof for Big-O notation. The other two proofs are similar. Assume $f(n) = O(g(n))$. Then by the definition of Big-O, there are positive constants c and n_0 such that $f(n) \le c\,g(n)$ for all $n \ge n_0$. Thus, if $n \ge n_0$,*
>
> $$k\,f(n) \le k\,c\,g(n) = c'g(n),$$
>
> *where $c' = k\,c$ is a positive constant. By the definition of Big-O, $kf(n) = O(g(n))$.*
> $\square$

Example 7.31. Example 7.19 showed that $\frac{1}{2}n^2 + 3\,n = \Theta(n^2)$. We can use Theorem 7.30 to conclude that $n^2 + 6\,n = \Theta(n^2)$ since $n^2 + 6\,n = 2\left(\frac{1}{2}n^2 + 3\,n\right)$.

Perhaps now is a good time to point out a related issue. Typically, we do not include constants inside asymptotic notations. For instance, although it is technically correct to say that $34n^3 + 2n^2 - 45n + 5 = O(5n^3)$ (or $O(50n^3)$, or any other constant you care to place there), it is best to just say it is $O(n^3)$. In particular, $\Theta(1)$ may be preferable to $\Theta(k)$.

Theorem 7.32. *Sums*
If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then

$$f_1(n) + f_2(n) = O(g_1(n) + g_2(n)) = O(max\{g_1(n), g_2(n)\}).$$

Similarly for Θ and Ω.

> **Proof:** *We will prove the assertion for Big-O. Assume $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$. Then there exists positive constants c_1 and n_1 such that for all $n \ge n_1$,*
>
> $$f_1(n) \le c_1 g_1(n),$$
>
> *and there exists positive constants c_2 and n_2 such that for all $n \ge n_2$,*
>
> $$f_2(n) \le c_2 g_2(n).$$
>
> *Let $c_0 = \max\{c_1, c_2\}$ and $n_0 = \max\{n_1, n_2\}$. Since n_0 is at least as large as n_1 and n_2, then for all $n \ge n_0$, $f_1(n) \le c_1 g_1(n)$ and $f_2(n) \le c_2 g_2(n)$. (If you don't see why this is, think about it. This is a subtle but important step.) Similarly, if $f_1(n) \le c_1 g_1(n)$, then clearly $f_1(n) \le c_0 g_1(n)$ since c_0 is at least as big as c_1 (and*

similarly for f_2). Then for all $n \geq n_0$, we have

$$\begin{aligned} f_1(n) + f_2(n) &\leq c_1 g_1(n) + c_2 g_2(n) \\ &\leq c_0 g_1(n) + c_0 g_2(n) \\ &\leq c_0[g_1(n) + g_2(n)] \\ &\leq c_0[\max\{g_1(n), g_2(n)\} + \max\{g_1(n), g_2(n)\}] \\ &\leq 2c_0 \max\{g_1(n), g_2(n)\} \\ &\leq c \max\{g_1(n), g_2(n)\}, \end{aligned}$$

where $c = 2c_0$. By the definition of Big-O, we have shown that $f_1(n) + f_2(n) = O(max\{g_1(n), g_2(n)\})$. □

Notice that in this proof we used $c = 2\max\{c_1, c_2\}$ and $n_0 = \max\{n_1, n_2\}$.

Without getting too technical, the previous theorem implies that you can upper bound the sum of two or more functions by finding the upper bound of the fastest growing of the functions. Another way of thinking about it is if you ever have two or more functions inside Big-O notation, you can simplify the notation by omitting the slower growing function(s). It should be pointed out that there is a subtle point in this result about how to precisely define the maximum of two functions. Most of the time the intuitive definition is sufficient so we won't belabor the point.

Example 7.33. Since we have previously shown that $5n^2 - 3n + 20 = O(n^2)$ and that $3n^3 - 2n^2 + 13n - 15 = O(n^3)$, we know that $(5n^2 - 3n + 20) + (3n^3 - 2n^2 + 13n - 15) = O(n^2 + n^3) = O(n^3)$.

Theorem 7.34. *Products*
If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then

$$f_1(n)f_2(n) = O(g_1(n)g_2(n)).$$

Similarly for Θ and Ω.

Example 7.35. Since we have previously shown that $5n^2 - 3n + 20 = O(n^2)$ and that $3n^3 - 2n^2 + 13n - 15 = O(n^3)$, we know that $(5n^2 - 3n + 20)(3n^3 - 2n^2 + 13n - 15) = O(n^2 n^3) = O(n^5)$. Notice that we could arrive at this same conclusion by multiplying the two polynomials and taking the highest term. However, this would require a lot more work than is necessary.

The next theorem essentially says that if $g(n)$ is an upper bound on $f(n)$, then $f(n)$ is a lower bound on $g(n)$. This makes perfect sense if you think about it.

Theorem 7.36. *Symmetry (sort of)*
$f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$.

It turns out that Θ defines an equivalence relation on the set of functions from $\mathbf{Z}^+$ to $\mathbf{Z}^+$. That is, it defines a partition on these functions, with two functions being in the same partition (or the same equivalence class) if and only if they have the same growth rate. But don't take our word for it. You will help to prove this fact next.

⋆**Fill in the details 7.37.** Let R be the relation on the set of functions from $\mathbf{Z}^+$ to $\mathbf{Z}^+$ such that $(f, g) \in R$ if and only if $f = \Theta(g)$. Show that R is an equivalence relation.

Proof: We need to show that R is reflexive, symmetric, and transitive.
Reflexive: Since $1 \cdot f(n) \leq f(n) \leq 1 \cdot f(n)$ for all $n \geq 1$, $f(n) = \Theta(f(n))$, so R is reflexive.
Symmetric: If $f(n) = \Theta(g(n))$, then there exist positive constants c_1, c_2, and n_0 such that ________________________________
This implies that

$$g(n) \leq \frac{1}{c_1} f(n) \text{ and } g(n) \geq \frac{1}{c_2} f(n) \text{ for all } n \geq n_0$$

which is equivalent to

$$\underline{\hspace{4cm}} \leq g(n) \leq \underline{\hspace{4cm}} \text{ for all } n \geq n_0.$$

Thus $g(n) = \Theta(f(n))$, and R is symmetric.
Transitive: If $f(n) = \Theta(g(n))$, then there exist positive constants c_1, c_2, and n_0 such that

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0.$$

Similarly if $g(n) = \Theta(h(n))$, then there exist positive constants c_3, c_4, and n_1 such that ________________________________
Then

$$f(n) \geq c_1\, g(n) \geq c_1 c_3\, h(n) \text{ for all } n \geq \max\{n_0, n_1\},$$

and

$$f(n) \leq \underline{\hspace{2cm}}\, g(n) \leq \underline{\hspace{2cm}}\, h(n) \text{ for all } n \geq \underline{\hspace{2cm}}$$

Thus,

$$\underline{\hspace{3cm}} \leq f(n) \leq \underline{\hspace{3cm}} \text{ for all } n \geq \max\{n_0, n_1\}.$$

Since $c_1 c_3$ and $c_2 c_4$ are both positive constants, $f(n) =$ ____________ by the definition of __________, so R is ____________. □

Example 7.38. The functions n^2, $3n^2 - 4n + 4$, $n^2 + \log n$, and $3n^2 + n + 1$ are all $\Theta(n^2)$. That is, they all have the same rate of growth and all belong to the same equivalence class.

⋆**Exercise 7.39.** Let's test your understanding of the material so far. Answer each of the following true/false questions, giving a very brief justification/counterexample. Justifications can appeal to a definition and/or theorem. For counterexamples, use simple functions. For instance, $f(n) = n$ and $g(n) = n^2$.

(a) ___If $f(n) = O(g(n))$, then $g(n) = \Omega(f(n))$

(b) ___If $f(n) = \Theta(g(n))$, then $f(n) = \Omega(g(n))$ and $f(n) = O(g(n))$

(c) ___If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n)+f_2(n) = O(max(g_1(n), g_2(n)))$

(d) ___$f(n) = O(g(n))$ iff $f(n) = \Theta(g(n))$

(e) ___$f(n) = O(g(n))$ iff $g(n) = O(f(n))$

(f) ___$f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$

(g) ___$f(n) = \Theta(g(n))$ iff $f(n) = \Omega(g(n))$ and $f(n) = O(g(n))$

(h) ___If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$

7.1.3 Proofs using the definitions

In this section we provide more examples and exercises that use the definitions to prove bounds.

The first example is annotated with comments (given in footnotes) about the techniques that are used in many of these proofs. We use the following terminology in our explanation. By *lower order term* we mean a term that grows slower, and *higher order* means a term that grows faster. The *dominating term* is the term that grows the fastest. For instance, in x^3+7x^2-4, the x^2 term is a lower order term than x^3, and x^3 is the dominating term. We will discuss common growth rates, including how they relate to each other, in Section 7.2. But for now we assume you know that x^5 grows faster than x^3, for instance.

Example 7.40. Find a tight bound on $f(n) = n^8 + 7n^7 - 10n^5 - 2n^4 + 3n^2 - 17$.

Solution: We will prove that $f(n) = \Theta(n^8)$. First, we will prove an upper bound for $f(n)$. It is clear that when $n \geq 1$,

$$\begin{aligned} n^8 + 7n^7 - 10n^5 - 2n^4 + 3n^2 - 17 &\leq n^8 + 7n^7 + 3n^2 \quad {}^{a} \\ &\leq n^8 + 7n^8 + 3n^8 \quad {}^{b} \\ &= 11n^8 \end{aligned}$$

Thus, we have

$$f(n) = n^8 + 7n^7 - 10n^5 - 2n^4 + 3n^2 - 17 \leq 11n^8 \text{ for all } n \geq 1,$$

and we have proved that $f(n) = O(n^8)$.

Now, we will prove the lower bound for $f(n)$. When $n \geq 1$,

$$\begin{aligned} n^8 + 7n^7 - 10n^5 - 2n^4 + 3n^2 - 17 &\geq n^8 - 10n^5 - 2n^4 - 17 \quad {}^{c} \\ &\geq n^8 - 10n^7 - 2n^7 - 17n^7 \quad {}^{d} \\ &= n^8 - 29n^7 \end{aligned}$$

Next, we need to find a value $c > 0$ such that $n^8 - 29n^7 \geq cn^8$. Doing a little algebra, we see that this is equivalent to $(1-c)n^8 \geq 29n^7$. When $n \geq 1$, we can divide by n^7 and obtain $(1-c)n \geq 29$. Solving for c we obtain

$$c \leq 1 - \frac{29}{n}.$$

If $n \geq 58$, then $c = 1/2$ suffices. We have just shown that if $n \geq 58$, then

$$f(n) = n^8 + 7n^7 - 10n^5 - 2n^4 + 3n^2 - 17 \geq \frac{1}{2}n^8.$$

Thus, $f(n) = \Omega(n^8)$. Since we have shown that $f(n) = \Omega(n^8)$ and that $f(n) = O(n^8)$, we have shown that $f(n) = \Theta(n^8)$.

[a] We can upper bound any function by removing the lower order terms with negative coefficients, as long as $n \geq 0$.

[b] We can upper bound any function by replacing lower order terms that have positive coefficients by the dominating term with the same coefficients. Here, we must make sure that the dominating term is larger than the given term for all values of n larger than some threshold n_0, and we must make note of the threshold value n_0.

[c] We can lower bound any function by removing the lower order terms with positive coefficients, as long as $n \geq 0$.

[d] We can lower bound any function by replacing lower order terms with negative coefficients by a sub-dominating term with the same coefficients. (By sub-dominating, I mean one which dominates all but the dominating term.) Here, we must make sure that the sub-dominating term is larger than the given term for all values of n larger than some threshold n_0, and we must make note of the threshold value n_0. Making a wise choice for which sub-dominating term to use is crucial in finishing the proof.

Let's see another example of a Ω proof. You should note the similarities between this and the second half of the proof in the previous example.

Example 7.41. Show that $(n \log n - 2\,n + 13) = \Omega(n \log n)$

Proof: We need to show that there exist positive constants c and n_0 such that

$$c\,n \log n \leq n \log n - 2\,n + 13 \text{ for all } n \geq n_0.$$

Since $n \log n - 2\,n \leq n \log n - 2\,n + 13$, we will instead show that

$$c\,n \log n \leq n \log n - 2\,n,$$

which is equivalent to

$$c \leq 1 - \frac{2}{\log n}, \text{ when } n > 1.$$

If $n \geq 8$, then $2/(\log n) \leq 2/3$, and picking $c = 1/3$ suffices. In other words, we have just shown that if $n \geq 8$,

$$\frac{1}{3}\,n \log n \leq n \log n - 2\,n.$$

Thus if $c = 1/3$ and $n_0 = 8$, then for all $n \geq n_0$, we have

$$c\,n \log n \leq n \log n - 2\,n \leq n \log n - 2\,n + 13.$$

Thus $(n \log n - 2\,n + 13) = \Omega(n \log n)$. $\square$

⋆**Fill in the details 7.42.** Show that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$

Proof: We need to find positive constants c_1, c_2, and n_0 such that

$$\underline{\qquad\qquad} \leq \frac{1}{2}n^2 - 3\,n \leq \underline{\qquad\qquad} \text{ for all } n \geq n_0$$

Dividing by n^2, we get

$$c_1 \leq \underline{\qquad\qquad} \leq c_2.$$

Notice that if $n \geq 10$,

$$\frac{1}{2} - \frac{3}{n} \geq \frac{1}{2} - \frac{3}{10} = \underline{\qquad\qquad\qquad},$$

so we can choose $c_1 = 1/5$. If $n \geq 10$, we also have that $\frac{1}{2} - \frac{3}{n} \leq \frac{1}{2}$, so we can choose $c_2 = 1/2$. Thus, we have shown that

$$\underline{\qquad\qquad} \leq \frac{1}{2}n^2 - 3\,n \leq \underline{\qquad\qquad} \text{ for all } n \geq \underline{\qquad}.$$

Therefore, $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. $\square$

⋆**Question 7.43.** In the previous proof, we claimed that if $n \geq 10$,

$$\frac{1}{2} - \frac{3}{n} \geq \frac{1}{2} - \frac{3}{10}.$$

Why is this true?

Answer ______________________________

____

____

Example 7.44. Show that $(\sqrt{2})^{\log n} = O(\sqrt{n})$, where the base of the log is 2.

Proof: It is not too hard to see that

$$(\sqrt{2})^{\log n} = n^{\log \sqrt{2}} = n^{\log 2^{1/2}} = n^{\frac{1}{2}\log 2} = n^{\frac{1}{2}} = \sqrt{n}.$$

Thus it is clear that $(\sqrt{2})^{\log n} = O(\sqrt{n})$. □

Note: *You may be confused by the previous proof. It seems that we never showed that $(\sqrt{2})^{\log n} \leq c\sqrt{n}$ for some constant c. But we essentially did by showing that $(\sqrt{2})^{\log n} = \sqrt{n}$ since this implies that $(\sqrt{2})^{\log n} \leq 1\sqrt{n}$.*

We actually proved something stronger than was required. That is, since we proved the two functions are equal, it is in fact true that $(\sqrt{2})^{\log n} = \Theta(\sqrt{n})$. But we were only asked to prove that $(\sqrt{2})^{\log n} = O(\sqrt{n})$.

In general, if you need to prove a Big-O bound, you may instead prove a Θ bound, and the Big-O bound essentially comes along for the ride.

⋆**Question 7.45.** In our previous note we mentioned that if you prove a Θ bound, you get the Big-O bound for free.

(a) What theorem implies this?

Answer ______________________________

(b) If we prove $f(n) = O(g(n))$, does that imply that $f(n) = \Theta(g(n))$? In other words, does it work the other way around? Explain, giving an appropriate example.

Answer ______________________________

____

⋆**Exercise 7.46.** Show that $n! = O(n^n)$. (Don't give up too easily on this one—the proof is very short and only uses elementary algebra.)

Example 7.47. Show that $\log(n!) = O(n \log n)$

Proof: It should be clear that if $n \geq 1$, $n! \leq n^n$ (especially after completing the previous exercise). Taking logs of both sides of that inequality, we obtain

$$\log n! \leq \log(n^n) = n \log n.$$

Therefore $\log n! = O(n \log n)$. $\square$

The last step used the fact that $\log(f(n)^a) = a \log(f(n))$, a fact that we assume you have seen previously (but may have forgotten).

Proving properties of the asymptotic notations is actually no more difficult than the rest of the proofs we have seen. You have already seen a few and helped write one. Here we provide one more example and then ask you to prove another result on your own.

Example 7.48. Prove that if $f(n) = O(g(n))$ and $g(n) = O(f(n))$, then $f(n) = \Theta(g(n))$.

Proof: If $f(n) = O(g(n))$, then there are positive constants c_2 and n_0' such that

$$f(n) \leq c_2\, g(n) \text{ for all } n \geq n_0'$$

Similarly, if $g(x) = O(f(x))$, then there are positive constants c_1' and n_0'' such that

$$g(n) \leq c_1'\, f(n) \text{ for all } n \geq n_0''.$$

We can divide this by c_1' to obtain

$$\frac{1}{c_1'} g(n) \leq f(n) \text{ for all } n \geq n_0''.$$

Setting $c_1 = 1/c_1'$ and $n_0 = \max\{n_0', n_0''\}$, we have

$$c_1 g(n) \leq f(n) \leq c_2\, g(n) \text{ for all } n \geq n_0.$$

Thus, $f(x) = \Theta(g(x))$. □

⋆**Exercise 7.49.** Let $f(x) = O(g(x))$ and $g(x) = O(h(x))$. Show that $f(x) = O(h(x))$. That is, prove Theorem 7.28 for Big-O notation.
Proof:

7.1.4 Proofs using limits

So far we have used the definitions of the various notations in all of our proofs. The following theorem provides another technique that is often much easier, assuming you understand and are comfortable with limits.

Theorem 7.50. *Let $f(n)$ and $g(n)$ be functions such that*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = A.$$

Then

1. *If $A = 0$, then $f(n) = O(g(n))$, and $f(n) \neq \Theta(g(n))$. That is, $f(n) = o(g(n))$.*
2. *If $A = \infty$, then $f(n) = \Omega(g(n))$, and $f(n) \neq \Theta(g(n))$. That is, $f(n) = \omega(g(n))$.*
3. *If $A \neq 0$ is finite, then $f(n) = \Theta(g(n))$.*

If the above limit does not exist, then you need to resort to using the definitions or using some other technique. Luckily, in the analysis of algorithms the above approach works most of the time.

Before we see some examples, let's review a few limits you should know.

Theorem 7.51. *Let a and c be real numbers. Then*

(a) $\lim_{n\to\infty} a = a$

(b) If $a > 0$, $\lim_{n\to\infty} n^a = \infty$

(c) If $a < 0$, $\lim_{n\to\infty} n^a = 0$

(d) If $a > 1$, $\lim_{n\to\infty} a^n = \infty$

(e) If $0 < a < 1$, $\lim_{n\to\infty} a^n = 0$

(f) If $c > 0$, $\lim_{n\to\infty} \log_c n = \infty$.

Example 7.52. The following are examples based on Theorem 7.51.

(a) $\lim_{n\to\infty} 13 = 13$

(b) $\lim_{n\to\infty} n = \infty$

(c) $\lim_{n\to\infty} n^4 = \infty$

(d) $\lim_{n\to\infty} n^{1/2} = \infty$

(e) $\lim_{n\to\infty} n^{-2} = 0$

(f) $\lim_{n\to\infty} \left(\frac{1}{2}\right)^n = 0$

(g) $\lim_{n\to\infty} 2^n = \infty$

(h) $\lim_{n\to\infty} \log_2 n = \infty$

Now it's your turn to try a few.

⋆**Exercise 7.53.** Evaluate the following limits

(a) $\lim_{n\to\infty} \log_{10} n =$

(b) $\lim_{n\to\infty} n^3 =$

(c) $\lim_{n\to\infty} 3^n =$

(d) $\lim_{n\to\infty} \left(\frac{3}{2}\right)^n =$

(e) $\lim_{n\to\infty} \left(\frac{2}{3}\right)^n =$

(f) $\lim_{n\to\infty} n^{-1} =$

(g) $\lim_{n\to\infty} 8675309 =$

Example 7.54. Prove that $5n^8 = \Theta(n^8)$ using Theorem 7.50.

Solution: Notice that

$$\lim_{n\to\infty} \frac{5n^8}{n^8} = \lim_{n\to\infty} 5 = 5,$$

so $f(n) = \Theta(n^8)$ by Theorem 7.50 (case 3).

The following theorem often comes in handy when using Theorem 7.50.

Theorem 7.55. *If* $\lim_{n\to\infty} f(n) = \infty$, *then* $\lim_{n\to\infty} \frac{1}{f(n)} = 0$.

Example 7.56. Prove that $n^2 = o(n^4)$ using Theorem 7.50.

Solution: Notice that

$$\lim_{n\to\infty} \frac{n^2}{n^4} = \lim_{n\to\infty} \frac{1}{n^2} = 0,$$

so $f(n) = o(n^4)$ by Theorem 7.50 (case 1).

$\star$**Question 7.57.** The proof in the previous example used Theorems 7.51 and 7.55. How and where?

Answer ____

____

⋆**Exercise 7.58.** Prove that $3x^3 = \Omega(x^2)$ using Theorem 7.50. Which case did you use?

Here are a few more useful properties of limits. Read carefully. These do not apply in all situations.

Theorem 7.59. *Let a be a finite real number and let $\lim_{n\to\infty} f(n) = A$ and $\lim_{n\to\infty} g(n) = B$, where A and B are finite real numbers. Then*

(a) $\lim_{n\to\infty} a\, f(n) = a\, A$

(b) $\lim_{n\to\infty} f(n) \pm g(n) = A \pm B$

(c) $\lim_{n\to\infty} f(n)g(n) = AB$

(d) If $B \neq 0$, $\lim_{n\to\infty} \dfrac{f(n)}{g(n)} = \dfrac{A}{B}$

We usually use the results from the previous theorem without explicitly mentioning them.

Example 7.60. Find a tight bound on $f(x) = x^8 + 7x^7 - 10x^5 - 2x^4 + 3x^2 - 17$ using Theorem 7.50.

Solution: We guess (or know, if we remember the solution to Example 7.40) that $f(x) = \Theta(x^8)$. To prove this, notice that

$$\begin{aligned}
\lim_{x\to\infty} \frac{x^8+7x^7-10x^5-2x^4+3x^2-17}{x^8} &= \lim_{x\to\infty} \frac{x^8}{x^8} + \frac{7x^7}{x^8} - \frac{10x^5}{x^8} - \frac{2x^4}{x^8} + \frac{3x^2}{x^8} - \frac{17}{x^8} \\
&= \lim_{x\to\infty} 1 + \frac{7}{x} - \frac{10}{x^3} - \frac{2}{x^4} + \frac{3}{x^6} - \frac{17}{x^8} \\
&= 1 + 0 - 0 - 0 + 0 - 0 = 1
\end{aligned}$$

Thus, $f(x) = \Theta(x^8)$ by the Theorem 7.50.

Compare the proof above with the proof given in Example 7.40. It should be pretty obvious that using Theorem 7.50 makes the proof a lot easier. Let's see another example that lets us compare the two proof methods.

Example 7.61. Prove that $f(x) = x^4 - 23x^3 + 12x^2 + 15x - 21 = \Theta(x^4)$.

Proof #1

We will use the definition of Θ. It is clear that when $x \geq 1$,

$$x^4 - 23x^3 + 12x^2 + 15x - 21 \leq x^4 + 12x^2 + 15x \leq x^4 + 12x^4 + 15x^4 = 28x^4.$$

Also, if $x \geq 88$, then $\frac{1}{2}x^4 \geq 44x^3$ or $-44x^3 \geq -\frac{1}{2}x^4$, so we have that

$$x^4 - 23x^3 + 12x^2 + 15x - 21 \geq x^4 - 23x^3 - 21 \geq x^4 - 23x^3 - 21x^3 = x^4 - 44x^3 \geq \frac{1}{2}x^4.$$

Thus

$$\frac{1}{2}x^4 \leq x^4 - 23x^3 + 12x^2 + 15x - 21 \leq 28x^4, \text{ for all } x \geq 88.$$

We have shown that $f(x) = x^4 - 23x^3 + 12x^2 + 15x - 21 = \Theta(x^4)$. □

If you did not follow the steps in this first proof, you should really review your algebra rules.

Proof #2

Since

$$\begin{aligned}
\lim_{x\to\infty} \frac{x^4 - 23x^3 + 12x^2 + 15x - 21}{x^4} &= \lim_{x\to\infty} \frac{x^4}{x^4} - \frac{23x^3}{x^4} + \frac{12x^2}{x^4} + \frac{15x}{x^4} - \frac{21}{x^4} \\
&= \lim_{x\to\infty} 1 - \frac{23}{x} + \frac{12}{x^2} + \frac{15}{x^3} - \frac{21}{x^4} \\
&= \lim_{x\to\infty} 1 - 0 + 0 + 0 - 0 = 1,
\end{aligned}$$

$f(x) = x^4 - 23x^3 + 12x^2 + 15x - 21 = \Theta(x^4)$ □

Example 7.62. Prove that $n(n+1)/2 = O(n^3)$ using Theorem 7.50.

Proof: Notice that $\lim_{n\to\infty} \frac{n(n+1)/2}{n^3} = \lim_{n\to\infty} \frac{n^2+n}{2n^3} = \lim_{n\to\infty} \frac{1}{2n} + \frac{1}{2n^2} = 0 + 0 = 0$, so $n(n+1)/2 = o(n^3)$, which implies that $n(n+1)/2 = O(n^3)$. □

⋆**Exercise 7.63.** Prove that $n(n+1)/2 = \Theta(n^2)$ using Theorem 7.50.
Proof:

⋆**Exercise 7.64.** Prove that $2^x = O(3^x)$

(a) Using Theorem 7.50.

(b) Using the definition of Big-O.

Now is probably a good time to recall a very useful theorem for computing limits, called **l'Hopital's Rule**. The version presented here is restricted to limits where the variable approaches infinity since those are the only limits of interest in our context.

Theorem 7.65 (l'Hopital's Rule)**.** *Let $f(x)$ and $g(x)$ be differentiable functions. If*

$$\lim_{x\to\infty} f(x) = \lim_{x\to\infty} g(x) = 0 \text{ or}$$

$$\lim_{x\to\infty} f(x) = \lim_{x\to\infty} g(x) = \infty,$$

then

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)}$$

Example 7.66. Since $\lim_{x\to\infty} 3x = \infty$ and $\lim_{x\to\infty} x^2 = \infty$,

$$\begin{aligned}
\lim_{x\to\infty} \frac{3x}{x^2} &= \lim_{x\to\infty} \frac{3}{2x} \quad \text{(l'Hopital)} \\
&= \frac{3}{2} \lim_{x\to\infty} \frac{1}{x} \\
&= \frac{3}{2} 0 \\
&= 0.
\end{aligned}$$

Example 7.67. Since $\lim_{x\to\infty} 3x^2 + 4x - 9 = \infty$ and $\lim_{x\to\infty} 12x = \infty$,

$$\begin{aligned}
\lim_{x\to\infty} \frac{3x^2 + 4x - 9}{12x} &= \lim_{x\to\infty} \frac{6x+4}{12} \quad \text{(l'Hopital)} \\
&= \lim_{x\to\infty} \frac{1}{2}x + \frac{1}{3} \\
&= \infty
\end{aligned}$$

Now let's apply it to proving asymptotic bounds.

Example 7.68. Show that $\log x = O(x)$.

Proof: Notice that

$$\begin{aligned}
\lim_{x\to\infty} \frac{\log x}{x} &= \lim_{x\to\infty} \frac{\frac{1}{x}}{1} \quad \text{(l'Hopital)} \\
&= \lim_{x\to\infty} \frac{1}{x} = 0.
\end{aligned}$$

Therefore, $\log x = O(x)$. □

We should mention that applying l'Hopital's Rule in the first step is legal since

$$\lim_{x\to\infty} \log x = \lim_{x\to\infty} x = \infty.$$

Example 7.69. Prove that $x^3 = O(2^x)$.

Proof: Notice that

$$\begin{aligned}
\lim_{x\to\infty} \frac{x^3}{2^x} &= \lim_{x\to\infty} \frac{3x^2}{2^x \ln(2)} \quad \text{(l'Hopital)} \\
&= \lim_{x\to\infty} \frac{6x}{2^x \ln^2(2)} \quad \text{(l'Hopital)} \\
&= \lim_{x\to\infty} \frac{6}{2^x \ln^3(2)} \quad \text{(l'Hopital)} \\
&= 0.
\end{aligned}$$

Therefore, $x^3 = O(2^x)$.

As in the previous example, at each step we checked that the functions on both the top and bottom go to infinity as n goes to infinity before applying l'Hopital's Rule. Notice that we did not apply it in the final step since 6 does not go to infinity. $\square$

⋆**Evaluate 7.70.** Prove that 7^x is an upper bound for 5^x, but that it is not a tight bound.

Proof 1: This is true if and only if 7^x always grows faster than 5^x which means $7^x - 5^x > 0$ for all $x \neq 0$. If it is a tight bound, then $7^x - 5^x = 0$, which is only true for $x = 0$. So 7^x is an upper bound on 5^x, but not a tight bound.

Evaluation ____

Proof 2: $\lim_{x\to\infty} \frac{5^x}{7^x} = \lim_{x\to\infty} \frac{x \log 5}{x \log 7}$. Both go to infinity, but $x \log 7$ gets there faster, showing that $5^x = O(7^x)$.

Evaluation ____

Proof 3: $\lim_{x\to\infty} \frac{7^x}{5^x} = \lim_{x\to\infty} \left(\frac{7}{5}\right)^x = \infty$ since $7/5 > 1$. Thus $5^x = O(7^x)$ by the limit theorem.

Evaluation ____

We should mention that it is important to remember to verify that l'Hopital's Rule applies before just blindly taking derivatives. You can actually get the incorrect answer if you apply it when it should not be applied.

Example 7.71. Find and prove a simple tight bound for $\sqrt{5n^2 - 4n + 12}$.

Solution: We will show that $\sqrt{5n^2 - 4n + 12} = \Theta(n)$. Since we are letting n go to infinity, we can assume that $n > 0$. In this case, $n = \sqrt{n^2}$. Using this, we can see that

$$\lim_{n\to\infty} \frac{\sqrt{5n^2 - 4n + 12}}{n} = \lim_{n\to\infty} \sqrt{\frac{5n^2 - 4n + 12}{n^2}} = \lim_{n\to\infty} \sqrt{5 - \frac{4}{n} + \frac{12}{n^2}} = \sqrt{5}.$$

Therefore, $\sqrt{5n^2 - 4n + 12} = \Theta(n)$.

⋆**Exercise 7.72.** Find and prove a good simple upper bound on $n\ln(n^2+1)+n^2\ln n$.

(a) Using the definition of Big-O.

(b) Using Theorem 7.50. You will probably need to use l'Hopital's Rule a few times.

Example 7.73. Find and prove a simple tight bound for $n\log(n^2)+(n-1)^2\log(n/2)$.

Solution: First notice that

$$n\log(n^2)+(n-1)^2\log(n/2) = 2n\log n + (n-1)^2(\log n - \log 2).$$

We can see that this is $\Theta(n^2\log n)$ since

$$\begin{aligned}
\lim_{n\to\infty}\frac{n\log(n^2)+(n-1)^2\log(n/2)}{n^2\log n} &= \lim_{n\to\infty}\frac{2n\log n+(n-1)^2(\log n-\log 2)}{n^2\log n}\\
&= \lim_{n\to\infty}\frac{2}{n}+\frac{(n-1)^2}{n^2}\frac{(\log n-\log 2)}{\log n}\\
&= \lim_{n\to\infty}\frac{2}{n}+\left(1-\frac{1}{n}\right)^2\left(1-\frac{\log 2}{\log n}\right)\\
&= 0+(1-0)^2(1-0)=1.
\end{aligned}$$

⋆**Exercise 7.74.** Find and prove a simple tight bound for $(n^2-1)^5$. You may use either the formal definition of Θ or Theorem 7.50. (The solution uses Theorem 7.50.)

⋆**Exercise 7.75.** Find and prove a simple tight bound for $2^{n+1} + 5^{n-1}$. You may use either the formal definition of Θ or Theorem 7.50. (The solution uses Theorem 7.50.)

7.2 Common Growth Rates

In this section we will take a look at the relative growth rates of various functions.

Figure 7.1 shows the value of several functions for various values of n to give you an idea of their relative rates of growth. The bottom of the table is labeled relative to the last column so you can get a sense of how slow $\log m$ and $\log(\log m)$ grow. For instance, the final row is showing that $\log_2(262144) = 18$ and $\log_2(\log_2(262144)) = 2.890$.

Figures 7.2 and 7.3 demonstrate that as n increases, the constants and lower-order terms do not matter. For instance, notice that although $100n$ is much larger than 2^n for small values of n, as n increases, 2^n quickly gets much larger than $100n$. Similarly, in Figure 7.3, notice that when $n = 74$, n^3 and $n^3 + 234$ are virtually the same.

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
0.6931	2	1.39	4	8	4
1.099	3	3.30	9	27	8
1.386	4	5.55	16	64	16
1.609	5	8.05	25	125	32
1.792	6	10.75	36	216	64
1.946	7	13.62	49	343	128
2.079	8	16.64	64	512	256
2.197	9	19.78	81	729	512
2.303	10	23.03	100	1000	1024
2.398	11	26.38	121	1331	2048
2.485	12	29.82	144	1728	4096
2.565	13	33.34	169	2197	8192
2.639	14	36.95	196	2744	16384
2.708	15	40.62	225	3375	32768
2.773	16	44.36	256	4096	65536
2.833	17	48.16	289	4913	131072
2.890	18	52.03	324	5832	262144
$\log \log m$	$\log m$				m

Figure 7.1: A comparison of growth rates

n	$100n$	n^2	$11n^2$	n^3	2^n
1	100	1	11	1	2
2	200	4	44	8	4
3	300	9	99	27	8
4	400	16	176	64	16
5	500	25	275	125	32
6	600	36	396	216	64
7	700	49	539	343	128
8	800	64	704	512	256
9	900	81	891	729	512
10	1000	100	1100	1000	1024
11	1100	121	1331	1331	2048
12	1200	144	1584	1728	4096
13	1300	169	1859	2197	8192
14	1400	196	2156	2744	16384
15	1500	225	2475	3375	32768
16	1600	256	2816	4096	65536
17	1700	289	3179	4913	131072
18	1800	324	3564	5832	262144
19	1900	361	3971	6859	524288

Figure 7.2: Constants don't matter

n	n^2	$n^2 - n$	$n^2 + 99$	n^3	$n^3 + 234$
2	4	2	103	8	242
6	36	30	135	216	450
10	100	90	199	1000	1234
14	196	182	295	2744	2978
18	324	306	423	5832	6066
22	484	462	583	10648	10882
26	676	650	775	17576	17810
30	900	870	999	27000	27234
34	1156	1122	1255	39304	39538
38	1444	1406	1543	54872	55106
42	1764	1722	1863	74088	74322
46	2116	2070	2215	97336	97570
50	2500	2450	2599	125000	125234
54	2916	2862	3015	157464	157698
58	3364	3306	3463	195112	195346
62	3844	3782	3943	238328	238562
66	4356	4290	4455	287496	287730
70	4900	4830	4999	343000	343234
74	5476	5402	5575	405224	405458

Figure 7.3: Lower-order terms don't matter

Figures 7.4 through 7.8 give a graphical representation of relative growth rates of functions. In these diagrams, `**` means exponentiation. For instance, `x**2` means x^2.

It is important to point out that you should *never* rely on the graphs of functions to determine relative growth rates. That is the point of Figures 7.6 and 7.7. Although graphs sometimes give you an accurate picture of the relative growth rates of the functions, they might just as well present a distorted view of the data depending on the values that are used on the axes. Instead,

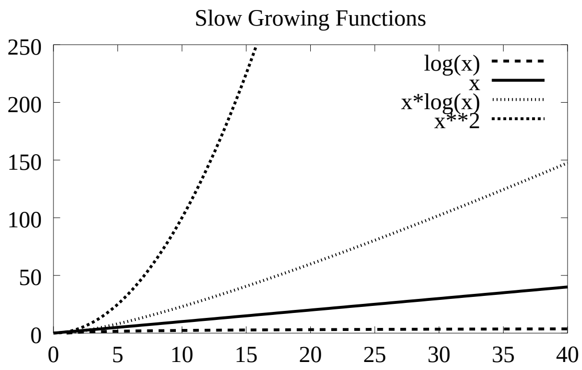


Figure 7.4: The growth rate of some slow growing functions.

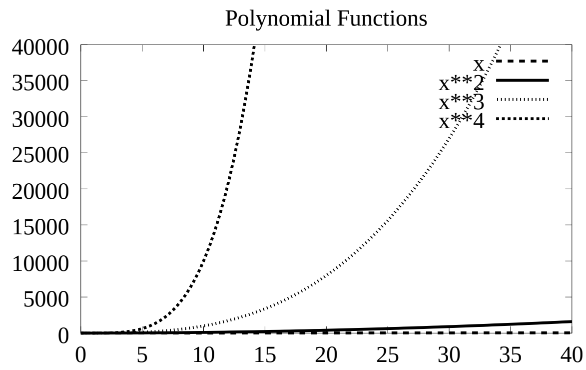


Figure 7.5: The growth rate of some polynomials.

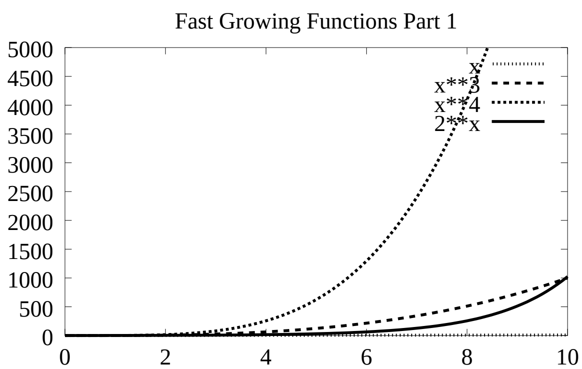


Figure 7.6: The growth rate of some polynomials and an exponential. This graph makes it look like x^4 is growing faster than 2^x. But see Figure 7.7.

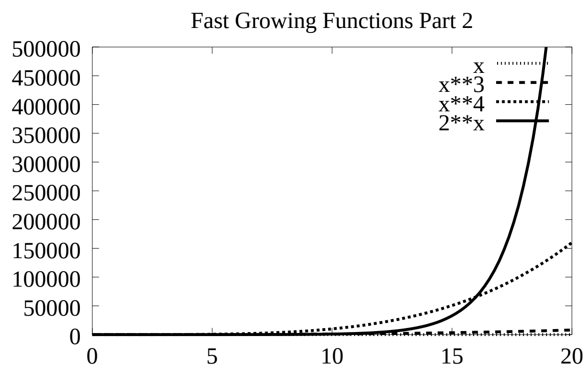


Figure 7.7: The growth rate of some polynomials and an exponential. If we make n large enough, it is clear that 2^n grows faster than n^4.

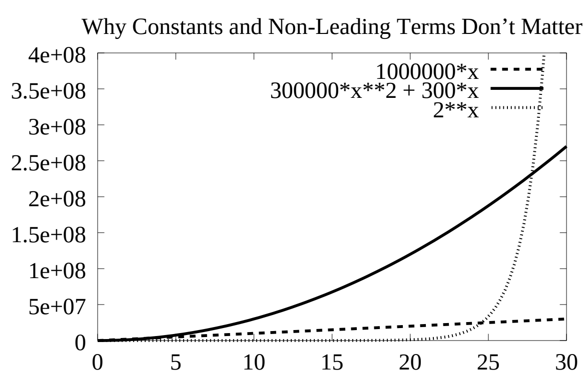


Figure 7.8: Notice that as n gets larger, the constants eventually matter less.

you should use the techniques we develop in this section.

Next we present some of the most important results about the relative growth rate of some common functions. We will ask you to prove each of them. Theorems 7.50 and 7.65 will help you do so. You will notice that most of the theorems are using little-o, not Big-O. Hopefully you understand the difference. If not, review those definitions before continuing.

We begin with something that is pretty intuitive: higher powers grow faster than lower powers.

Theorem 7.76. *Let $a < b$ be real numbers. Then $n^a = o(n^b)$.*

Example 7.77. According to Theorem 7.76, $n^2 = o(n^3)$ and $n^5 = o(n^{5.1})$.

⋆**Exercise 7.78.** Prove Theorem 7.76. (Hint: Use Theorem 7.50 and do a little algebra before you try to compute the limit.)

The next theorem tells us that exponentials with different bases do not grow at the same rate. More specifically, the higher the base, the faster the growth rate.

Theorem 7.79. *Let $0 < a < b$ be real numbers. Then $a^n = o(b^n)$.*

Example 7.80. According to Theorem 7.79, $2^n = o(5^n)$ and $4^n = o(4.5^n)$.

⋆**Exercise 7.81.** Prove Theorem 7.79. (See the hint for Exercise 7.78.)

Recall that a logarithmic function is the inverse of an exponential function. That is, $b^x = n$ is equivalent to $x = \log_b n$. The following identity is very useful.

Theorem 7.82. *Let a, b, and x be positive real numbers with $a \neq 1$ and $b \neq 1$. Then*

$$\log_a x = \frac{\log_b x}{\log_b a}.$$

Example 7.83. Most calculators can compute $\ln n$ or $\log_{10} n$, but are unable to compute logarithms with any given base. But Theorem 7.82 allows you to do so. For instance, you can compute $\log_2 39$ as $\log_{10} 39/\log_{10} 2$.

Notice that the formula in Theorem 7.82 can be rearranged as $(\log_b a)(\log_a x) = \log_b x$. This form should make it evident that changing the base of a logarithm just changes the value by a constant amount. This leads to the following result.

Corollary 7.84. *Let a and b be positive real numbers with $a \neq 1$ and $b \neq 1$. Then $\log_a n = \Theta(\log_b n)$.*

Proof: *Follows from the definition of Θ and Theorem 7.82.* □

Example 7.85. According to Corollary 7.84, $\log_2 n = \Theta(\log_{10} n)$ and $\ln n = \Theta(\log_2 n)$.

Corollary 7.84 is stating that all logarithms have the same rate of growth regardless of their bases. That is, the base of a logarithm does not matter when it is used in asymptotic notation. Because of this, the base is often omitted in asymptotic notation. In computer science, it is usually safe to assume that the base of logarithms is 2 if it is not specified.

⋆**Exercise 7.86.** Indicate whether each of the following is *true* (T) or *false* (F).

(a) ____ $2^n = \Theta(3^n)$

(b) ____ $2^n = o(3^n)$

(c) ____ $3^n = O(2^n)$

(d) ____ $\log_3 n = \Theta(\log_2 n)$

(e) ____ $\log_2 n = O(\log_3 n)$

(f) ____ $\log_{10} n = o(\log_3 n)$

Next we see that logarithms grow slower than positive powers of n.

Theorem 7.87. *Let $b > 0$ and $c > 0$ be real numbers. Then $\log_c(n) = o(n^b)$.*

Example 7.88. According to Theorem 7.87, $\log_2 n = o(n^2)$, $\log_{10} n = o(n^{1.01})$, and $\ln n = o(\sqrt{n})$.

⋆**Exercise 7.89.** Prove Theorem 7.87. (Hint: This is easy if you use Theorems 7.50 and 7.65)

More generally, the next theorem states that any positive power of a logarithm grows slower than any positive power of n. Since this one is a little tricky, we will provide the proof. In case you have not seen this notation before, you should know that $\log^a n$ means $(\log n)^a$, which is *not* the same thing as $\log(n^a)$.

Theorem 7.90. *Let $a > 0$, $b > 0$, and $c > 0$ be real numbers. Then $\log_c^a(n) = o(n^b)$. In other words, any power of a log grows slower than any polynomial.*

Proof: *First, we need to know that if $a > 0$ is a constant, and $\lim_{n\to\infty} f(n) = C$, then*

$$\lim_{n\to\infty} (f(n))^a = \left(\lim_{n\to\infty} f(n)\right)^a = C^a.$$

Using this and the limit computed in the proof of Theorem 7.87, we have that

$$\lim_{n\to\infty} \frac{\log_c^a(n)}{n^b} = \lim_{n\to\infty} \left(\frac{\log_c(n)}{n^{b/a}}\right)^a = \left(\lim_{n\to\infty} \frac{\log_c(n)}{n^{b/a}}\right)^a = 0^a = 0.$$

Thus, Theorem 7.50 tells us that $\log_c^a(n) = o(n^b)$. □

Example 7.91. According to Theorem 7.90, $\log_2^4 n = o(n^2)$, $\ln^{10} n = o(\sqrt{n})$, and $\log_{10}^{1,000,000} n = o(n^{.00000001})$.

Finally, any exponential function with base larger than 1 grows faster than any polynomial.

Theorem 7.92. *Let $a > 0$ and $b > 1$ be real numbers. Then $n^a = o(b^n)$.*

Example 7.93. According to Theorem 7.92, it is easy to see that $n^2 = o(2^n)$, $n^{15} = o(1.5^n)$, and $n^{1{,}000{,}000} = o(1.0000001^n)$.

There are several ways to prove Theorem 7.92, including using repeated applications of l'Hopital's rule, using induction, or doing a little algebraic manipulation and using one of several clever tricks. But the techniques are beyond what we generally need in the course, so we will omit a proof (and, perhaps more importantly, we will not ask you to provide a proof!).

⋆**Fill in the details 7.94.** Fill in the following blanks with Θ, Ω, O, or o. You should give the most precise answer possible. (e.g. If you put O, but the correct answer is o, your answer is correct but not precise enough.)

(a) $n(n-1) = \underline{\hspace{3em}}(500n^2)$.

(b) $50n^2 = \underline{\hspace{3em}}(.001n^4)$.

(c) $\log_2 n = \underline{\hspace{3em}}(\ln n)$.

(d) $\log_2(n^2) = \underline{\hspace{3em}}(\log_2^2(n))$.

(e) $2^{n-1} = \underline{\hspace{3em}}(2^n)$.

(f) $5^n = \underline{\hspace{3em}}(3^n)$.

(g) $(n-1)! = \underline{\hspace{3em}}(n!)$.

(h) $n^3 = \underline{\hspace{3em}}(2^n)$.

(i) $\log^{100} n = \underline{\hspace{3em}}(1.01^n)$.

(j) $\log^{100} n = \underline{\hspace{3em}}(n^{1.01})$.

An alternative notation for little-o is $\ll$. In other words, $f(n) = o(g(n))$ iff $f(n) \ll g(n)$. This notation is useful in certain contexts, including the following comparison of the growth rate of common functions. The previous theorems in this section provide proofs of some of these relationships. The others are given without proof.

Theorem 7.95. *Here are some relationships between the growth rates of common functions:*

$$c \ll \log n \ll log^2 n \ll \sqrt{n} \ll n \ll n \log n \ll n^{1.1} \ll n^2 \ll n^3 \ll n^4 \ll 2^n \ll 3^n \ll n! \ll n^n$$

You should convince yourself that each of the relationships given in the previous theorem is correct. You should also memorize them or (preferably) understand why each one is correct so you can 'recreate' the theorem.

⋆**Exercise 7.96.** Give a Θ bound for each of the following functions. You do not need to prove them.

(a) $f(n) = n^5 + n^3 + 1900 + n^7 + 21n + n^2$

(b) $f(n) = (n^2 + 23n + 19)(n^2 + 23n + n^3 + 19)n^3$ (Don't make this one harder than it is)

(c) $f(n) = n^2 + 10,000n + 100,000,000,000$

(d) $f(n) = 49 * 2^n + 34 * 3^n$

(e) $f(n) = 2^n + n^5 + n^3$

(f) $f(n) = n \log n + n^2$

(g) $f(n) = \log^{300} n + n^{.000001}$

(h) $f(n) = n! \log n + n^n + 3^n$

⋆**Exercise 7.97.** Rank the following functions in increasing rate of growth. Clearly indicate if two or more functions have the same growth rate. Assume the logs are base 2.

x, x^2, 2^x, 10000, $\log^{300} x$, x^5, $\log x$, $x^{\log 3}$, $x^{.000001}$, 3^x, $x\log(x)$, $\log(x^{300})$, $\log(2^x)$

7.3 Algorithm Analysis

The overall goal of this chapter is to deal with a seemingly simple question: *Given an algorithm, how good is it?* I say "seemingly" simple because unless we define what we mean by "good", we cannot answer the question. Do we mean how elegant it is? How easy it is to understand? How easy it is to update if/when necessary? Whether or not it can be generalized?

Although all of these may be important questions, in algorithm analysis we are usually more interested in the following two questions: *How long does the algorithm take to run, and how much space (memory) does the algorithm require.* In fact, we follow the tradition of most books and focus our discussion on the first question. This is usually reasonable since the amount of memory used by most algorithms is not large enough to matter. There are times, however, when analyzing the space required by an algorithm is important. For instance, when the data is really large (e.g. the graph that represents friendships on Facebook) or when you are implementing a *space-time-tradeoff* algorithm.

Although we have simplified the question, we still need to be more specific. What do we mean by "time"? Do we mean how long it takes in real time (often called *wall-clock time*)? Or the actual amount of time our processor used (called *CPU time*)? Or the exact *number of instructions* (or *number of operations*) executed?

⋆**Question 7.98.** Why aren't wall-clock time and CPU time the same?

Answer ____

____

____

Because the running time of an algorithm is greatly affected by the characteristics of the computer system (e.g. processor speed, number of processors, amount of memory, file-system type, etc.), the running time does not necessarily provide a comparable measure, regardless of whether you use CPU time or wall-clock time. The next question asks you to think about why.

⋆**Question 7.99.** Sue and Stu were competing to write the fastest algorithm to solve a problem. After a week, Sue informs Stu that her program took 1 hour to run. Stu declared himself victorious since his program took only 3 minutes. But the real question is this: Who's algorithm was more efficient? Can we be certain Stu's algorithm was better than Sue's? Explain. (Hint: Make sure you don't jump to any conclusion too quickly. Think about all of the possibilities.)

Answer ____

____

____

The answer to the previous question should make it clear that you cannot compare the running times of algorithms if they were run on different machines. Even if two algorithms are run on the same computer, the wall-clock times may not be comparable.

⋆**Question 7.100.** Why isn't the *wall-clock time* of two algorithms that are run on the same computer always a reliable indicator of their relative performances?

Answer ______________________________

____

____

In fact, if you run the same algorithm on the same machine multiple times, it will not always take the same amount of time. Sometimes the differences between trial runs can be significant.

⋆**Question 7.101.** If two algorithms are run on the same machine, can we reliably compare the *CPU-times*?

Answer ______________________________

____

____

So the CPU-time turns out to be a pretty good measure of algorithm performance. Unfortunately, it does not really allow one to compare two *algorithms*. It only allows us to compare specific *implementations of the algorithms*. It also requires us to implement the algorithm in an actual programming language before we even know how good the algorithm is (that is, before we know if we should even spend the time to implement it).

But we can analyze and compare algorithms before they are implemented if we use the *number of instructions* as our measure of performance. There is still a problem with this measure. What is meant by an "instruction"? When you write a program in a language such as Java or C++, it is not executed exactly as you wrote it. It is compiled into some sort of machine language. The process of compiling does not generally involve a one-to-one mapping of instructions, so counting Java instructions versus C++ instructions wouldn't necessarily be fair. On the other hand, we certainly do not want to look at the machine code in order to count instructions—machine code is ugly. Further, when analyzing an algorithm, should we even take into account the exact implementation in a particular language, or should we analyze the algorithm apart from implementation?

O.K., that's enough of the complications. Let's get to the bottom line. When analyzing algorithms, we generally want to ignore what sort of machine it will run on and what language it will be implemented in. We also generally do not want to know *exactly* how many instructions it will take. Instead, we want to know the *rate of growth* of the number of instructions. This is sometimes called the *asymptotic running time* of an algorithm. In other words, as the size of the input increases, how does that affect the number of instructions executed? We will typically use the notation from Section 7.1 to specify the running time of an algorithm. We will call this the *time complexity* (or often just *complexity*) of the algorithm.

7.3.1 Analyzing Algorithms

Given an algorithm, the *size of the input* is exactly what it sounds like—the amount of space required to specify the input. For instance, if an algorithm operates on an array of size n, we generally say the input is of size n. For a graph, it is usually the number of vertices or the number of vertices and edges. When the input is a single number, things get more complicated for reasons I do not want to get into right now. We usually don't need to worry about this, though.

Algorithm analysis involves determining the size of the input, n, and then finding a function based on n that tells us how long the algorithm will take if the input is of size n. By "how long" we of course mean how many operations.

Example 7.102 (Sequential Search)**.** Given an array of n elements, often one needs to determine if a given number *val* is in the array. One way to do this is with the *sequential search* algorithm that simply looks through all of the elements in the array until it finds it or reaches the end. The most common version of this algorithm returns the index of the element, or -1 if the element is not in the array. Here is one implementation.

```
int sequentialSearch(int a[],int n, int val) {
    for(int i=0;i<a.size();i++) {
        if(a[i]==val) {
            return i;
        }
    }
    return -1;
}
```

What is the size of the input to this algorithm?

Solution: There are a few possible answers to this question. The input technically consists of an array of n elements, the numbers n, and the value we are searching for. So we could consider the size of the input to be $n + 2$. However, typically we ignore constants with input sizes. So we will say the size of the input is n.

In general, if an algorithm takes as input an array of size n and some constant number of other numeric parameters, we will consider the size of the input to be n.

⋆**Exercise 7.103.** Consider an algorithm that takes as input an n by m matrix, an integer v, and a real number r. What is the size of the input?

Answer ______________________________

Example 7.104. How many operations does `sequentialSearch` take on an array of size n?

Solution: As mentioned above, we consider n as the size of the input. Assigning $i = 0$ takes one instruction. Each iteration through the `for` loop increments i, compares i with $a.size()$, and compares $a[i]$ with val. Don't forget that accessing $a[i]$ and calling `a.size()` each take (at least) one instruction. Finally, it takes an instruction to return the value. If the val is in the array at position k, the algorithm will take $2 + 5k = \Theta(k)$ operations, the 2 coming from the assignment `i=0` and the return statement. If val is not in the array, the algorithm takes $2 + 5n = \Theta(n)$ instructions.

This last example should bring up a few questions. Did we miss any instructions? Did we miss any possible outcomes that would give us a different answer? How exactly should we specify our analysis?

Let's deal with the possible outcomes question first. Generally speaking, when we analyze an algorithm we want to know what happens in one of three cases: The best case, the average case, or the worst case. When thinking about these cases, we always consider them for a given value of n (the input size). We will see in a moment why this matters.

As the name suggests, when performing a *best case* analysis, we are trying to determine the smallest possible number of instructions an algorithm will take. Typically, this is the least useful type of analysis. If you have experienced a situation when someone said something like "it will only take an hour (or a day) to fix your cell phone," and it actually took 3 hours (or days), you will understand why.

When determining the best-case performance of an algorithm, remember that we need to determine the best-case performance for a given input size n. This is important since otherwise every algorithm would take a constant amount of time in the best case simply by giving it an input of the smallest possible size (typically 0 or 1). That sort of analysis is not very informative.

Note: *When you are asked to do a best-case analysis of an algorithm, remember that it is implied that what is being asked is the best-case analysis* **for an input of size** n. *This actually applies to average and worst-case analysis as well, but it is easier to make this mistake when doing a best-case analysis.*

Worst case analysis considers what is the largest number of instructions that will execute (again, for a given input size n). This is probably the most common analysis, and typically the most useful. When you pay Amazon for guaranteed 2-day delivery, you are paying for them to guarantee a worst-case delivery time. However, this analogy is imperfect. When you do a worst-case analysis, you know the algorithm will *never* take longer than what your analysis specified, but occasionally an Amazon delivery is lost or delayed. When you perform a worst-case analysis of an algorithm, you always consider what can happen that will make an algorithm take as long as possible, so it will never take longer than the worst-case analysis implies.

The *average case* is a little more complicated, both to define and to compute. The first problem is determining what "average" means for a particular input and/or algorithm. For instance, what does an "average" array of values look like? The second problem is that even with a good definition, computing the average case complexity is usually much more difficult than the other two. It also must be used appropriately. If you know what the average number of instructions for an algorithm is, you need to remember that sometimes it might take less time and sometimes it might take more time–possibly significantly more time.

I sometimes use the term *expected running time* instead of one of these three. This is almost synonymous with *average case*, but when I use this term I am being less formal. Thus, I will not necessarily do a complete average case analysis to determine what I call the expected running time. Think of it as being how long an algorithm will *usually* take. It will typically coincide with either the average- or worst-case complexity.

Example 7.105. Continuing the `sequentialSearch` example, notice that our analysis above reveals that the best-case performance is $7 = \Theta(1)$ operations (if the element sought is the first one in the array) and the worst-case performance is $2 + 5n = \Theta(n)$ operations (if the element is not in the array). If we assume that the element we are searching for is equally likely to be anywhere in the array or not in the array, then the average-case performance should be about $2 + 5(n/2) = \Theta(n)$ operations. We will do a more thorough average-case analysis of this algorithm shortly.

Notice that in the previous example, the average- and worst-case complexities are the same. This makes sense. We estimate that the average case takes about half as long as the worst case. But no matter how large n gets, it is still just half as long. That is, the *rate of growth* of the average and worst-case running times are the same. Also note the logic we used to obtain the best-case complexity of $\Theta(1)$. We did not say the best case was $\Theta(1)$ because the best-case input was an array of size one. Instead it is $\Theta(1)$ because in the best case the element we are searching for is the first element of the array, no matter how large the array is.

Here is another important question: How do we know we counted all of the operations? As it turns out, we don't actually care. This is good because determining the exact number is very difficult, if not impossible. Recall that we said we wanted to know the rate of growth of an algorithm, not the exact number of instructions. As long as we count all of the "important" ones, we will get the correct rate of growth. But what are the "important" ones? The term *abstract operation* is sometimes used to describe the operations that we will count. Typically you choose one type of operation or a set of operations that you know will be performed the most often and consider those as the abstract operation(s).

Example 7.106. The analysis of `sequentialSearch` can be done more easily than in the previous example. We repeat the algorithm here for convenience.

```
int sequentialSearch(int a[],int n, int val) {
    for(int i=0;i<a.size();i++) {
        if(a[i]==val) {
            return i;
        }
    }
    return -1;
}
```

Notice that the comparison (`a[i]==val`) is executed as often as any other instruction. Therefore if we count the number of times that instruction executes, we can use that to determine the rate of growth of the running time.

In the best case the comparison is executed once (if the element being searched for is the first one in the array), so the best-case complexity is $\Theta(1)$.

In the worst case the comparison is executed $n = \Theta(n)$ times (if the element being searched for is either at the end or not present in the array).

As before, we expect the average case to be about $n/2 = \Theta(n)$, although in the next example we will do a more complete analysis.

Notice that we obtained the same answers here as we did above when we tried to take into account every operation.

Example 7.107. Let's determine the average-case complexity of `sequentialSearch`. If we assume that the element is equally likely to be anywhere in the array, then there is a $1/n$ chance that it will be in any given spot. If it is in the first spot, the comparison executes once. If it is in the second spot, it executes twice. In general it takes k comparisons if it is in the kth spot. Since each possibility has a $1/n$ chance, the average expected search time is

$$\sum_{k=1}^{n} \frac{k}{n} = \frac{1}{n}\sum_{k=1}^{n} k = \frac{1}{n}\frac{n(n+1)}{2} = \frac{n+1}{2} = \Theta(n).$$

Our analysis simplified things a bit—we didn't take into account the possibility that the element was not in the array. To do so, let's assume the element searched for is equally likely to be anywhere in the array or not in the array. That is, there is now a $1/(n+1)$ chance that it will be in any of the n spots in the array and a $1/(n+1)$ chance that it is not in the array. (We divide by $n+1$ because there are now $n+1$ possibilities, each equally likely.) If it is not in the array, the number of comparisons is n. In this case the expected time would be

$$\sum_{k=1}^{n}\left(\frac{k}{n+1}\right)+\frac{n}{n+1}=\frac{1}{n+1}\left(\sum_{k=1}^{n}k+n\right)=\frac{1}{n+1}\left(\frac{n(n+1)}{2}+n\right)=\frac{n^2+3n}{2(n+1)}=\Theta(n).$$

We'll leave it to you to prove that $\frac{n^2+3n}{2(n+1)}=\Theta(n)$. (Use Theorem 7.50 and a little algebra).

The previous example demonstrates how performing an average-case analysis is typically much more difficult than the other two, even with a relatively simple algorithm. In fact, did we even do it correctly? Is it a valid assumption that there is a $1/(n+1)$ chance that the element searched for is not in the array? If we are searching for a lot of values in a small array, perhaps it is the case that most of the values we are searching for are *not* in the array. Maybe it is more realistic to assume there is a 50% chance it is in the array and 50% chance that it is not in the array. I could propose several other reasonable assumptions, too. As stated before, it can be difficult to define "average." In this case it actually doesn't matter a whole lot because under any reasonable assumptions the average-case analysis will always come out as $\Theta(n)$.

As you might be able to imagine, things get much more complicated as the algorithms get more complex. This is one of the reasons that in some cases we will skip or gloss over the details of the average-case analysis of an algorithm.

It is important to make sure that you choose the operation(s) you will count carefully so your analysis is correct. In addition, you need to look at every instruction in the algorithm to determine whether or not it can be accomplished in constant time. If some step takes longer than constant time, that needs to be properly taken into consideration. In particular, consider function/method calls and operations on data structures very carefully. For instance, if you see a method call like `insert(x)` or `get(x)`, you cannot just assume they take constant time. You need to determine how much time they actually take.

Note: *When you are asked for the* complexity *of an algorithm, you should do the following three things:*

1. *Give the best, average, and worst-case complexities unless otherwise specified. Sometimes the average case is quite complicated and can be skipped.*

2. *Give answers in the form of $\Theta(f(n))$ for some function $f(n)$, or $O(f(n))$ if a tight bound is not possible. The function $f(n)$ you choose should be as simple as possible. For instance, instead of $\Theta(3n^2+2n+89)$, you should use $\Theta(n^2)$ since the constants and lower order terms don't matter.*

3. *Clearly justify your answers by explaining how you arrived at them in sufficient detail.*

Example 7.108. What is the complexity of `max(x,y)`? Justify your answers.

```
int max(int x, int y) {
    if(x >= y) {
        return x;
    } else {
        return y;
    }
}
```

Solution: No matter what, the algorithm does a single comparison followed by a return statement. Therefore, in the best, average, and worst case, `max` takes about 2 operations. Therefore, the complexity is always $\Theta(1)$ (otherwise known as *constant*).

⋆**Exercise 7.109.** Analyze the following algorithm that finds the maximum value in an array. Start by deciding which operation(s) should be counted. Don't forget to give the best, worst, and average-case complexities.

```
int maximum(int a[],int n) {
    int max = int.MIN_VAL;
    for (int i=0; i<n; i++)
        max = max(max, a[i]);
    return max;
}
```

When an algorithm has no conditional statements (like the `maximum` algorithm from the previous exercise), or at least none that can cause the algorithm to end earlier, the best, average, and worst-case complexities will usually be the same. I say usually because there is always the possibility of a weird algorithm that I haven't thought of that could be an exception.

Example 7.110. Give the complexity of the following code.

```
int q=0;
for (int i=1; i<=n; i++) {
    q=q+i*i;
}
for (int j=1; j<=n; j++) {
    q=q*j;
}
```

Solution: This algorithm has two independent loops, each of which do slightly different things. Thus, we cannot pick a single operation to count. Instead we will pick the assignment statements that involve q. That is, we will use both `q=q+i*i` and `q=q*j`. The first assignment executes n times since the first loop executes for every value of i from 1 to n. The second loop also executes its assignment n times for the same reason. Since the loops happen one after another, we add the number of operations, so the total is $n + n = 2n$ assignment statements. Since there are no conditional statements, this is the best, worst, and average-case number of assignment statements. Thus, the complexity for all three cases is $\Theta(n)$.

Example 7.111. Give the complexity of the following code.

```
double V = 0;
for (int i=1; i<=n; i++) {
    for (int j=1; j<=n; j++) {
        V=V+A[i]*A[j];
    }
}
```

Solution: Clearly the assignment (`V=A[i]*A[j]`) occurs the most often. The inner loop[a] always executes n times, each time doing one assignment. The outer loop executes n times, and each time it executes, it executes the inner loop. Therefore the total time is $n \cdot n = \Theta(n^2)$. This is the best, worst, and average case complexity since nothing about the input can change what the algorithm does.

Here is another way to think about it. The inner loop executes the assignment statement n times every time it executes. The first time through the outer loop, the whole inner loop executes an calls the assignment n times. The second time through the outer loop, the whole inner loop executes an calls the assignment n times. This happens all the way until the nth time through the outer loop during which the whole inner loop executes an calls the assignment n times. Thus, the total number of times the assignment is called is $n+n+\cdots+n$ times (where there are n terms in the sum), which is just $n \cdot n$. Thus the complexity is $\Theta(n^2)$.

[a] *Always* analyze from the inside out. The more practice you get, the more it will be obvious that this is the only way that will consistently work.

Sometimes people mistakingly think the algorithm Example 7.110 takes $\Theta(n^2)$ operations. But it is not executing one loop inside another loop. It is executing one loop n times followed by another loop n times. On the other hand, the algorithm in Example 7.111 does *not* take $n + n$

operations. It is not executing one loop n times followed by another loop n times. It is executing one loop n times, and each of those n times it is executing another loop that takes n time.

Here is an analogy. If you climb a flight of 10 stairs followed by another flight of 10 stairs, you climbed a total of $10 + 10 = 20$ stairs. Now assume you go into a building that has 10 floors. There are 10 steps between floors (so it takes 10 steps to get from floor 1 to 2, etc.) If you climb to the top of the building, how many stairs did you climb? It is $10 + 10 + \cdots + 10$ (where there are 10 terms in the sum), which is $100 = 10^2$. How does this relate to the previous examples? Simple. In the first case, you executed:

```
for(stair 1 through 10)
    climb stair
for(stair 1 through 10)
    climb stair
```

and in the second case you executed:

```
for(floors 1 through 10)
    for(stair 1 through 10)
        climb stair
```

Do you see the resemblance to the code from Examples 7.110 and 7.111? And do you see how we are really performing the same analysis?

It is important to be careful not to jump to conclusions when analyzing algorithms. For instance, a double-nested for-loop should always take $\Theta(n^2)$ to execute, right?

⋆**Exercise 7.112.** What is the worst-case complexity of the following algorithm?

```
int k=50;
for (i = 0; i < n; i ++) {
    for (j = 0; j < k; j ++) {
        a[i][j] = b[i][j] * x;
    }
}
```

If you read the solution to the previous exercise (which you definitely should have—*always* read the solutions!), you will see that you need to be careful not to jump to conclusions too quickly. A double-nested loop does not always mean an algorithm takes $\Theta(n^2)$ time. But does it guarantee it will take $O(n^2)$ (in other words, no more than quadratic time)?

⋆**Exercise 7.113.** What is the worst-case complexity of the following algorithm?

```
for (i = 0; i < n; i ++) {
    for (j = 0; j < n*n; j ++) {
        a[i][j] = b[i][j] * x;
    }
}
```

7.3.2 Common Time Complexities

We have already discussed the relative growth rates of functions. In this section we apply that understanding to the analysis of algorithms. That is, we will discuss common time complexities that are encountered when analyzing algorithms. Let n be the size of the input and k a constant. We will briefly discuss each of the following complexity classes, which are listed (mostly) in order of rate of growth.

- Constant: $\Theta(k)$, for example $\Theta(1)$
- Logarithmic: $\Theta(\log_k n)$
- Linear: $\Theta(n)$
- $n \log n$: $\Theta(n \log_k n)$
- Quadratic: $\Theta(n^2)$
- Polynomial: $\Theta(n^k)$
- Exponential: $\Theta(k^n)$

Definition 7.114 (Constant)**.** *An algorithm with running time $\Theta(1)$ (or $\Theta(k)$ for some constant k) is said to have* **constant** *complexity. Note that this does not necessarily mean that the algorithm takes exactly the same amount of time for all inputs, but it* **does** *mean that there is some number K such that it always takes no more than K operations.*

Example 7.115. The following algorithms have constant complexity.

```
int FifthElement(int A[],int n)
{
    return A[4];
}
```

```
int PartialSum(int A[],int n) {
    int sum=0;
    for(int i=0;i<42;i++)
       sum=sum+A[i];
    return sum;
}
```

The algorithm `FifthElement` just indexes into an array and returns that value. Since array indexing takes constant time, as does returning a single value, this algorithm clearly takes just constant time, no matter how large n is.

The algorithm `PartialSum` might seem to take $O(n)$ time since it contains a loop. But don't jump to conclusions too quickly. Notice that the loop executes 42 times, regardless of how large n might be. All of the other operations (both in and out of the loop) takes constant time. Thus, the overall complexity is something like $c_1 + 42 * c_2$, where c_1 is the time it takes to do the operations outside the loop, and c_2 is the time it takes to execute the code in the loop each time it executes, including the comparison and increment in the for loop itself. Since both c_1 and c_2 are constant, so is $c_1 + 42 * c_2$. Thus, the algorithm takes constant time.

⋆**Exercise 7.116.** Which of the following algorithms have constant complexity? Briefly justify your answers.

(a) The `AreaTrapezoid` algorithm from Example 3.1.

Answer ____

(b) The `factorial` algorithm from Example 3.40.

Answer ____

(c) The `absoluteValue` algorithm from Problem 3.12.

Answer ____

Definition 7.117 (Logarithmic)**.** *Algorithms with running time $\Theta(\log n)$ are said to have* **logarithmic** *complexity. As the input size n increases, so does the running time, but very slowly. Logarithmic algorithms are typically found when the algorithm can systematically ignore fractions of the input.*

Example 7.118. In Example 7.161 we will see that *binary search* has complexity $\Theta(\log n)$.

Definition 7.119 (Linear)**.** *Algorithms with running time $\Theta(n)$ are said to have* **linear** *complexity. As n increases, the run time increases in proportion with n. Linear algorithms access each of their n inputs at most some constant number of times.*

Example 7.120. The following are linear algorithms.

```
void sumFirstN(int n) {
    int sum=0;
    for (int i=1;i<=n;i++)
        sum = sum + i;
}
```

```
void mSumFirstN(int n) {
    int sum=0;
    for(int i=1;i<=n;i++)
        for(int k=1;k<7;k++)
            sum = sum + i;
}
```

It is pretty easy to see that `sumFirstN` takes linear time since it contains a single for loop that executes n times and does a constant amount of work each time.

At first glance it may seem that `mSumFirstN` takes $\Theta(n^2)$ time since it has a double nested loop. You will think about why it is actually $\Theta(n)$ in the next question.

⋆**Question 7.121.** Why is the complexity of `mSumFirstN` from the previous example $\Theta(n)$ and not $\Theta(n^2)$?

Answer ____

____

____

Definition 7.122 ($n \log n$)**.** *Many divide-and-conquer algorithms have complexity $\Theta(n \log n)$. These algorithms break the input into a constant number of subproblems of the same type, solve them independently, and then combine the solutions together. Not all divide-and-conquer algorithms have this complexity, however.*

Example 7.123. Two of the most well known sorting algorithms, *Quicksort* and *Mergesort*, have an average case complexity of $\Theta(n \log n)$. We will do a complete analysis of both algorithms in Chapter 8.

Definition 7.124 (Quadratic)**.** *Algorithms with a running time of $\Theta(n^2)$ are said to have* **quadratic** *complexity. As n doubles, the running time quadruples.*

Example 7.125. The following algorithm is quadratic.

```
int compute_sums(int A[], int n) {
    int M[n][n];
    for (int i=0;i<n;i++)
        for (int j=0;j<n;j++)
            M[i][j]=A[i]+A[j];
    return M;
}
```

This one is pretty easy to see since it has double nested loops that each execute n times, and the amount of work done in the inner loop is constant.

⋆**Exercise 7.126.** Which of the following algorithms have quadratic complexity? Briefly justify your answers.

(a) The `factorial` algorithm from Example 3.40.

Answer ______________________________

____

(b) An algorithm that tries to find the smallest element in an array of size $n \times n$ by searching through the entire array.

Answer ______________________________

____

⋆**Question 7.127.** In a previous course you may have encountered several quadratic sorting algorithms. Name them. (Note: We will analyze two of them soon.)

Answer ______________________________

____

Definition 7.128 (Polynomial)**.** *Algorithms with running time $\Theta(n^k)$ for some constant k are said to have* **polynomial** *complexity. We call them* **polynomial-time algorithms**. *Note that linear and quadratic are special cases of polynomial. When we say an* **efficient** *algorithm exists to solve a problem, we typically mean a polynomial-time algorithm.*

Example 7.129. As we will see in Example 7.150, `MatrixMultiply` takes $\Theta(n^3)$ time. Since 3 is a constant, that is a polynomial-time algorithm. We will also mention Strassen's algorithm that has a complexity of about $\Theta(n^{2.8})$. That is also a polynomial-time algorithm. It's actual complexity is $\Theta(n^{\log_2 7})$.

Definition 7.130 (Exponential)**.** *Algorithms with running time $\Theta(k^n)$ for some constant k are said to have* **exponential** *complexity. Since exponential algorithms can only be run for small values of n, they are not considered to be efficient. Brute-force algorithms are often exponential.*

Example 7.131. Since there are 2^n binary numbers of length n, an algorithm that lists all binary numbers of length n would take $\Theta(2^n)$ time, which is exponential.

Note: *As we have already seen, exponentials with different bases do not grow at the same rate. Thus, two exponential algorithms do not belong to the same complexity class unless the base of the exponent is the same. In other words, $a^n \neq \Theta(b^n)$ unless $a = b$.*

Let me end on a very important note regarding analysis of algorithms and asymptotic growth of functions. If algorithm A is faster than algorithm B, then the running time of A is less than the running time of B. On the other hand, if A's running time is asymptotically faster than the running time of B, that means B is a faster algorithm! In other words, the words fast/slow need to be reversed when discussing algorithm speeds versus the growth of the functions. Put simply: *A faster growing complexity means a slower algorithm, and vice-versa.*

7.3.3 Basic Sorting Algorithms

In this section we will analyze algorithms that are slightly more complex than the previous examples.

Example 7.132. Find the complexity of `bubblesort`, where n is the size of the array a.

```
void bubblesort(int a[],int n) {
    for(int i=n-1;i>0;i--) {
        for(int j=0;j<i;j++) {
            if(a[j] > a[j+1]) {
                swap(a,j,j+1);
            }
        }
    }
}
```

Solution: First, notice that the input size is n since we are sorting an array with n elements.

Example 3.46 gives an implementation of `swap` that takes constant time (verify this!). The conditional statement, including the swap, takes constant time (we'll call it c, as usual), regardless of whether or not the condition is true. It takes longer if the condition is true, but it is constant either way—about 3 operations (array indexing ($\times 2$) and comparison) versus about 6 (the swap adds about 3).

The inner loop goes from $j = 0$ to $j = i - 1$, so it executes i times and takes ci time. But what is i? This is where things get a little more complicated than in the previous examples. Notice that the outer loop is changing the value of i. We need to look at this a little more carefully.

1. The first time through the outer loop $i = n-1$. So the inner loop takes $c(n-1)$ time.
2. The second time through the outer loop $i = n-2$. So the inner loop takes $c(n-2)$ time.
3. The kth time through the outer loop $i = n-k$. So the inner loop takes $c(n-k)$ time.
4. This goes all the way to the nth time through the outer loop when $i = 1$ and the inner loop takes $c \cdot 1$ time.

The outer loop is simply causing the inner loop to be executed over and over again, but with different parameters (specifically, it is changing the limit on the inner loop). Thus, we need to add up the time taken for all of these calls to the inner loop. Doing so, we see that the total time required for `bubblesort` is

$$\begin{aligned}
c(n-1)+c(n-2)+c(n-3)+\cdots+c\,1 &= c((n-1)+(n-2)+(n-3)+\cdots+1) \\
&= c(1+2+3+\cdots+(n-1)) \\
&= c\sum_{k=1}^{n-1} k \\
&= c\frac{(n-1)n}{2} \\
&= \Theta(n^2)
\end{aligned}$$

Thus, the complexity (worst, best, and average) of `bubblesort` is $\Theta(n^2)$.

Note: *Part way through our analysis of* `bubblesort` *we had k as part of our complexity. But notice that the k did not show up as part of the final complexity. This is because in the context of the entire algorithm, k has no meaning. It is a local variable from the algorithm that we needed to use to determine the overall complexity of the algorithm.* **The only variables that should appear in the complexity of an algorithm are those that are related to the size of the input.**

$\star$**Question 7.133.** In the best case, the code in the conditional statement in `bubblesort` never executes. Why does this still result in a complexity of $\Theta(n^2)$?

Answer ______________________________

____

____

In reality, the best and worst case performance of `bubblesort` are different—the worst case is about twice as many operations. But when we are discussing the *complexity* of algorithms, we care about the asymptotic behavior—that is, what happens as n gets larger. In that case, the difference is still just a factor of 2. The best and worst-case complexities have the same growth rate (quadratic).

Consider how this is different if the best-case complexity of an algorithm is $\Theta(n)$ and the worst-case complexity is $\Theta(n^2)$. As n gets larger, the gap between the performance in the best and worst cases also gets larger. In this case, the best and worst-case complexities are not the same since one is linear and the other is quadratic.

Note: *If an algorithm contains nested loops and the limit on one or more of the inner loops depends on a variable from an outer loop, analyzing the algorithm will generally involve one or more summations, as it did with the previous example. As mentioned previously, variables related to those loops that are used in your analysis (e.g. i, j, k, etc.) should never show up in your final answer! They have no meaning in that context.*

Example 7.134. Find the complexity of `insertionSort`, where n is the size of the array a.

```
void insertionSort(int a[], int n) {
    for (int i=1;i<n;i++) {
        int v=a[i];
        int j=i-1;
        while (j >= 0 && a[j] > v) {
            a[j+1] = a[j];
            j--;
        }
        a[j+1]=v;
    }
}
```

Solution: The code inside the while loop takes constant time. The loop can end for one of two reasons—if j gets to 0, or if `a[j]>v`. In the worst case, it goes until $j = 0$. Since j starts out being i at the beginning, and it is decremented in the loop, that means the loop executes i times in the worst case.

The for loop (the outer loop) changes the value of i from 1 to $n-1$, executing a constant amount of code plus the while loop each time. So the ith time through the outer loop takes c_1+c_2i operations. We will simplify this to just i operations—you can think of it as counting the number of assignments in the while loop if you wish. So the worst-case complexity is

$$\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} = \Theta(n^2).$$

This happens, by the way, if the elements in the array start out in reverse order.

In the best case, the loop only executes once each time because `a[j]>v` is always true (which happens if the array is already sorted). In this case, the complexity is $\Theta(n)$ since the outer loop executes $n-1$ times, each time doing a constant amount of work.

We should point out that if we had done our computations using $c_1 + c_2i$ instead of i we would have arrived at the same answer, but it would have been more work:

$$\sum_{i=1}^{n-1} c_1 + c_2 i = \sum_{i=1}^{n-1} c_1 + \sum_{i=1}^{n-1} c_2 i = c_1 \cdot (n-1) + c_2 \sum_{i=1}^{n-1} i = c_1 \cdot (n-1) + c_2 \frac{(n-1)n}{2} = \Theta(n^2).$$

The advantage of including the constants is that we can stop short of the final step and get a better estimate of the actual number of operations used by the algorithm. In other words, if we want an exact answer, we need to include the constants and lower order terms. If we just want a bound, the constants and lower order terms can often be ignored.

Note: *There are rare cases when ignoring constants and lower order terms can cause trouble (meaning that it can lead to an incorrect answer) for subtle reasons that are beyond the scope of this book. Unless you take more advanced courses dealing with these topics, you most likely won't run into those problems.*

Let's complicate things a bit by bringing in some basic data structures.

Example 7.135. Consider the following implementation of insertion sort that works on lists of integers (written using Java syntax).

```
void insertionSort(List<Integer> A, int n) {
    for (int i = 1; i < n; i++) {
        Integer T = A.get(i);
        int j = i-1;
        while (j >= 0 && A.get(j).compareTo(T) > 0) {
            A.set(j + 1, A.get(j));
            j--;
        }
        A.set(j+1, T);
    }
}
```

Note: In some languages the size of the list, n, does not need to be passed in since it can be obtained with a method call (e.g. `A.size()`*). We will pass it in just to be clear that the size of the list is n.*

Give the complexity of `insertionSort` assuming that the list is an array-based implementation (e.g. an ArrayList in Java). To be clear, this means that both `get` and `set` take constant time. We also assume that comparing two integers (with `compareTo`) takes constant time (which should be true for any reasonable implementation).

Solution: Notice that this algorithm is almost identical to the earlier version that is implemented on an array. Array indexing and assignment are simply replaced with calls to `get` and `set`. Since the time of these calls remains constant, the earlier analysis still holds. Thus, the algorithm has a complexity of $\Theta(n^2)$.

The following should be review from a previous course.

⋆**Question 7.136.** What are the complexities of the methods `set(i,x)` (set the ith element of the the list to x) and `get(i)` (return the ith element of the list) for a *linked list*, assuming a reasonable implementation?

Answer ____

____

Example 7.137. Analyze the previous `insertionSort` algorithm assuming that the list is now a *linked list*.

Solution: The analysis here is a bit more complicated than we have previously seen, but we can still do it. We start by analyzing the code inside the `while` loop. In the worst case, each iteration of the loop makes two calls to `A.get(j)` and one call to `A.set(j+1)` and a constant amount of other work. The total time for each iteration is therefore about $2j + (j+1) + c = 3j + c'$ ($c' = c + 1$ is still just some constant). The index of the while loop starts at $j = i$ and can go until $j = 1$ (with j decrementing each iteration). Thus, the complexity of the while loop is about

$$\sum_{j=1}^{i}(3j + c') = 3\sum_{j=1}^{i} j + \sum_{j=1}^{i} c' = 3\frac{(i-1)i}{2} + ic' = \Theta(i^2)$$

The rest of the code inside the for loop takes constant time, except the one call to `get` and one call to `set` which take time $\Theta(i)$.[a] Thus, the code inside the for loop has complexity $\Theta(i^2 + i) = \Theta(i^2)$. The outer for loop makes i go from 1 to $n - 1$. Thus, the overall complexity is

$$\sum_{i=1}^{n-1}\Theta(i^2) = \Theta\left(\sum_{i=1}^{n-1} i^2\right) = \Theta\left(\frac{(n-1)n(2(n-1)+1)}{6}\right) = \Theta(n^3).$$

Clearly using a linked list in this implementation of insertion sort is a bad idea.

[a] Note that there is a tricky part here. It is subtle, but important. The call to *set* actually takes j as a parameter. However we cannot use $\Theta(j)$ as the complexity of this because the j has no meaning in this context. Therefore we use the fact that $1 \leq j \leq i$ to instead call it $\Theta(i)$.

7.3.4 Basic Data Structures (Review)

Now is a good time to review the complexities of operations on some common data structures (It is assumed that you have either learned this material before or that you can work them out based on your knowledge of how these data structures are implemented).

⋆**Exercise 7.138.** For each of the following implementations of a *stack*, give a tight bound (using Θ-notation, of course) on the expected running time of the given operations, assuming that the data structure has n items in it before the operation is performed.

Stack	array	linked list
push		
pop		
peek		
size		
isEmpty		

⋆**Exercise 7.139.** For each of the following implementations of a *queue*, give a tight bound on the expected running time of the given operations, assuming that the data structure has n items in it before the operation is performed.

Queue	array	linked list	circular array
enqueue			
dequeue			
first			
size			
isEmpty			

⋆**Exercise 7.140.** For each of the following implementations of a *list*, give a tight bound on the expected running time of the given operations, assuming that the data structure has n items in it before the operation is performed.

List	array	linked list
addToFront		
addToEnd		
removeFirst		
contains		
size		
isEmpty		

⋆**Exercise 7.141.** For each of the following implementations of a *binary search tree (BST)*, give a tight bound on the expected running time of the given operations, assuming that the data structure has n items in it before the operation is performed. Assume a linked implementations (rather than arrays). For balanced, assume an implementation like red-black tree or AVL tree.

BST	unbalanced	balanced
insert/add		
delete/remove		
search/contains		
maximum		
successor		

⋆**Exercise 7.142.** Give the average- and worst-case complexity of the following operations on a *hash table* (implemented with open-addressing or chaining–it doesn't matter), assuming that the data structure has n items in it before the operation is performed.

Hash Table	average	worst
insert/add		
delete/remove		
search/contains		

7.3.5 More Examples

Before presenting several more examples of algorithm analysis, let's summarize a few principles from the examples we have seen so far.

1. We can usually replace constants with 1. For instance, if something performs 30 operations, we can say it is constant and call it 1. This is only valid if it really is always 30, of course.
2. We can usually ignore lower-order terms. So if an algorithm takes $c_1 n + c_2$ operations, we can usually say that it takes n.
3. Nested loops must be treated with caution. If the limits in an inner loop change based on the outer loop, we generally need to write this as a summation.
4. We should generally work from the inside-out. Until you know how much time it takes to execute the code inside a loop, you cannot determine how much time the loop takes.
5. Function calls must be examined carefully. Do not assume that a function takes constant time unless you know that to be true. We already saw a few examples where function calls did *not* take constant time, and the next example will demonstrate it again.

6. Only the size of the input should appear as a variable in the complexity of an algorithm. If you have variables like i, j, or k in your complexity (because they were indexes of a loop, for instance), you should probably rethink your analysis of the algorithm. Loop variables should *never* appear in the complexity of an algorithm.

Now it's time to see if you can spot where someone didn't follow some of these principles.

⋆**Evaluate 7.143.** Consider the following code that computes $a^0 + a^1 + a^2 + \cdots + a^{n-1}$.

```
double addPowers(double a, int n) {
    if(a==1) {
        return n;
    } else {
        double sum = 0;
        for(int i=0;i<n;i++) {
            sum += power(a,i);
        }
        return sum;
    }
}
```

The function `power(a,i)` computes a^i, and takes i operations. Regard the input size as n. What is the worst-case complexity of `addPowers(a,n)`?

Solution 1: Since a^n is an exponential function, the complexity is $O(a^n)$.

Evaluation ____

____

____

Solution 2: The worst-case is ni since `power(a,i)` takes i time and the for loop executes n times.

Evaluation ____

____

____

Solution 3: The for loop executes n times. Each time it executes, it calls `power(a,i)`, which takes i time. In the worst case, $i = n - 1$, so the complexity is $(n-1)n = O(n^2)$.

Evaluation ____

____

____

⋆**Exercise 7.144.** What is the worst-case complexity of `addPowers` from Evaluate 7.143? Justify your answer.

⋆**Exercise 7.145.** Give an implementation of the `addPowers` algorithm that takes $\Theta(n)$ time. Justify the fact that it takes $\Theta(n)$ time. (Hint: Why compute a^5 (for instance) from scratch if you have already computed a^4?)

```
double addPowers(double a, int n) {

}
```

Justification of complexity:

⋆**Exercise 7.146.** Give an implementation of the `addPowers` algorithm that takes $\Theta(n)$ time *but does not use a loop*. Justify the fact that it takes $\Theta(n)$ time. (Hint: This solution should be much shorter than your previous one.)

```
double addPowers(double a, int n) {

}
```

Justification of complexity:

Example 7.147. A student turned in the code below (which does as its name suggests). I gave them a 'C' on the assignment because although it works, it is very inefficient. About how many operations does their implementation require?

```
int sumFromMToN(int m, int n) {
    int sum = 0;
    for(int i=1;i<=n;i++) {
        sum = sum + i;
    }
    for(int i=1;i<m;i++) {
        sum = sum - i;
    }
    return sum;
}
```

Solution: The first loop takes about $1 + 4n$ operations, and the second loop takes about $1 + 4(m - 1)$ operations. The first statement and return statement add 2 operations. So the total number of operations is about $4 + 4n + 4(m - 1) = 4(n + m) = \Theta(n + m)$.

⋆**Evaluate 7.148.** Write an 'A' version of the method from Example 7.147. You can assume that $1 \leq m \leq n$. *For each solution, determine how many operations are required and evaluate it based on that as well as whether or not it is correct.*

Solution 1:

```
int sumFromMToN(int m,int n) {
    int sum = 0;
    for(int i=0;i<n;i++) {
        sum = sum + i;
    }
    for(int i=0;i<m;i++) {
        sum = sum - i;
    }
    return sum;
}
```

Evaluation ____

Solution 2:

```
int sumFromMToN(int m,int n) {
    int sum = 0;
    for(int i=m;i<n;i++) {
        sum = sum + i;
    }
    return sum;
}
```

Evaluation ____

Solution 3:

```
int sumFromMToN(int m,int n) {
    return (n*(n-1)/2 - m(m-1)/2);
}
```

Evaluation ____

⋆**Exercise 7.149.** Write an 'A' version of the method from Example 7.147. You can assume that $1 \leq m \leq n$. Explain why your solution is correct and give its efficiency.

```
int sumFromMToN(int m,int n) {

}
```

Justification

Efficiency with justification

Example 7.150. The `MatrixMultiply` algorithm given below is the standard algorithm used to compute the product of two matrices. Find the worst-case complexity of `MatrixMultiply`. Assume that A and B are $n \times n$ matrices.

```
Matrix MatrixMultiply(Matrix A, Matrix B) {
    Matrix C;
    for(int i=0 ; i < n; i++) {
        for(int j=0 ; j < n ; j++) {
            C[i][j]=0;
            for(int k=0 ; k < n ; k++) {
                C[i][j] += A[i][k]*B[k][j];
            }
        }
    }
    return C;
}
```

Solution: The code inside the inner loop does array indexing, multiplication, addition, and assignment. All of these together take just constant time. Therefore, let's count the number of times the statement `C[i][j]+=A[i][k]*B[k][j]` executes. We will ignore the calls to `C[i][j]=0` since it executes just once every time the entire middle loop executes, so it has a negligible contribution. Similarly, the statement `C[i][j]+=A[i][k]*B[k][j]` is called at least as often as any of the code in the for loops (i.e. the comparisons and increments) so we will ignore that code as well. The bottom line is that if we count the number of times `C[i][j]+=A[i][k]*B[k][j]` executes, it will give us a tight bound on the complexity of `MatrixMultiply`.

The inner loop executes the statement n times. The middle loop executes n times, each time executing the inner loop (which executes the statement n times). Thus, the middle loop executes the statement $n \times n = n^2$ times. The outer loop simply executes the middle loop n times. Therefore the outer loop (and thus the whole algorithm) executes the statement $n \times n^2 = n^3$ times. Thus, the worst-case complexity of `MatrixMultiply` is $\Theta(n^3)$. Notice that this is also the best and average-case complexity since there are no conditional statements in this code.

Example 7.151. In Java, the ArrayList `retainAll` method is implemented as follows (this code is simplified a little from the actual implementation, but the changes do not affect the complexity of the code). Note that `Object[] elementData` and `int size` are fields of ArrayList whose meaning should be obvious.

```
public boolean retainAll(Collection<?> c) {
    boolean modified = false;
    int w = 0;
    for (int r = 0; r < size; r++) {
        if (c.contains(elementData[r])) {
            elementData[w++] = elementData[r];
        }
    }
    if (w != size) {
        for (int i = w; i < size; i++)
            elementData[i] = null;
        size = w;
        modified = true;
    }
    return modified;
}
```

Let `al1` be an ArrayList of size n.

(a) What is the complexity of `al1.retainAll(al2)`, where `al2` is an ArrayList with m elements?

> **Solution:** The method call `c.contains(elementData[r])` takes $\Theta(m)$ time since c is an ArrayList. The rest of the code in that for loop takes constant time. Since this is done inside a for loop that executes n times, the first half of the code takes $\Theta(nm)$ time. In the worst case ($w = 0$), the second half of the code takes $\Theta(n)$ time. Thus, the worst-case complexity of the method is $\Theta(nm + n) = \Theta(nm)$.

(b) What is the complexity of `al1.retainAll(ts2)`, where `ts2` is a TreeSet with m elements?

> **Solution:** The method call `c.contains(elementData[r])` takes $\Theta(\log m)$ time since c is a TreeSet. The rest of the code in that for loop takes constant

time. Since this is done inside a for loop that executes n times, the first half of the code takes $\Theta(n \log m)$ time. In the worst case ($w = 0$), the second half of the code takes $\Theta(n)$ time. Thus, the worst-case complexity of the method is $\Theta(n \log m + n) = \Theta(n \log m)$.

⋆**Exercise 7.152.** Answer the following two questions based on the code from Example 7.151.

(a) What is the complexity of `al1.retainAll(ll2)`, where `ll2` is a LinkedList with m elements? Answer ______________________________

____

(b) What is the complexity of `al1.retainAll(hs2)`, where `hs2` is a HashSet with m elements? Answer ______________________________

____

Example 7.153. In Java, the `retainAll` method is implemented as follows for `LinkedList`, `TreeSet`, and `HashSet`.

```
public boolean retainAll(Collection<?> c) {
    boolean modified = false;
    Iterator<E> iter = iterator();
    while (iter.hasNext()) {
        if (!c.contains(iter.next())) {
            iter.remove();
            modified = true;
        }
    }
    return modified;
}
```

Assume that the calls `iter.hasNext()` and `iter.next()` take constant time. Let `ts1` be a TreeSet of size n. Find the worst-case complexity of each of the following method calls.

(a) `ts1.retainAll(al2)`, where `al2` is an ArrayList of size m.

Solution: The call to `iter.remove()` takes $\Theta(\log n)$ time since the iterator is over a TreeSet. The call to `contains` takes $\Theta(m)$ time since in this case c is the ArrayList `al2`. The other operations in the loop take constant time. Thus, each iteration of the while loop takes $\Theta(\log n + m)$ time in the worst case (which occurs if the conditional statement is always true and `remove` is called every time). Since the loop executes n times, and the rest of the code takes constant time, the overall complexity is $\Theta(n(\log n + m))$.

(b) `ts1.retainAll(hs2)`, where `hs2` is an HashSet of size m.

Solution: The call to `iter.remove()` takes $\Theta(\log n)$ time. The call to `contains` takes $\Theta(1)$ time since c is the HashSet `hs2`. Thus, each iteration of the while loop takes $\Theta(\log n + 1)$ time and the overall complexity is therefore $\Theta(n(\log n + 1)) = \Theta(n \log n)$.

⋆**Exercise 7.154.** Using the setup and code from Example 7.153, determine the complexity of the following method calls.

(a) `ts1.retainAll(ll2)`, where `ll2` is a LinkedList of size m. Answer ____________

____

____

(b) `ts1.retainAll(ts2)`, where `ts2` is an TreeSet of size m. Answer ____________

____

____

It is important to note that the number of examples related to the `retainAll` method is not reflective of the importance of this method. It just turns out to be an interesting method to analyze the complexity of given different data structures.

We end this section with a comment that perhaps too few people think about. *Theory* and *practice* don't always agree. Since asymptotic notation ignores the *constants*, two algorithms that have the same complexity are not always equally good in practice. For instance, if one takes $4 \cdot n^2$ operations and the other $10{,}000 \cdot n^2$ operations, clearly the first will be preferred even though they are both $\Theta(n^2)$ algorithms.

As another example, consider matrix multiplication, which is used extensively in many scientific applications. As we saw, the standard algorithm has complexity $\Theta(n^3)$. Strassen's algorithm for matrix multiplication (the details of which are beyond the scope of this book) has complexity of about $\Theta(n^{2.8})$. Clearly, Strassen's algorithm is better asymptotically. In other words, if your matrices are large enough, Strassen's algorithm is certainly the better choice. However, it turns out that if $n = 50$, the standard algorithm performs better. There is debate about the "crossover point." This is the point at which the more efficient algorithm is worth using. For smaller inputs, the overhead associated with the cleverness of the algorithm isn't worth the extra time it takes. For larger inputs, the extra overhead is far outweighed by the benefits of the algorithm. For Strassen's algorithm, this point may be somewhere between 75 and 100, but don't quote me on that. The point is that for small enough matrices, the standard algorithm should be used. For matrices that are large enough, Strassen's algorithm should be used. Neither one is *always* better to use.

Analyzing recursive algorithms can be a little more complex. We will consider such algorithms in Chapter 8, where we develop the necessary tools.

7.3.6 Binary Search

Next we analyze the *binary search* algorithm. Before we can do that, we need to develop a few useful results that will make the proof much easier to understand. We start by trying to get you to understand how the binary representation of n and $\lfloor n/2 \rfloor$ are related to each other.

Example 7.155. How is the binary representation of a number n related to the binary representation of $\lfloor n/2 \rfloor$? Let's try some examples. If $n = 9$, $\lfloor n/2 \rfloor = 4$. Notice that the binary representation of 9 is 1001 and the binary representation of 4 is 100. If $n = 22$, $\lfloor n/2 \rfloor = 11$. The binary representation of 22 is 10110 and the binary representation of 11 is 1011. Is there a pattern here? This probably isn't enough data to be certain yet.

Let's see if you can find the pattern with just a few more data points.

⋆**Exercise 7.156.** Fill in the following table with the binary representations.

n		$\lfloor n/2 \rfloor$	
decimal	binary	decimal	binary
12		6	
13		6	
32		16	
33		16	
118		59	
119		59	

⋆**Question 7.157.** How are the binary representations of n and $\lfloor n/2 \rfloor$ related?

Answer ____

____

Hopefully you observed a clear pattern in the previous exercise. The next theorem formalizes this idea. We provide a proof of the theorem to make it clear what is going on.

Theorem 7.158. *The binary representation of $\lfloor n/2 \rfloor$ is the binary representation of n shifted to the right one bit. That is, the binary representation of $\lfloor n/2 \rfloor$ is the same as that of n with the last bit (the lowest order bit) chopped off.*

Proof: *Let the binary representation of n be $a_m a_{m-1} a_{m-2} \ldots a_2 a_1 a_0$, where $a_m = 1$ (so the highest order bit is a 1). Then*

$$n = a_m\, 2^m + a_{m-1}\, 2^{m-1} + \ldots + a_2\, 2^2 + a_1\, 2^1 + a_0\, 2^0.$$

From this we can see that

$$\begin{aligned}
\lfloor n/2 \rfloor &= \lfloor (a_m\, 2^m + a_{m-1}\, 2^{m-1} + \ldots + a_2\, 2^2 + a_1\, 2^1 + a_0\, 2^0)/2 \rfloor \\
&= \lfloor a_m\, 2^m/2 + a_{m-1}\, 2^{m-1}/2 + \ldots + a_2\, 2^2/2 + a_1\, 2^1/2 + a_0\, 2^0/2 \rfloor \\
&= \lfloor a_m\, 2^{m-1} + a_{m-1}\, 2^{m-2} + \ldots + a_2\, 2^1 + a_1\, 2^0 + a_0/2 \rfloor \\
&= a_m\, 2^{m-1} + a_{m-1}\, 2^{m-2} + \ldots + a_2\, 2^1 + a_1\, 2^0.
\end{aligned}$$

Notice that in the last step, $a_0/2$ is chopped off by the floor since it is either $0/2$ or $1/2$ and the other numbers are integers. From this we can see that the binary representation of $\lfloor n/2 \rfloor$ is $a_m a_{m-1} a_{m-2} \ldots a_2 a_1$, which is the binary representation of n shifted to the right one bit. □

Corollary 7.159. *If the number n requires exactly k bits to represent in binary, then $\lfloor n/2 \rfloor$ requires exactly $k-1$ bits to represent in binary.*

Proof: *According to Theorem 7.158, the binary representation of $\lfloor n/2 \rfloor$ is the binary representation of n shifted to the right one bit. Thus it is clear that $\lfloor n/2 \rfloor$ requires one less bit to represent.* □

We need just one more result.

Theorem 7.160. *It takes $\lfloor \log_2 n \rfloor + 1$ bits to represent n in binary.*

Proof: *Recall that $\log_c b$ is defined as "the number that c must be raised to in order to get b." That is, if $k = \log_c b$, then $c^k = b$. Also, it should be clear that 2^k is the smallest number that requires $k+1$ bits to represent in binary. (If you are not convinced of this, write out some binary numbers near powers of two until you see it.) Let k be the number such that*

$$2^{k-1} \le n < 2^k. \tag{7.4}$$

Since writing 2^{k-1} takes k bits and 2^k is the smallest number that requires $k+1$ bits, it should be clear that n requires exactly k bits to represent in binary. Taking the logarithm of equation 7.4, we get

$$\log_2 2^{k-1} \le \log_2 n < \log_2 2^k,$$

which leads to

$$k - 1 \le \log_2 n < k.$$

Clearly $\lfloor \log_2 n \rfloor = k-1$ since it is an integer. Thus, $k = \lfloor \log_2 n \rfloor + 1$, so it takes $\lfloor \log_2 n \rfloor + 1$ bits to represent n in binary. □

Now we are ready to analyze *binary search*.

Example 7.161. You are probably already familiar with the *binary search* algorithm. It is given here for reference.[a]

```
int binarySearch(int a[], int n, int val) {
   int left=0, right=n-1;
   while (right-left>=0) {
       int middle = (left+right)/2;
       if(val==a[middle])
           return middle;
       else if(val<a[middle])
           right=middle-1;
       else
           left=middle+1;
    }
   return -1;
}
```

Binary search finds the index of a value in a sorted array by comparing the value being searched for with the middle element of the array. If they are the same, it returns the index of the element. Otherwise it continues the search in only half of the array. In other words, it removes from consideration half of the array. Which half depends on whether the search value was greater than or less than the middle value.

We will show that *binary search* has worst-case complexity $\Theta(\log n)$. More precisely, we will prove that the while loop executes no more than $\lfloor \log_2 n \rfloor + 1$ times.

Proof: Since the code inside the while loop takes a constant amount of time, the complexity of *binary search* depends only on the number of iterations of the loop. Clearly the worst case is when a value is not in the array since otherwise the loop ends early with the `return` statement. Thus we will assume the value is not in the array.

Notice that the value `right-left` is the number of entries of the array that are still under consideration by the algorithm. The loop executes until `right-left` < 0. Before the first iteration, `right-left` $= n$. During each iteration, either `right` or `left` is set to the middle value between `right` and `left` (plus or minus 1). So after the first iteration, `right-left` $\leq \lfloor n/2 \rfloor$. In other words, the algorithm has discarded at least half of the entries of the array. During each subsequent iteration, `right-left` continues to be no more than the floor of half of its previous value, so the algorithm continues to discard half of the entries of the array each time through the loop.

According to Corollary 7.159, each iteration of the loop reduces the number of bits used to represent `right-left` by one. According to Theorem 7.160, it takes $\lfloor \log_2 n \rfloor + 1$ bits to represent n in binary, and $right - left$ started out as n. Therefore, after $\lfloor \log_2 n \rfloor$ iterations through the loop, `right-left` becomes 1, and the next iterations ensures that `right-left` becomes negative and the loop terminates (check this!). Since the loop executes at most $\lfloor \log_2 n \rfloor + 1$ times, the worst-case complexity of *binary search* is $\Theta(\log n)$. □

[a] You may be familiar with the recursive version of this algorithm instead of this iterative implementation. We analyze the iterative version because we have not yet covered recursion or the analysis of recursive algorithms.

7.4 Problems

Problem 7.1. Prove Theorem 7.18.

Problem 7.2. Θ can be thought of as a relation on the set of positive functions, where $(f, g) \in \Theta$ iff $f(n) = \Theta(g(n))$. Prove that Θ is an equivalence relation.

Problem 7.3. Rank the following functions in increasing rate of growth. Indicate if two or more functions have the same growth rate.

$x!,\ x^3,\ x^2 \log x,\ x,\ x^{\log_2 3},\ \sqrt{x},\ 3^x,\ x \log x,\ x^2,\ x^x,\ x^{3/2},\ x^{\log_3 7},\ x \log(x^2),\ x \log(\log(x)),\ \left(\frac{3}{2}\right)^x$

Problem 7.4. Prove that $3n^3 - 4n^2 + 13n = O(n^3)$

(a) Using the definition of O.

(b) Using limits.

Problem 7.5. Prove that $5n^2 - 7n = \Theta(n^2)$

(a) Using the definition of Θ and/or Theorem 7.18.

(b) Using limits.

Problem 7.6. Prove that $n \log n = o(n^2)$.

Problem 7.7. Prove that $\log(x^2 + x) = \Theta(\log x)$.

Problem 7.8. Prove that $\sqrt{5x^2 + 11x} = \Theta(x)$.

Problem 7.9. Prove that $n^2 = o(1.01^n)$.

Problem 7.10. Give tight bounds for the **best** and **worst** case running times of each of the following in terms of the size of the input.

(a)
```
void foo1(int n) {
    int foo = 0;
    for(int i = 0 ; i < n ; i++)
      foo += i;
}
```

(b)
```
void blah(int n) {
    int blah = 0;
    for(int i = 0 ; i < sqrt(n) ; i++)
      blah += i;
}
```

(c)
```
void ferzle1(int a[], int n) {
    int ferzle = 0;
    for(int i = 0 ; i < n ; i++) {
        for(int j = 0 ; j < n ; j++) {
            ferzle += a[i]*a[j];
            if(ferzle==10000) {
                j=n;
            }
        }
    }
}
```

(d)
```
void ferzle2(int n) {
    int ferzle = 0;
    for(int i = 0 ; i < n ; i++) {
        for(int j = i ; j < n ; j++) {
            ferzle += i*j;
        }
    }
}
```

(e)
```
void ferzle3(int a[], int n) {
    int ferzle = 0;
    for(int i = 0 ; i < n ; i++) {
        for(int j = 0 ; j < n ; j++) {
            ferzle += a[i]*a[j];
            if(ferzle==10000) {
                i=n;
            }
        }
    }
}
```

(f)
```
void ferzle4(int a[], int n) {
    int ferzle = 0;
    for(int i = 0 ; i < n ; i++) {
        for(int j = 0 ; j < n ; j++) {
            ferzle += a[i]*a[j];
        }
        if(ferzle==10000) {
            i=n;
        }
    }
}
```

(g)
```
void gruhop1(int n) {
    int gruhop = 0;
    for(int i = 0 ; i < n/2 ; i++) {
        for(int j = 0 ; j < n/2 ; j++) {
            gruhop += i*j;
        }
    }
}
```

(h)
```
void gruhop2(int n) {
    int gruhop = 0;
    for(int i = 0 ; i < sqrt(n) ; i++)
        for(int j = 0 ; j < n ; j++)
            gruhop += i*j;
}
```

(i)
```
int sumSomeStuff(int []A) {
    int sum=0;
    int i=0;
    while(i < A.length) {
        sum = sum + A[i];
        i++;
        if(sum > 100000) {
            i=A.length;
        }
    }
    return sum;
}
```

(j)
```
int doMoreStuff(int []A) {
    int sum=0;
    for(int i=0 ; i < A.length ; i++) {
        for(int j=0 ; j < A.length ; j++) {
            sum = sum + A[i]*A[j];
        }
        if(sum==123) {
            i = A.length;
        }
    }
    return sum;
}
```

(k)
```
int sumTimesM(int []A) {
    int M = 100;
    int sum=0;
    for(int i=0 ; i < A.length ; i++) {
        for(int j=0 ; j < M ; j++) {
            sum = sum + A[j] + A[i];
            if(sum==123) {
                j = M;
            }
        }
    }
    return sum;
}
```

(l)
```
void foo2(int n,int m) {
    int foo = 0;
    for(int i = 0 ; i < n ; i++)
        foo++;
    for(int j = 0 ; j < m ; j++)
        foo++;
}
```

(m)
```
void foo3(int n) { // Tricky one
    int foo = 0;
    for(int i = 1 ; sqrt(i)  <=  n ; i++)
        for(int j = 1 ; j <= i; j++)
            doIt(j) // takes j steps;
}
```

(n)
```
void HalfIt(int n) {
    while(n > 0) {
        n = n/2;
    }
}
```

Problem 7.11. Consider the problem of computing the product of two matrices, A and B, where A is $l \times m$ and B is $m \times n$.

(a) Give an efficient algorithm to compute the product $A \times B$. Assume you have a `Matrix` type with fields `rows` and `columns` that specify the number of rows/columns the matrix has. Thus, you can call `A.rows` to get the number of rows `A` has, for instance. Also assume you can index a Matrix like an array. Thus, `A[i][j]` accesses the element in row i and column j.

(b) Give the best and worst-case complexity of your algorithm.

Problem 7.12. Consider the following two implementations of selection sort.

```
void selectionSort(int a[],int n) {
    for (int i=0 ; i<n-1 ; i++) {
        int min = i;
        for (int j=i+1 ; j<n ; j++) {
            if(a[j] < a[min])
                min = j;
        }
        int temp = a[min];
        a[min] = a[i];
        a[i] = temp;
    }
}

void selectionSort(List<Integer> a,int n) {
    for (int i=0 ; i<n-1 ; i++) {
        int min = i;
        for (int j=i+1 ; j<n ; j++) {
            if(a.get(j) < a.get(min))
                min = j;
        }
        int temp = a.get(min);
        a.set(min, a.get(i));
        a.set(i,temp);
    }
}
```

(a) Give the worst-case complexity of the array version of selection sort.

(b) Give the worst-case complexity of the list version of selection sort assuming the list is an array-based implementation (e.g. ArrayList).

(c) Give the worst-case complexity of the list version of selection sort assuming the list is a linked-list implementation.

(d) Compare the three options. Is one of them the clear choice to use? Should any of them never be used? Explain.

Problem 7.13. Using the code from Example 7.153, determine the complexity of the following method calls.

(a) `hs1.retainAll(al2)`, where `hs1` is a HashSet of size n and `al2` is an ArrayList of size m.

(b) `hs1.retainAll(ll2)`, where `hs1` is a HashSet of size n and `ll2` is a LinkedList of size m.

(c) `hs1.retainAll(ts2)`, where `hs1` is a HashSet of size n and `ts2` is a TreeSet of size m.

(d) `hs1.retainAll(hs2)`, where `hs1` is a HashSet of size n and `hs2` is a HashSet of size m.

(e) `ll1.retainAll(al2)`, where `ll1` is a LinkedList of size n and `al2` is an ArrayList of size m.

(f) `ll1.retainAll(ll2)`, where `ll1` is a LinkedList of size n and `ll2` is a LinkedList of size m.

(g) `ll1.retainAll(ts2)`, where `ll1` is a LinkedListof size n and `ts2` is a TreeSet of size m.

(h) `ll1.retainAll(hs2)`, where `ll1` is a LinkedList of size n and `hs2` is a HashSet of size m.

Problem 7.14. You need to choose data structures for two collections of data, A and B, and the only thing you know is that the most common operation you will perform is `A.retainAll(B)`. Given this, what are you best choices for A and B? Clearly justify your choices.

Problem 7.15. In Java, a TreeMap is an implementation of the Map interface that uses a balanced binary search tree (a red-black tree) to store the keys and values. In particular, the keys are used as keys in a BST with each key having an associated value. TreeMaps have methods like[1] `put(Object key, Object value)` (add the key-value pair to the map), `Object get(Object key)` (returns the value associated with the key), and `ArrayList keySet()` (returns an ArrayList of all the keys). As should be expected, `put` and `get` both take $\Theta(\log n)$ time. You can assume that `keySet` takes $\Theta(n)$ time.

A method that might be useful on a TreeMap is `ArrayList getAll(ArrayList keys)` that returns an ArrayList containing the values associated to the keys passed into the method. Consider the following implementation of this method.

```
public ArrayList getAll(ArrayList keys) {
    ArrayList toReturn = new ArrayList();
    for (Object key : keySet()) {
        for (Object k : keys) {
            if (k.equals(key)) {
                toReturn.add(get(key));
            }
        }
    }
    return toReturn;
}
```

(a) Does this method work properly? Explain why it does or does not.

(b) What is the worst-case complexity of this method?

(c) Rewrite the method so that it is as efficient as possible and give the worst-case complexity of the new version.

[1] For simplicity we are ignoring generics here. If you don't know what that means but you understand what this problem is saying, don't worry about it.

Problem 7.16. Consider the following implementation of binary search on a List.

```
int binarySearch(List A, int n, int val) {
   int left=0, right=n-1;
   while (right-left>=0) {
       int middle = (left+right)/2;
       if(val==A.get(middle))
           return middle;
       else if(val<A.get(middle))
           right=middle-1;
       else
           left=middle+1;
    }
   return -1;
}
```

(a) Give the worst-case complexity of this algorithm if A is an array-based list (e.g., an ArrayList).

(b) Give the worst-case complexity of this algorithm if A is linked list.

(c) Would it ever make sense to implement binary search on a linked list? Explain.

Chapter 8

Recursion, Recurrences, and Mathematical Induction

In this chapter we will explore a proof technique, an algorithmic technique, and a mathematical technique. Each topic is in some ways very different than the others, yet they have a whole lot in common. They are also often used in conjunction.

You have already seen *recurrence relations*. Recall that a recurrence relation is a way of defining a sequence of numbers with a formula that is based on previous numbers in the sequence. You are probably also familiar with *recursion*, an algorithmic technique in which an algorithm calls itself (such an algorithm is called *recursive*), typically with "smaller" input. Finally, the *principle of mathematical induction* is a slick proof technique that works so well that sometimes it feels like you are cheating.

We will see that induction can be used to prove formulas, prove that algorithms—especially recursive ones—are correct, and help solve recurrence relations. Among other things, recurrence relations can be used to analyze recursive algorithm. Recursive algorithms can be used to compute the values defined by recurrence relations and to solve problems that can be broken into smaller versions of themselves.

As we will see, each of these has one or more *base cases* that can be proved/computed/determined directly and a *recursive* or *inductive* step that relies on previous steps. With each, the inductive/recursive steps must eventually lead to a base case.

Because induction can be used to prove things about the other two, we will begin there.

8.1 Mathematical Induction

Let's begin our study of mathematical induction (often just called induction) with an example that should look familiar. It is actually Theorem 5.25 that we proved in an earlier chapter. Following that, we will explain how/why induction works and give plenty of other examples.

Example 8.1. Let A be a set with n elements. Prove that $|P(A)| = 2^n$.

Proof: We use induction and the idea from the solution to Exercise 5.21. Clearly if $|A| = 1$, A has $2^1 = 2$ subsets: $\varnothing$ and A itself.
Assume every set with $n-1$ elements has 2^{n-1} subsets. Let A be a set with n elements. Choose some $x \in A$. Every subset of A either contains x or it doesn't.

Those that do not contain x are subsets of $A \setminus \{x\}$. Since $A \setminus \{x\}$ has $n-1$ elements, the induction hypothesis implies that it has 2^{n-1} subsets. Every subset that does contain x corresponds to one of the subsets of $A \setminus \{x\}$ with the element x added. That is, for each subset $S \subseteq A \setminus \{x\}$, $S \cup \{x\}$ is a subset of A containing x. Clearly there are 2^{n-1} such new subsets. Since this accounts for all subsets of A, A has $2^{n-1} + 2^{n-1} = 2^n$ subsets. $\square$

Now we will go into detail about how and why induction works.

8.1.1 The Basics

The *principle of mathematical induction* (*PMI*, or simply *induction*) is usually used to prove statements of the form

$$\text{for all } n \geq a, P(n) \text{ is true},$$

where a is an integer, and $P(n)$ is a propositional function with domain $\{a, a+1, a+2, \ldots\}$. Most often a is either 0 or 1, so the domain is usually $\mathbb{N}$ or $\mathbb{Z}^+$.

Induction is based on the following fairly intuitive observation (which we will formalize next). Suppose that we are to perform a task that involves a certain number of steps. Suppose that these steps must be followed in strict numerical order. Finally, suppose that we know how to perform the n-th task provided we have accomplished the $(n-1)$-th task. Thus if we are ever able to start the job (that is, if we have a base case), then we should be able to finish it (because starting with the base case we go to the next case, and then to the case following that, etc.).

⋆**Exercise 8.2.** Based on the description given so far, which of the following statements might we be able to use induction to prove (indicate with 'Y' or 'N')? Give a brief justification.

(a) ___The square of any integer is positive.

(b) ___Every positive integer can be written as the sum of two other positive integers.

(c) ___Every integer greater than 1 can be written as the product of prime numbers.

(d) ___If $n \geq 1$, $\displaystyle\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$

(e) ___Every real number is the square of another real number.

The following example illustrates the idea behind induction. It uses *modus ponens*. Recall that modus ponens states that if p is true and $p \to q$ is true, then q is true. In English, "If p is true, and whenever p is true q is true, then q is true."[1]

Example 8.3. Assume that we know that $P(1)$ is true and that whenever $k \geq 1$, $P(k) \to P(k+1)$ is true. What can we conclude?

> **Solution:** Let's start from the ground up. We know that $P(1)$ is true. We also know that $P(k) \to P(k+1)$ is true for any integer $k \geq 1$. For instance, since $4 \geq 1$, we know that $P(4) \to P(5)$ is true. It should be noted that we don't (yet) know anything about the truth values of $P(4)$ and $P(5)$.
>
> - We know $P(1)$ is true, and since $1 \geq 1$, $P(1) \to P(2)$ is true, so $P(2)$ is true.
> - Since $P(2)$ is true, and since $2 \geq 1$, $P(2) \to P(3)$ is true, therefore $P(3)$ is true.
> - Since $P(3)$ is true, and since $3 \geq 1$, $P(3) \to P(4)$ is true, therefore $P(4)$ is true.
> - Since $P(4)$ is true, and since $4 \geq 1$, $P(4) \to P(5)$ is true, therefore $P(5)$ is true.
> - Since $P(5)$ is true, and since $5 \geq 1$, $P(5) \to P(6)$ is true, therefore $P(6)$ is true.
>
> It seems pretty clear that this pattern continues for all values of $k > 6$ as well, so $P(k)$ is true for all $k \geq 1$.

⋆**Question 8.4.** Example 8.3 had several statements like the following:

> "Since $P(4)$ is true, and since $4 \geq 1$, $P(4) \to P(5)$ is true, therefore $P(5)$ is true."

What is the justification for the conclusion that $P(5)$ is true?

Answer ________________________________

Example 8.3 did not give a formal *proof* of the conclusion. The idea is to get you thinking about how induction works, not to provide a formal proof that it does. Once you wrap your head around it (it takes some people longer than others), you will believe it works regardless of whether or not you have seen a formal/complete proof that it does.

Before moving on, we should make sure you understand what has already been said.

⋆**Question 8.5.** If you know that $P(5)$ is true, and you also know that $P(k) \to P(k+1)$ whenever $k \geq 1$, what can you conclude?

Answer ________________________________

Now it is time to get really formal with our discussion. Induction is based on the fact that if $P(a)$ is true for some $a \geq 0$ (the *base case*), and for any $k \geq a$, if $P(k)$ is true, then $P(k+1)$ is true (the *inductive case*), then $P(n)$ is true for all $n \geq a$. In other words, the principle of mathematical induction is based on the tautology

$$[P(a) \wedge \forall k(P(k) \to P(k+1))] \to (\forall n P(n)),$$

where the universe is $\{a, a+1, a+2, \ldots\}$.

[1]We can also write this as the tautology $[p \wedge (p \to q)] \to q$.

⋆**Exercise 8.6.** Restate $[P(a) \wedge \forall k(P(k) \rightarrow P(k+1))] \rightarrow (\forall n P(n))$ (where the universe is $\{a, a+1, a+2, \ldots\}$) in English.

Answer ____

____

____

We won't prove that this is a tautology, but hopefully Example 8.3 helped convince you that it is indeed a tautology. *It is definitely worth your time to convince yourself that this is a tautology. If you aren't convinced, reread the example, think about it some more, and/or ask someone to help you understand it.*

⋆**Question 8.7.** Are you convinced that $[P(a) \wedge \forall k(P(k) \rightarrow P(k+1))] \rightarrow (\forall n P(n))$ is a tautology?

Answer ____

We call $P(a)$ the *base case*. Sometimes we actually need to prove several base cases (we will see why later). For instance, we might need to prove $P(a)$, $P(a+1)$, and $P(a+2)$ are all true.

The *inductive step* involves proving that $\forall k(P(k) \rightarrow P(k+1))$ is true. To prove it, we show that if $P(k)$ is true for any k which is *at least as large as the base case(s)*, then $P(k+1)$ is true. The assumption that $P(k)$ is true is called the *inductive hypothesis*.

Based on our discussion so far, here is the procedure for writing induction proofs.

Procedure 8.8. *To use induction to prove that $\forall n P(n)$ is true on domain $\{a, a+1, \ldots\}$:*

1. **Base Case:** *Show that $P(a)$ is true (and possible one or more additional base cases).*
2. *Show that $\forall k(P(k) \rightarrow P(k+1))$ is true. To show this:*
   - *(a)* **Inductive Hypothesis:** *Let $k \geq a$ be an integer and assume that $P(k)$ is true.*
   - *(b)* **Inductive Step:** *Prove that $P(k+1)$ is true, typically using the fact that $P(k)$ is true.*

   Assuming we used no special facts about k other than $k \geq a$, this means we have shown that $\forall k(P(k) \rightarrow P(k+1))$ (again, where it is understood that the domain is $\{a, a+1, \ldots\}$).
3. **Summary:** *Conclude that $\forall n P(n)$ is true, usually by saying something like "Since $P(a)$ and $P(k) \rightarrow P(k+1)$ for all $k \geq a$, $\forall n P(n)$ is true by induction."*

As you will quickly learn, the *base case* is generally pretty easy, as is writing down the *inductive hypothesis*. The *summary* is even easier, since it almost always says the same thing. The *inductive step* is the longest and most complicated step. In fact, in mathematics and theoretical computer science journals, induction proofs often only include the inductive step since anyone reading papers in such journals can generally fill in the details of the other three parts. But keep in mind that you are not (yet) writing papers for such journals, so you *cannot* omit these steps!

Let's see another example.

Example 8.9. Prove that the sum of the first n odd integers is n^2. That is, show that $\sum_{i=1}^{n}(2i-1) = n^2$ for all $n \geq 1$.

Proof: Let $P(n)$ be the statement "$\sum_{i=1}^{n}(2i-1) = n^2$". We need to show that $P(n)$ is true for all $n \geq 1$.

Base Case: Since $\sum_{i=1}^{1}(2i-1) = 2 \cdot 1 - 1 = 1 = 1^2$, $P(1)$ is true.

Inductive Hypothesis: Let $k \geq 1$ and assume that $P(k)$ is true. That is, assume that $\sum_{i=1}^{k}(2i-1) = k^2$ when $k \geq 1$.

Inductive Step: Then

$$\begin{aligned}
\sum_{i=1}^{k+1}(2i-1) &= \sum_{i=1}^{k}(2i-1) + (2(k+1)-1) \quad \text{(take } k+1 \text{ term from sum)} \\
&= k^2 + (2k+2-1) \quad \text{(by the \textit{inductive hypothesis})} \\
&= k^2 + 2k + 1 \\
&= (k+1)^2
\end{aligned}$$

Thus $P(k+1)$ is true.

Summary: Since we proved that $P(1)$ is true, and that $P(k) \to P(k+1)$ whenever $k \geq 1$, $P(n)$ is true for all $n \geq 1$ by the principle of mathematical induction. $\square$

The previous proof had the four components we discussed. We proved the *base case*. We then assumed it was true for k. That is, we made the *inductive hypothesis*. Next we proved that it was true for $k+1$ based on the assumption that it is true for k. That is, we did the *inductive step*. Finally, we appealed to the principle of mathematical induction in the *summary*.

Note: *Recall the following statement from Example 8.9:*

Let $P(n)$ be the statement "$\sum_{i=1}^{n}(2i-1) = n^2$".

Did you notice the quotes? It is important that you include these. This is particularly important if you use notation such as $P(n) =$ "$\sum_{i=1}^{n}(2i-1) = n^2$". Without the quotes, this becomes $P(n) = \sum_{i=1}^{n}(2i-1) = n^2$, which is defining $P(n)$ to be $\sum_{i=1}^{n}(2i-1)$ and saying that it is also equal to n^2. These are **not** *saying the same thing. With the quotes, $P(n)$ is a propositional function. Without them, it is a function from $\mathbb{Z}$ to $\mathbb{Z}$.*

In fact, to avoid this confusion, I recommend that you never use the equals sign with propositional functions, especially when writing induction proofs.

⋆**Fill in the details 8.10.** Reprove Theorem 6.48 using induction. That is, prove that for $n \geq 1$, $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.

Proof: Let $P(k)$ be the statement "$\sum_{i=1}^{k} i = \frac{k(k+1)}{2}$". We need to show that $P(n)$ is true for all $n \geq 1$.

Base Case: When $k = 1$, we have $\sum_{i=1}^{1} i = 1 =$ ______________. Therefore,

____________________________.

Inductive Hypothesis: Let $k \geq 1$, and assume that ________________________.

That is, assume that ____.

[This is not part of the proof, but it will help us see what's next. Our goal in the next step is to prove that __________ is true. That is, we need to show that ___________________________________.]

Inductive Step: Notice that

$$
\begin{aligned}
\sum_{i=1}^{k+1} i &= \underline{\hspace{6cm}} + (k+1) \\
&= \underline{\hspace{8cm}} + (k+1) \text{(by the inductive hypothesis)} \\
&= (k+1)\left(\frac{\hspace{6cm}}{}\right) \\
&= \underline{\hspace{8cm}}
\end{aligned}
$$

Thus, ____________________.

Summary: We showed that _________________ and that whenever ___________, $P(k) \rightarrow P(k+1)$, therefore $P(n)$ is true for ______________ by _________________

□

8.1.2 Equalities/Inequalities

The last few example induction proofs have dealt with statements of the form

$$LHS(k) = RHS(k),$$

where LHS stands for *left hand side* and RHS stands for *right hand side*. For instance, in Example 8.9, the statement was

$$\sum_{i=1}^{n}(2i-1) = n^2,$$

so $LHS(k) = \displaystyle\sum_{i=1}^{k}(2i-1)$ and $RHS(k) = k^2$.

⋆**Question 8.11.** Let $P(n)$ be the statement "$\displaystyle\sum_{i=1}^{n} i \cdot i! = (n+1)! - 1$." Determine each of the following:

(a) $P(k)$ is the statement ________________________________.

(b) $P(k+1)$ is the statement ________________________________.

(c) $LHS(k) =$ ________________________________

(d) $RHS(k) =$ ________________________________

(e) $LHS(k+1) =$ ________________________________

(f) $RHS(k+1) =$ ________________________________

For statements of this form, the goal of the inductive step is to show that $LHS(k+1) = RHS(k+1)$ given the fact that $LHS(k)$=$RHS(k)$ (the inductive hypothesis). The way this should generally be done is as follows:

Procedure 8.12. *Given a proposition of the form "$LHS(n) = RHS(n)$," the algebra in the inductive step of an induction proof should be done as follows:*

$$\begin{array}{rcll} LHS(k+1) & = & LHS(k) + stuff & \textit{(apply algebra to separate } LHS(k) \textit{ from the rest)} \\ & = & RHS(k) + stuff & \textit{(use the inductive hypothesis to replace } LHS(k) \\ & & & \textit{with } RHS(k)\textit{)} \\ & = & \cdots & \textit{(1 or more steps, usually involving algebra, that} \\ & = & RHS(k+1) & \textit{result in the goal of getting to } RHS(k+1)\textit{)} \end{array}$$

The last few examples followed this procedure, and your proofs should also follow it. Notice that these examples *do not* begin the inductive step by writing out $LHS(k+1) = RHS(k+1)$. One of them wrote it out, but it was *before* the inductive step for the purpose of making the goal in the inductive step clear. The inductive step should always begin by writing just $LHS(k+1)$, and should then use algebra, the inductive hypothesis, etc., until $RHS(k+1)$ is obtained.

This technique also works (with the appropriate slight modifications) with inequalities, e.g.

$$LHS(k) \leq RHS(k) \text{ and}$$

$$LHS(k) \geq RHS(k).$$

For instance, if $P(k)$ is the statement "$k > 2^k$", $LHS(k) = k$, and $RHS(k) = 2^k$. In addition, the '$+stuff$' is not always literally addition. For instance, it might be $LHS(k) \times stuff$.

Here is another example of this type of induction proof–this time using an inequality.

Example 8.13. Prove that $n < 2^n$ for all integers $n \geq 1$.

Proof: Let $P(n)$ be the statement "$n < 2^n$". We want to prove that $P(n)$ is true for all $n \geq 1$.
Base Case: Since $1 < 2^1$, $P(1)$ is clearly true.
Hypothesis: We assume $P(k)$ is true if $k \geq 1$. That is, $k < 2^k$.

Next we need to show that $P(k+1)$ is true. That is, we need to show that $(k+1) < 2^{k+1}$. (Notice that I did not state that this was true, and I do not start with this statement in the next step. I am merely pointing out what I need to prove.*) This paragraph is not really part of the proof–think of it as a side-comment or scratch work.*

Inductive: Given that $k < 2^k$, we can see that

$$\begin{aligned} k+1 &< 2^k + 1 && (\text{since } k < 2^k) \\ &< 2^k + 2^k && (\text{since } 1 < 2^k \text{ when } k \geq 1) \\ &= 2(2^k) \\ &= 2^{k+1} \end{aligned}$$

Since we have shown that $k+1 < 2^{k+1}$, $P(k+1)$ is true.
Summary: Since we proved that $P(1)$ is true, and that $P(k) \rightarrow P(k+1)$, by *PMI*, $P(n)$ is true for all $n \geq 1$. $\square$

In the previous example, $LHS(k) = k$, so $LHS(k+1)$ is already in the form $LHS(k) + stuff$ since $LHS(k+1) = k+1 = LHS(k)+1$. So the first step of algebra is unnecessary and we were able to apply the inductive hypothesis immediately. Don't let this confuse you. This is essentially the same as the other examples minus the need for algebra in the first step.

Note: *By the time you are done with this section, you will likely be tired of hearing this, but since it is the most common mistake made in induction proofs, it is worth repeating* ad nauseam. **Never begin the inductive step of an induction proof by writing down $\mathbf{P(k+1)}$.** *You do not know it is true yet, so it is not valid to write it down as if it were true so that you can use a technique such as working both sides to verify that it is true (which, as we have also previously stated, is not a valid proof technique).*

You **can** *(and sometimes* **should***) write down $P(k+1)$ on another piece of paper or with a comment such as "We need to prove that" preceding it so that you have a clear direction for the inductive step.*

If you can complete the next exercise without too much difficulty, you are well on your way to understanding how to write induction proofs.

⋆**Exercise 8.14.** Use induction to prove that for all $n \geq 1$, $\displaystyle\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$.
(Hint: Follow the techniques and format of the previous examples and be smart about your algebra and it will go a lot easier. Also, you will need to factor a polynomial in the inductive step, but if you determine what the goal is ahead of time, it shouldn't be too difficult.)

Proof:

8.1.3 Variations

In this section we will discuss a few slight variations of the details we have presented so far. First we discuss the fact that we do not need to use a propositional function. Then we will discuss a variation regarding the inductive hypothesis.

It is not always necessary to explicitly define $P(k)$ for use in an induction proof. $P(k)$ is used mostly for convenience and clarity. For instance, in the solution to the previous exercise, it allowed us to just say

$$\text{“}P(k) \text{ is true”}$$

instead of saying

$$\text{“}\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}\text{”} \quad \text{(which is long)}$$

or

"the statement is true for k" (which is a little vague/awkward).

Here is an example that does not use $P(k)$. It also does not label the four parts of the proof. That is perfectly fine. The main reason we have done so in previous examples is to help you identify them more clearly.

Example 8.15. Let f_n be the n-th Fibonacci number. Prove that for all integers $n \geq 1$,

$$f_{n-1}f_{n+1} = f_n^2 + (-1)^n.$$

Proof: For $k = 1$, we have

$$f_0 f_2 = 0 \cdot 1 = 0 = 1 - 1 = 1^2 + (-1)^1 = f_1^2 + (-1)^1,$$

and so the assertion is true for $k = 1$. Suppose $k \geq 1$, and that the assertion is true for k. That is,

$$f_{k-1}f_{k+1} = f_k^2 + (-1)^k.$$

This can be rewritten as

$$f_k^2 = f_{k-1}f_{k+1} - (-1)^k$$

(a fact that we will find useful below). Then

$$\begin{aligned}
f_k f_{k+2} &= f_k(f_{k+1} + f_k) && \text{(by definition of } f_n \text{ applied to } f_{k+2}\text{)} \\
&= f_k f_{k+1} + f_k^2 \\
&= f_k f_{k+1} + f_{k-1}f_{k+1} - (-1)^k && \text{(by rewritten inductive hypothesis)} \\
&= f_{k+1}(f_k + f_{k-1}) + (-1)^{k+1} \\
&= f_{k+1}f_{k+1} + (-1)^{k+1} && \text{(by the definition of } f_k\text{)} \\
&= f_{k+1}^2 + (-1)^{k+1},
\end{aligned}$$

and so the assertion is true for $k + 1$. The result follows by induction. $\square$

⋆**Exercise 8.16.** Use induction to prove that for all $n \geq 1$,

$$1 \cdot 2 + 2 \cdot 2^2 + 3 \cdot 2^3 + \cdots + n \cdot 2^n = 2 + (n-1)2^{n+1}$$

or if you prefer,

$$\sum_{i=1}^{n} i \cdot 2^i = 2 + (n-1)2^{n+1}.$$

Do so without using a propositional function. You may label the four parts of your proof, but it is not required.

Proof:

Example 8.17. Prove the generalized form of DeMorgan's law. That is, show that for any $n \geq 2$, if p_1, p_2, ..., p_n are propositions, then

$$\neg(p_1 \vee p_2 \vee \cdots \vee p_n) = (\neg p_1 \wedge \neg p_2 \wedge \cdots \wedge \neg p_n).$$

We provide several appropriate proofs of this one (and one inappropriate one).

Proof 1: (A typical proof)

Let $P(n)$ be the statement "$\neg(p_1 \vee p_2 \vee \cdots \vee p_n) = (\neg p_1 \wedge \neg p_2 \wedge \cdots \wedge \neg p_n)$." We want to show that for all $n \geq 2$, $P(n)$ is true. $P(2)$ is DeMorgan's law, so the base case is true. Assume $P(k)$ is true. Then

$$\begin{array}{rcll} \neg(p_1 \vee p_2 \vee \cdots \vee p_{k+1}) & = & \neg((p_1 \vee p_2 \vee \cdots \vee p_k) \vee p_{k+1}) & \text{associative law} \\ & = & \neg(p_1 \vee p_2 \vee \cdots \vee p_k) \wedge \neg p_{k+1} & \text{DeMorgan's law} \\ & = & (\neg p_1 \wedge \neg p_2 \wedge \cdots \wedge \neg p_k) \wedge \neg p_{k+1} & \text{hypothesis} \\ & = & (\neg p_1 \wedge \neg p_2 \wedge \cdots \wedge \neg p_k \wedge \neg p_{k+1}) & \text{associative law} \end{array}$$

Thus $P(k+1)$ is true. Since we proved that $P(2)$ is true, and that $P(k) \to P(k+1)$ if $k \geq 2$, by *PMI*, $P(n)$ is true for all $n \geq 2$. $\square$

Proof 2: (Not explicitly defining/using $P(n)$)

We know that $\neg(p_1 \vee p_2) = (\neg p_1 \wedge \neg p_2)$ since this is simply DeMorgan's law. Assume the statement is true for k. That is, $\neg(p_1 \vee p_2 \vee \cdots \vee p_k) = (\neg p_1 \wedge \neg p_2 \wedge \cdots \wedge \neg p_k)$. Then we can see that

$$\begin{array}{rcll} \neg(p_1 \vee p_2 \vee \cdots \vee p_{k+1}) & = & \neg((p_1 \vee p_2 \vee \cdots \vee p_k) \vee p_{k+1}) & \text{associative law} \\ & = & \neg(p_1 \vee p_2 \vee \cdots \vee p_k) \wedge \neg p_{k+1} & \text{DeMorgan's law} \\ & = & (\neg p_1 \wedge \neg p_2 \wedge \cdots \wedge \neg p_k) \wedge \neg p_{k+1} & \text{hypothesis} \\ & = & (\neg p_1 \wedge \neg p_2 \wedge \cdots \wedge \neg p_k \wedge \neg p_{k+1}) & \text{associative law} \end{array}$$

Thus the statement is true for $k+1$. Since we have shown that the statement is true for $n = 2$, and that whenever it is true for k it is true for $k+1$, by *PMI*, the statement is true for all $n \geq 2$. $\square$

Sometimes it is acceptable to omit the justification in the summary. That is, there isn't necessarily a need to restate what you have proven and you can just jump to the conclusion. So the previous proof could end as follows:

> Thus the statement is true for $k+1$. By *PMI*, the statement is true for all $n \geq 2$.

Proof 3: (common in journal articles, unacceptable for this class)

The result follows easily by induction. $\square$

⋆**Evaluate 8.18.** Prove that for all positive integers n, $\sum_{i=1}^{n} i \cdot i! = (n+1)! - 1$.

Solution: Base: $n = 1$

$$\begin{aligned} 1 \cdot 1! &= (1+1)! - 1 \\ 1 &= 2! - 1 \\ 1 &= 1 \end{aligned}$$

Assume $\sum_{i=1}^{n} i \cdot i! = (n+1)! - 1$ for $n \geq 1$.

Induction:

$$\begin{aligned} \sum_{i=1}^{n+1} i \cdot i! &= \sum_{i=1}^{n} i \cdot i! + (n+1)(n+1)! \\ &= (n+1)! - 1 + (n+1)(n+1)! \\ &= (n+1+1)(n+1)! - 1 \\ &= (n+2)(n+1)! - 1 \\ &= (n+2)! - 1 \end{aligned}$$

Therefore it is true for n. Thus by PMI it is true for $n \geq 1$.

Evaluation ____

The second variation we wish to discuss has to do with the inductive hypothesis/step. In the inductive step, we can replace $P(k) \to P(k+1)$ with $P(k-1) \to P(k)$ as long as we prove the statement for all k *larger than any of the base cases*. In general, we can use whatever index we want for the inductive hypothesis as long as we use it to prove that the statement is true for the next index, and as long as we are sure to cover all of the indices down to the base case. For instance, if we prove $P(k+3) \to P(k+4)$, then we need to show it for all $k+3 \geq a$ (that is, all $k \geq a-3$), assuming a is the base case. Put simply, the assumption we make about the value of k must guarantee that the inductive hypothesis includes the base case(s).

⋆**Question 8.19.** Consider a 'proof' of $\forall n P(n)$ that shows that $P(1)$ is true and that $P(k) \to P(k+1)$ for $k > 1$. What is wrong with such a proof?

Answer ____

Note: *Whether you assume $P(k)$ or $P(k-1)$ is true, you must specify the values of k precisely based on your choice. For instance, if you assume $P(k)$ is true for all $k > a$, you have a problem. Although you known $P(a)$ is true (because it is a base case), when you assume $P(k)$ is true for $k > a$, the smallest k can be is $a + 1$. In other words, when you prove $P(k) \to P(k+1)$, you leave out $P(a) \to P(a+1)$. But that means you can't get anywhere from the base case, so the whole proof is invalid.*

If you are wondering why we would use $P(k-1)$ as the inductive hypothesis instead of $P(k)$, it is because sometimes it makes the proof easier–for instance, the algebra steps involved might be simpler.

Example 8.20. Prove that the expression

$$3^{3n+3} - 26n - 27$$

is a multiple of 169 for all natural numbers n.

Proof: Let $P(k)$ be the statement "$3^{3k+3} - 26k - 27 = 169N$ for some $N \in \mathbb{N}$."
We will prove that $P(0)$ is true and that $P(k-1) \to P(k)$.
When $k = 0$ notice that $3^{3\cdot 0+3} - 26 \cdot 0 - 27 = 27 - 27 = 0 = 169 \cdot 0$, so $P(0)$ is true.
Let $k > 0$ and assume $P(k-1)$ is true. That is, there is some $N \in \mathbb{N}$ such that $3^{3(k-1)+3} - 26(k-1) - 27 = 169N$. After a little algebra, this is the same as $3^{3k} - 26k - 1 = 169N$. Then

$$\begin{aligned}
3^{3k+3} - 26k - 27 &= 27 \cdot 3^{3k} - 26k - 27 \\
&= 27 \cdot 3^{3k} + (26 - 27)26k - 27 \\
&= 27 \cdot 3^{3k} - 27 \cdot 26k - 27 + 26 \cdot 26k \\
&= 27(3^{3k} - 26k - 1) + 676k \\
&= 27 \cdot 169N + 169 \cdot 4k \text{ (By the inductive hypothesis)} \\
&= 169(27 \cdot N + \cdot 4k)
\end{aligned}$$

which is divisible by 169. The assertion is thus established by induction. □

⋆**Question 8.21.** Did you notice that in the previous example we assumed $k > 0$ instead of $k \geq 0$? Why did we do that?

Answer ______________________________

8.1.4 Strong Induction

The form of induction we have discussed up to this point only assumes the statement is true for one value of k. This is sometimes called *weak induction*. In *strong induction*, we assume that the statement is true for all values up to and including k. In other words, with strong induction, the inductive hypothesis involves proving that

$$[P(a) \wedge P(a+1) \wedge \cdots \wedge P(k)] \to P(k+1) \text{ if } k \geq a.$$

This may look more complicated, but practically speaking, there is really very little difference. Essentially, strong induction just allows us to assume *more* than weak induction. Let's see an example of why we might need strong induction.

Example 8.22. Show that every integer $n \geq 2$ can be written as the product of primes.

Proof: Let $P(n)$ be the statement "n can be written as the product of primes." We need to show that for all $n \geq 2$, $P(n)$ is true.
Since 2 is clearly prime, it can be written as the product of one prime. Thus $P(2)$ is true.
Assume $[P(2) \wedge P(3) \wedge \cdots \wedge P(k-1)]$ is true for $k > 2$. In other words, assume all of the numbers from 2 to $k-1$ can be written as the product of primes.
We need to show that $P(k)$ is true. If k is prime, clearly $P(k)$ is true. If k is not prime, then we can write $k = a \cdot b$, where $2 \leq a \leq b < k$. By hypothesis, $P(a)$ and $P(b)$ are true, so a and b can be written as the product of primes. Therefore, k can be written as the product of primes, namely the primes from the factorizations of a and b. Thus $P(k)$ is true.
Since we proved that $P(2)$ is true, and that $[P(2) \wedge P(3) \wedge \cdots \wedge P(k-1)] \rightarrow P(k)$ if $k > 2$, by the principle of mathematical induction, $P(n)$ is true for all $n \geq 2$. That is, every integers $n \geq 2$ can be written as the product of primes. $\square$

Example 8.23. In the country of SmallPesia coins only come in values of 3 and 5 pesos. Show that any quantity of pesos greater than or equal to 8 can be paid using the available coins.

Proof: **Base Case:** Observe that $8 = 3+5$, $9 = 3+3+3$, and $10 = 5+5$, so we can pay 8, 9, or 10 pesos with the available coinage.

Inductive Hypothesis: Assume we can pay any value from 8 to $k-1$ pesos, where $k \geq 11$.

Inductive step: The inductive hypothesis implies that we can pay with $k-3$ pesos. We can add to the coins used for $k-3$ pesos a single coin of value 3 in order to pay for k pesos.

Summary: Since we can pay for 8, 9, and 10 pesos, and whenever we can pay for anything between 8 and $k-1$ pesos we can pay for k pesos, the strong form of induction implies that we can pay for any quantity of pesos $n \geq 8$.

Notice that the reason we needed three base cases for this proof was the fact that we looked back at $k-3$, *three value previous to the value of interest. If we had only proven it for* 8, *we would have needed to prove* 9 *and (more importantly)* 10 *in the inductive step. But the inductive step doesn't work for* 10 *since there is no solution for* $10-3=7$ *pesos.* $\square$

Notice that there is no way we could have used weak induction in either of the previous examples.

8.1.5 Induction Errors

The following examples should help you appreciate why we need to be very precise/careful when writing induction proofs.

Example 8.24. What is wrong with the following (supposed) proof that $a^n = 1$ for $n \geq 0$:

Proof: *Base case:* Since $a^0 = 1$, the statement is true for $n = 0$.
Inductive step: Let $k > 0$ and assume $a^j = 1$ for $0 \leq j \leq k$. Then

$$a^{k+1} = \frac{a^k \cdot a^k}{a^{k-1}} = \frac{1 \cdot 1}{1} = 1.$$

Summary: Therefore by PMI, $a^n = 1$ for all $n \geq 0$. □

Solution: The base case is correct, and there is nothing wrong with the summary, assuming the inductive step is correct. $a^k = 1$ and $a^{k-1} = 1$ are correct by the inductive hypothesis since we are assuming $k > 0$. The algebra is also correct. So what is wrong? The problem is that when $k = 0$, a^{-1} would be in the denominator. But we don't know whether or not $a^{-1} = 1$. Thus we needed to assume $k > 0$. As it turns out, that is precisely where the problem lies. We proved that $P(0)$ is true and that $P(k) \to P(k+1)$ is true when $k > 0$. Thus, we know that $P(1) \to P(2)$, and $P(2) \to P(3)$, etc., but we never showed that $P(0) \to P(1)$ because, of course, it isn't true. The induction doesn't work without $P(0) \to P(1)$.

⋆**Evaluate 8.25.** Prove or disprove that all goats are the same color.

Solution: If there is one goat, it is obviously the same color as itself. Let $n \geq 1$ and assume that any collection of n goats are all the same color. Consider a collection of $n+1$ goats. Number the goats 1 through $n+1$. Then goats 1 through n are the same color (since there are n of them) and goats 2 through $n+1$ are the same color (again, since there are n of them). Since goat 2 is in both collections, the goats in both collections are the same color. Thus, all $n+1$ goats are the same color.

Evaluation ____

____

____

The next example deals with *binary palindromes*. Binary palindromes can be defined recursively by $\lambda, 0, 1 \in P$, and whenever $p \in P$, then $1p1 \in P$ and $0p0 \in P$. (Note: λ is the notation sometimes used to denote the *empty string*—that is, the string of length 0. Also, $1p1$ means the binary string obtained by appending 1 to the begin and end of string p. Similarly for $0p0$.) Notice that there is 1 palindrome of length 0 (λ), 2 of length 1 (0, 1), 2 of length 2 (00, 11), 4 of length 3 (000, 010, 101, 111), etc.

⋆**Evaluate 8.26.** Use induction to prove that the number of binary palindromes of length $2n$ (even length) is 2^n for all $n \geq 0$.

Proof 1: Base case: $k = 1$. The total number of palindromes of length $2 = 2$ is $2^1 = 2$. It is true.
Assume the total number of binary palindromes with length $2k$ is 2^k. To form a binary palindrome with length $2(k+1) = 2k+2$, with every element in the set of binary palindromes with length $2k$ we either put (00) or (11) to the end or beginning of it. Therefore, the number of binary palindromes with length $2(k+1)$ is twice as many as the number of binary palindromes with length $2k$, which is $2 \times 2^k = 2^{k+1}$. Thus it is true for $k+1$. By the principle of mathematical induction, the total number of binary palindromes of length $2n$ for $n \geq 1$ is 2^n.

Evaluation ____

Proof 2: For the base case, notice that there is $1 = 2^0$ palindromes of length 0 (the empty string). Now assume it is true for all n. For each consecutive binary number with n bits, you are adding a bit to either end, which multiplies the total number by 2^2 permutations, but for it to be a palindrome, they both have to be either 0 or 1, so it would just be 2 instead, so for binary numbers of length $2k$, there are 2^k palindromes.

Evaluation ____

Proof 3: The empty string is the only string of length 0, and it is a palindrome. Thus there is $1 = 2^0$ palindromes of length 0. Let $2n$ be the length, assume $2n \to 2^n$ palindromes. Now we look at $n+1$ so we know the length is $2n+2$ and it starts and ends with either 0 or 1 and has $2n$ values in between. Both possibilities imply 2^n palindromes, so $2^n + 2^n = 2^{n+1}$.

Evaluation ____

⋆**Exercise 8.27.** Based on the feedback from the previous Evaluate exercise, construct a proper proof that the number of binary palindromes of length $2n$ is 2^n for all $n \geq 0$.
Proof:

8.1.6 Summary/Tips

Induction proofs are both intuitive and non-intuitive. On the one hand, when you talk through the idea, it seems to make sense. On the other hand, it almost seems like you are using *circular reasoning*. It is important to understand that induction proofs do *not* rely on circular reasoning. Circular reasoning is when you assume p in order to prove p. But here we are not doing that. We are assuming $P(k)$ and using that fact to prove $P(k+1)$, a different statement. However, we are *not* assuming that $P(k)$ is true for all $k \geq a$. We are proving that ***if we assume that*** $P(k)$ ***is true***, then $P(k+1)$ is true. The difference between these statements may seem subtle, but it is important.

Let's summarize our approach to writing an induction proof. This is similar to Procedure 8.8 except we include several of the unofficial steps we have been using that often come in handy. You are not required to use this procedure, but if you are having a difficult time with induction proofs, try this out. Here is the brief version. After this we provide some further comments about each step.

Procedure 8.28. *A slightly longer approach to writing an induction proof is as follows:*

1. **Define:** *(optional) Define $P(n)$ based on the statement you need to prove.*
2. **Rephrase:** *(optional) Rephrase the statement you are trying to prove using $P(n)$. This step is mostly to help you be clear on what you need to prove.*
3. **Base Case:** *Prove the base case or cases.*
4. **Inductive Hypothesis:** *Write down the inductive hypothesis. Usually it is as simple as "Assume that $P(k)$ is true".*
5. **Goal:** *(optional) Write out the goal of the* inductive step *(coming next). It is usually "I need to show that $P(k+1)$ is true" It can be helpful to explicitly write out $P(k+1)$, although see important comments about this step below. This is another step that is mostly for your own clarity.*
6. **Inductive:** *Prove the* goal *statement, usually using the* inductive hypothesis.
7. **Summary:** *The typical induction summary.*

Here are some comments about the steps in Procedure 8.28.

1. **Define:** $P(n)$ should be a statement about a single instance, not about a series of instances. For example, it should be statements like "$2n$ is even" or "A set with n elements has 2^n subsets." It should *NOT* be of the form "$2n$ is even if $n > 1$," "$n^2 > 0$ if $n \neq 0$," or "For all $n > 1$, a set with n elements has 2^n subsets."

2. **Rephrase:** In almost all cases, the rephrased statement should be "For all $n \geq a$, $P(n)$ is true," where a is some constant, often 0 or 1. If the statement cannot be phrased in this way, induction may not be appropriate.

3. **Base Case:** For most statements, this means showing that $P(a)$ is true, where a is the value from the rephrased statement. Although usually one base case suffices, sometimes one must prove multiple base cases, usually $P(a)$, $P(a+1)$, $\ldots$, $P(a+i)$ for some $i > 0$. This depends on the details of the inductive step.

4. **Inductive Hypothesis:** This is almost always one of the following:

   - Assume that $P(k)$ is true.
   - Assume that $P(k-1)$ is true.
   - Assume that $[P(a) \wedge P(a+1) \wedge \cdots \wedge P(k)]$ is true (strong induction)

   Sometimes it is helpful to write out the hypothesis explicitly (that is, write down the whole statement with k or $k-1$ plugged in).

5. **Goal:** As previously stated, this is almost always "I need to show that $P(k+1)$ is true" (or "I need to show that $P(k)$ is true"). But it can be very helpful to explicitly write out what $P(k+1)$ is so you have a clear direction for the next step. However, *it is very important that you do not just write out $P(k+1)$ without prefacing it with a statement like "I need to show that...".* Since you are about to prove that $P(k+1)$ is true, you don't know that it is

true yet, so writing it down as if it is a fact is incorrect and confusing. In fact, it is probably better write the goal separate from the rest if the proof (e.g. on another piece of paper).

The goal does not need to be written down and is not really part of the proof. The only purpose of doing so it to help you see what you need to do in the next step. For instance, knowing the goal often helps you to figure out the required algebra steps to get there.

6. **Inductive:** This is the longest, and most varied, part of the proof. Once you get the hang of induction, you will typically only think about two parts of the proof—the base case and this step. The rest will become second nature.

   The inductive step should *not* start with writing down $P(k+1)$. Some students want to write out $P(k+1)$ and work both sides until they get them to be the same. As we have emphasized on several occasions, this is *not* a proper proof technique. You cannot start with something you do not know and then work it until you get to something you do know and then declare it is true.

7. **Summary:** This is easy. It is almost always either:

   > "Since we proved that $P(a)$ is true, and that $P(k) \to P(k+1)$, for $k \geq a$, then we know that $P(n)$ is true for all $n \geq a$ by *PMI*, " or

   > "Since we proved that $P(a)$ is true, and that $[P(a) \wedge P(a+1) \wedge \cdots \wedge P(k)] \to P(k+1)$, for $k \geq a$, $P(n)$ is true for all $n \geq a$ by *PMI*."

   The details change a bit depending on what your inductive hypothesis was (e.g. if it was $P(k-1)$ instead of $P(k)$). Technically speaking, you can just summarize your proof by saying

   > "Thus, $P(n)$ is true for all $n \geq a$ by *PMI*."

   As long as someone can look back and see that you included the two necessary parts of the proof, you do not necessarily need to point them out again.

8.2 Recursion

You have seen examples of recursion if you have seen Russian Matryoshka dolls (Google it), two almost parallel mirrors, a video camera pointed at the monitor, or a picture of a painter painting a picture of a painter painting a picture of a painter... More importantly for us, recursion is a very useful tool to implement algorithms. You probably already learned about recursion in a previous programming course, but we present the concept in this brief section for the sake of review, and because it ties in nicely with the other two topics in this chapter.

Definition 8.29. *An algorithm is* recursive *if it calls itself.*

Examples of recursion that you may have already seen include *binary search*, *Quicksort*, and *Mergesort*.

⋆**Question 8.30.** Is following algorithm recursive? Briefly explain.

```
int ferzle(int n) {
    if(n<=0) {
        return 3;
    } else {
        return ferzle(n-1) + 2;
    }
}
```

Answer ____

If a subroutine/function simply called itself as a part of its execution, it would result in infinite recursion. This is a bad thing. Therefore, when using recursion, one must ensure that at some point, the subroutine/function terminates without calling itself. We will return to this point after we see what is perhaps the quintessential example of recursion.

Example 8.31. Notice that

$$\begin{array}{rcll} 0! & = & 1 & \\ 1! & = & 1 & = \quad 1 \times 0! \\ 2! & = & 2 \times 1 & = \quad 2 \times 1! \\ 3! & = & 3 \times 2 \times 1 & = \quad 3 \times 2! \\ 4! & = & 4 \times 3 \times 2 \times 1 & = \quad 4 \times 3! \\ & & \text{and in general, when } n > 1, & \\ n! & = & n \times (n-1) \times \cdots \times 2 \times 1 & = \quad n \times (n-1)! \end{array}$$

In other words, we can define $n!$ recursively as follows:

$$n! = \begin{cases} 1 & \text{when } n = 0 \\ n * (n-1)! & \text{otherwise.} \end{cases}$$

This leads to the following recursive algorithm to compute $n!$.

```
// Returns n!, assuming n>=0.
int factorial(int n) {
```

```
        if(n<=0) {
            return 1;
        } else {
            return n*factorial(n-1);
        }
    }
```

To guarantee that they will terminate, every recursive algorithm needs all of the following.

1. *Base case(s)*: One or more cases which are solved non-recursively. In other words, when an algorithm gets to the base case, it does not call itself again. This is also called a *stopping case* or *terminating condition*.
2. *Inductive case(s)*: One or more recursive rule for all cases except the base case.
3. *Progress:* The inductive case(s) should always progress toward the base case. Often this means the arguments will get smaller until they approach the base case, but sometimes it is more complicated than this.

Example 8.32. Let's take a closer look at the `factorial` algorithm from Example 8.31. Notice that if $n \leq 0$, `factorial` does not make a recursive call. Thus, it has a *base case*. Also notice that when a recursive call is made to `factorial`, the argument is smaller, so it is approaching a base case (i.e. making *progress*). When $n > 0$, it is clearly making a recursive call, so it has *inductive cases*.

⋆**Question 8.33.** Consider the `ferzle` algorithm from Question 8.30 above.

(a) What is/are the base case/cases?

Answer ______________________________

(b) What are the inductive cases?

Answer ______________________________

(c) Do the inductive cases make progress?

Answer ______________________________

Example 8.34. Prove that the recursive `factorial(n)` algorithm from Example 8.31 returns $n!$ for all $n \geq 1$.

Proof: Notice that if $n = 0$, `factorial(0)` returns $1 = 0!$, so it works in that case. For $k \geq 0$, assume `factorial(k)` works correctly. That is, it returns $k!$. `factorial(k+1)` return $k+1$ times the value returned by `factorial(k)`. By the inductive hypothesis, `factorial(k)` returns $k!$, so `factorial(k+1)` returns $(k+1) \times k! = (k+1)!$, as it should. By PMI, `factorial(n)` returns $n!$ for all $n \geq 0$. ☐

Example 8.35. Implement an algorithm `countdown(int n)` that outputs the integers from n down to 1, where $n > 0$. So, for example, `countdown(5)` would output "5 4 3 2 1".

Solution: One way to do this is with a simple loop:

```
void countdown(int n) {
    for(i=n;i>0;i--)
        print(i);
}
```

We wouldn't learn anything about recursion if we used this solution. So let's consider how to do it with recursion. Notice that `countdown(n)` outputs n followed by the numbers from $n-1$ down to 1. But the numbers $n-1$ down to 1 are the output from `countdown(n-1)`. This leads to the following recursive algorithm:

```
void countdown(int n) {
    print(n);
    countdown(n-1):
}
```

To see if this is correct, we can trace through the execution of `countdown(3)`. The following table give the result.

Execution of	outputs	then executes
`countdown(3)`	3	`countdown(2)`
`countdown(2)`	2	`countdown(1)`
`countdown(1)`	1	`countdown(0)`
`countdown(0)`	0	`countdown(-1)`
`countdown(-1)`	-1	`countdown(-2)`
$\vdots$	$\vdots$	$\vdots$

Unfortunately, `countdown` will never terminate. We are supposed to stop printing when $n = 1$, but we didn't take that into account. In other words, we don't have a base case in our algorithm. To fix this, we can modify it so that a call to `countdown(0)` produces no output and does not call `countdown` again.

Calls to `countdown(n)` should also produce no output when $n < 0$. The following algorithm takes care of both problems and is our final solution.

```
void countdown(int n) {
    if(n>0) {
        print(n);
        countdown(n-1);
    }
}
```

Notice that when $n \leq 0$, `countdown(n)` does nothing, making $n \leq 0$ the *base cases*. When $n > 0$, `countdown(n)` calls `countdown(n-1)`, making $n > 0$ the *inductive cases*. Finally, when `countdown(n)` makes a recursive call it is to `countdown(n-1)`, so the inductive cases *progress* to the base case.

⋆**Exercise 8.36.** Prove that the recursive `countdown(n)` algorithm from Example 8.35 works correctly. (Hint: Use induction.)
Proof:

In general, we can solve a problem with recursion if we can:

1. Find one or more simple cases of the problem that can be solved directly.
2. Find a way to break up the problem into smaller instances of the *same* problem.
3. Find a way to combine the smaller solutions.

Let's see a few classic examples of the use of recursion.

Example 8.37. Consider the *binary search* algorithm to find an item v on a sorted list of size n. The algorithm works as follows.

- We compare the middle value m of the array to v.
- If the $m = v$, we are done.
- Else if $m < v$, we binary search the left half of the array.
- Else $(m > v)$, we binary search the right half of the array.
- Now, we have the same problem, but only half the size.

In Example 7.161 we saw the following iterative implementation of binary search:

```
int binarySearch(int a[], int n, int val) {
    int left=0, right=n-1;
    while (right>=left) {
        int middle = (left+right)/2;
        if(val==a[middle])
            return middle;
        else if(val<a[middle])
            right=middle-1;
        else
            left=middle+1;
     }
    return -1;
}
```

Here is a version that uses recursion. In this version we need to pass the endpoints of the array so we know what part of the array we are currently looking at.

```
int binarySearch(int[] a, int left, int right, int val) {
    if(right>=left) {
        int middle = (left+right)/2;
        if(val==a[middle])
            return middle;
        else if(val<a[middle])
            return binarySearch(a,left,middle-1,val);
        else
            return binarySearch(a,middle+1,right,val);
    } else {
        return -1;
    }
}
```

You should notice that in this case, the iterative and recursive algorithms are very similar, and it is not clear that one implementation is better than the other. However, if you were asked to write the algorithm from scratch, it is probably easier to get the details right for the recursive one.

Example 8.38. Prove that the *recursive* `binarySearch` algorithm from Example 8.37 is correct.

Proof: We will prove it by induction on $n = right - left + 1$ (that is, the size of the array).

Base case: If $n = 0$, that means $right < left$, and `binarySearch` returns -1 as it should (since val cannot possible be in an empty array). So it works correctly for $n = 0$.

Inductive Hypothesis: Assume that `binarySearch` works for arrays of size 0 through $k - 1$ (we need strong induction for this proof).

Inductive step: Assume `binarySearch` is called on an array of size k. There are three cases.

- If $val = a[middle]$, the algorithm returns $middle$ which is the correct answer.
- If $val < a[middle]$, a recursive call is made on the first half of the array (from $left$ to $middle - 1$). Because a is sorted, if val is in the array, it is in that half of the array, so we just need to prove that the recursive call returns the correct value. Notice that the first half of the array has less than n elements (it does not contain $middle$ or anything to the right of $middle$, so it is clearly smaller by at least one element). Thus, by the inductive hypothesis, it returns the correct index or -1 if val is not in that part of the array. Therefore it returns the correct value.
- The case for $val > a[middle]$ is symmetric to the previous case and the details are left to the reader.

In all cases, it works correctly on an array of size k.

Summary: Since it works for an array of size 0 and whenever it works for arrays of size at most $k-1$ it works for arrays of size k, by the principle of mathematical induction, it works for arrays of any nonnegative size. $\square$

Note: *You might think the base case in the previous proof should be $n = 1$, but that is not actually correct. A failed search will always make a final call to* `binarySearch` *with $n = 0$. If we don't prove it works for an empty array then we cannot be certain that it works for failed searches.*

Example 8.39. Recall the *Fibonacci sequence*, defined by the recurrence relation

$$f_n = \begin{cases} 0 & \text{if } n{=}0 \\ 1 & \text{if } n{=}1 \\ f_{n-1} + f_{n-2} & \text{if } n > 1. \end{cases}$$

Let's see an iterative and a recursive algorithm to compute f_n. The iterative algorithm (on the left) starts with f_0 and f_1 and computes each f_i based on f_{i-1} and f_{i-2} for i from 2 to n. As it goes, it needs to keep track of the previous two values. The recursive algorithm (on the right) just uses the definition and is pretty straightforward.

```
int Fib(int n) {
    int fib;
    if(n <= 1) {
        return(n);
    } else {
        int fibm2=0;
        int fibm1=1;
        int index=1;
        while(index < n) {
           fib=fibm1+fibm2;
           fibm2=fibm1;
           fibm1=fib;
           index++;
        }
        return(fib);
    }
}
```

```
int FibR(int n) {
    if(n <= 1) {
        return(n);
    } else {
        return(FibR(n-1)+FibR(n-2));
    }
}
```

⋆**Question 8.40.** Which algorithm is better, `Fib` or `FibR`? Give several reasons to justify your answer.

Answer ______________________________

____

____

Although recursion is a great technique to solve many problems, care must be taken when using it. It easy to make simple mistakes like we did in Example 8.35. They can also be very inefficient on occasion, as we alluded to in the previous example (and will prove later). In addition, recursive algorithms often take more memory than iterative ones, as we will see next.

Example 8.41. Consider our algorithms for $n!$. The iterative one from Example 3.40 uses memory to store four numbers: n, f, i, and return value.[a] The recursive one from Example 8.31 uses memory to store two numbers: n and the return value. Although the recursive algorithm uses less memory, it is called multiple times, and every call needs its own memory. For instance, a call to `factorial(3)` will call `factorial(2)` which will call `factorial(1)`. Thus, computing 3! requires enough memory to store 6 numbers, which is more than the 4 required by the iterative algorithm. In general, the recursive algorithm to compute $n!$ will need to store $2n$ numbers, whereas the iterative one will still just need 4, no matter how large n gets.

[a] I won't get technical here, but memory needs to be allocated for the value returned by a function.

Since computers have a finite amount of memory, and since every call to a function requires its own memory, there is a limit to how many recursive calls can be made in practice. In fact some languages, including Java, have a defined limit of how deep the recursion can be. Even for those that don't have a limit, if you run out of memory, you can certainly expect bad things to happen. This is one of the reasons recursion is avoided when possible.

Good compilers attempt to remove recursion, but it is not always possible. Good programmers do the same. Since recursive algorithms are often more intuitive, it often makes sense to think in terms of them. But many recursive algorithms can be turned into iterative algorithms that are as efficient and use less memory. There is no single technique to do so, and it is not always necessary, but it is a good thing to keep in mind.

Let's see a few more examples of the subtle problems that we can run into when using recursion.

Example 8.42. The following algorithm is supposed to sum the numbers from 1 to n:

```
void Sum1toN(int n) {
     if (n == 0)   return(0);
     else          return(n + Sum1toN(n-1));
}
```

Although this algorithm works fine for non-negative values of n, it will go into infinite recursion if $n < 0$. Like our original solution to the `countdown` problem, the mistake here is an *improper base case.*

It is easy to get things backwards when recursion is involved. Consider the following example.

⋆**Question 8.43.** One of these routines prints from 1 up to n, the other from n down to 1. Which does which?

```
void PrintN(int n) {
     if (n > 0) {
        PrintN(n-1);
        print(n);
     }
}
```

```
void NPrint(int n) {
    if (n > 0) {
       print(n);
       NPrint(n-1);
    }
}
```

Answer ____

____

We conclude this section by summarizing some of the advantages and disadvantages of recursion.

The advantages include:

1. Recursion often mimics the way we think about a problem, thus the recursive solutions can be very intuitive to program.

2. Often recursive algorithms to solve problems are much shorter than iterative ones. This can make the code easier to understand, modify, and/or debug.

3. The best known algorithms for many problems are based on a divide-and-conquer approach:

   - Divide the problem into a set of smaller problems
   - Solve each small problem separately
   - Put the results back together for the overall solution

   These divide-and-conquer techniques are often best thought of in terms of recursive algorithms.

Perhaps the main disadvantage of recursion is the extra time and space required. We have already discussed the extra space. The extra time comes from the fact that when a recursive call is made, the operating system has to record how to restart the calling subroutine later on, pass the parameters from the calling subroutine to the called subroutine (often by pushing the parameters onto a stack controlled by the system), set up space for the called subroutine's local variables, etc. The bottom line is that calling a function is not "free".

Another disadvantage is the fact that sometimes a slick-looking recursive algorithm turns out to be very inefficient. We alluded to this in Example 8.40. On the other hand, if such inefficiencies are found, there are techniques that can often easily remove them (e.g. a technique called memoization[2]). But you first have to remember to analyze your algorithm to determine whether or not there might be an efficiency problem.

[2]No, that's not a typo. Google it.

8.3 Solving Recurrence Relations

Recall that a *recurrence relation* is simply a sequence that is recursively defined. More formally, a recurrence relation is a formula that defines a_n in terms of a_i, for one or more values of $i < n$.[3]

Example 8.44. We previously saw that we can define $n!$ by $0! = 1$, and if $n > 0$, $n! = n \cdot (n-1)!$. This is a recurrence relation for the sequence $n!$.

Similarly, we have seen the Fibonacci sequence several times. Recall that n-th Fibonacci number is given by $f_0 = f_1 = 1$ and for $n > 1$, $f_n = f_{n-1} + f_{n-2}$. This is recurrence relation for the sequence of Fibonacci numbers.

Example 8.45. Each of the following are recurrence relations.

$$\begin{aligned} t_n &= n \cdot t_{n-1} + 4 \cdot t_{n-3} \\ r_n &= r_{n/2} + 1 \\ a_n &= a_{n-1} + 2 \cdot a_{n-2} + 3 \cdot a_{n-3} + 4 \cdot a_{n-4} \\ p_n &= p_{n-1} \cdot p_{n-2} \\ s_n &= s_{n-3} + n^2 - 4n + 32 \end{aligned}$$

We have not given any initial conditions for these recurrence relations. Without initial conditions, we cannot compute particular values. We also cannot solve the recurrence relation uniquely.

Recurrence relations have 2 types of terms: *recursive* term(s) and the *non-recursive* terms. In the previous example, the recursive term of s_n is s_{n-3} and the non-recursive term is $n^2 - 4n + 32$.

⋆**Question 8.46.** Consider the recurrence relations r_n and a_n from Example 8.45.

(a) What are the *recursive* terms of r_n?

Answer ____

(b) What are the *non-recursive* terms of r_n?

Answer ____

(c) What are the *recursive* terms of a_n?

Answer ____

(d) What are the *non-recursive* terms of a_n?

Answer ____

In computer science, the most common place we use recurrence relations is to analyze recursive algorithms. We won't get too technical yet, but let's see a simple example.

[3] You might also see recurrence relations written using function notation, like $a(n)$. Although there are technical differences between these notations, you can think of them as being essentially equivalent in this context.

Example 8.47. How many multiplications are required to compute $n!$ using the `factorial` algorithm given in Example 8.31 (repeated below)?

```
int factorial(int n) {
    if(n<=0) {
        return 1;
    } else {
        return n*factorial(n-1);
    }
}
```

Solution: Let M_n be the number of multiplications needed to compute $n!$ using the `factorial` algorithm from Example 8.31. From the code, it is obvious that $M_0 = 0$. If $n > 0$, the algorithm uses one multiplication and then makes a recursive call to `factorial(n-1)`. By the way we defined M_n, `factorial(n-1)` does M_{n-1} multiplications. Therefore, $M_n = M_{n-1} + 1$.

So the recurrence relation for the number of multiplications is

$$M_n = \begin{cases} 0 & \text{if } n{=}0 \\ M_{n-1} + 1 & \text{if } n > 0. \end{cases}$$

Given a recurrence relation for a_n, you can't just plug in n and get an answer. For instance, if $a_n = n \cdot a_{n-1}$, and $a_1 = 1$, what is a_{100}? The only obvious way to compute it is to compute $a_2, a_3, \ldots, a_{99}$, and then finally a_{100}. That is the reason why *solving* recurrence relations is so important. As mentioned previously, solving a recurrence relation simply means finding a *closed form expression* for it.

Example 8.48. It is not too difficult to see that the recurrence from Example 8.47 has the solution $M_n = n$. To prove it, notice that with this assumption, $M_{n-1} + 1 = (n-1) + 1 = n = M_n$, so the solution is consistent with the recurrence relation.

We can also prove it with induction: We know that $M_0 = 0$, so the base case of $k = 0$ is true. Assume $M_k = k$ for $k \geq 0$. Then we have

$$M_{k+1} = M_k + 1 = k + 1,$$

so the formula is correct for $k+1$. Thus, by PMI, the formula is correct for all $k \geq 0$.

The last example demonstrates an important fact about recurrence relations used to analyze algorithms. The recursive terms come from when a recursive function calls itself. The non-recursive terms come from the other work that is done by the function, including any splitting or combining of data that must be done.

Example 8.49. Consider the recursive *binary search* algorithm we saw in Example 8.37:

```
int binarySearch(int[] a, int left, int right, int val) {
    if(right>=left) {
        int middle = (left+right)/2;
        if(val==a[middle])
            return middle;
        else if(val<a[middle])
            return binarySearch(a,left,middle-1,val);
        else
            return binarySearch(a,middle+1,right,val);
    } else {
        return -1;
    }
}
```

Find a recurrence relation for the worst-case complexity of `binarySearch`.

Solution: Let T_n be the complexity of `binarySearch` for an array of size n. Notice that the only things done in the algorithm are to find the middle element, make a few comparisons, perhaps make a recursive call, and return a value. Aside from the recursive call, the amount of work done is constant, which we will just call 1 operation. Notice that at most one recursive call is made, and that the array passed in is half the size. Therefore $T_n = T_{n/2} + 1$.[a] If we want a base case, we can use $T_0 = 1$ since the algorithm will simply return -1 for an empty array, and that clearly takes constant time. We'll see how to solve this recurrence shortly.

[a] Technically, the recurrence relation is $T_n = T_{\lfloor n/2 \rfloor} + 1$ since $n/2$ might not be an integer. It turns out that most of the time we can ignore the floors/ceilings and still obtain the correct answer.

We will discuss using recurrence relations to analyze recursive algorithms in more detail in section 8.4. But first we will discuss how to solve recurrence relations. There is no general method to solve recurrences. There are many strategies, however. In the next few sections we will discuss four common techniques: the *substitution method*, the *iteration method*, the *Master Theorem* (or *Master Method*), and the *characteristic equation method* for linear recurrences.

⋆**Question 8.50.** Let's see if you have been paying attention. What does it mean to solve a recurrence relation?

Answer ____

____

As we continue our discussion of recurrence relations, you will notice that we will begin to sometimes use the function notation (e.g. $T(n)$ instead of T_n). We do this for several reasons. The first is so that you are comfortable with either notation. The second is that in algorithm analysis, this notation seems to be more common, at least in my experience.

8.3.1 Substitution Method

The *substitution method* might be better called the *guess and prove it by induction method*. Why? Because to use it, you first have to figure out what you think the solution is, and then you need to actually prove it. Because of the close tie between recurrence relations and induction, it is the most natural technique to use. Let's see an example.

Example 8.51. Consider the recurrence

$$S(n) = \begin{cases} 1 & \text{when } n = 1 \\ S(n-1) + n & \text{otherwise} \end{cases}$$

Prove that the solution is $S(n) = \dfrac{n(n+1)}{2}$.

Proof: When $n = 1$, $S(1) = 1 = \frac{1(1+1)}{2}$. Assume that $S(k-1) = \frac{(k-1)k}{2}$. Then

$$\begin{aligned} S(k) &= S(k-1) + k \quad \text{(Definition of } S(k)\text{)} \\ &= \frac{(k-1)(k)}{2} + k \quad \text{(Inductive hypothesis)} \\ &= \frac{k^2 - k}{2} + k \quad \text{(The rest is just algebra)} \\ &= \frac{k^2 - k + 2k}{2} \\ &= \frac{k^2 + k}{2} \\ &= \frac{k(k+1)}{2}. \end{aligned}$$

By PMI, $S(n) = \frac{n(n+1)}{2}$ for all $n \geq 1$. □

⋆**Exercise 8.52.** Recall that in Example 8.49, we developed the recurrence relation $T(n) = T(n/2) + 1, T(0) = 1$ for the complexity of `binarySearch`. For technical reasons, ignore $T(0)$ and assume $T(1) = 1$ is the base case. Use substitution to prove that $T(n) = \log_2 n + 1$ is a solution to this recurrence relation.

Example 8.53. Solve the recurrence

$$H_n = \begin{cases} 1 & \text{when } n = 1 \\ 2H_{n-1} + 1 & \text{otherwise} \end{cases}$$

Proof: Notice that $H_1 = 1$, $H_2 = 2 \cdot 1 + 1 = 3$, $H_3 = 2 \cdot 3 + 1 = 7$, and $H_4 = 2 \cdot 7 + 1 = 15$. It sure looks like $H_n = 2^n - 1$, but now we need to prove it. Since $H_1 = 1 = 2^1 - 1$, we have our base case of $n = 1$. Assume $H_n = 2^n - 1$. Then

$$\begin{aligned} H_{n+1} &= 2H_n + 1 \\ &= 2(2^n - 1) + 1 \\ &= 2^{n+1} - 1, \end{aligned}$$

and the result follows by induction. □

⋆**Exercise 8.54.** Solve the following recurrence relation and use induction to prove your solution is correct: $A(n) = A(n-1) + 2$, $A(1) = 2$.

Example 8.55. Why was the recursive algorithm to compute f_n from Example 8.39 so bad?

Solution: Let's count the number of additions `FibR(n)` computes since that is the main thing that the algorithm does.[a] Let $F(n)$ be the number of additions required to compute f_n using `FibR(n)`. Since `FibR(n)` calls `FibR(n-1)` and `FibR(n-2)` and then performs one addition, it is easy to see that

$$F(n) = F(n-1) + F(n-2) + 1,$$

where $F(0) = F(1) = 0$ is clear from the algorithm. We could use the method for linear recurrences that will be outlined later to solve this, but the algebra gets a bit messy. Instead, Let's see if we can figure it out by computing some values.

$$\begin{aligned}
F(0) &= 0 \\
F(1) &= 0 \\
F(2) &= F(1) + F(0) + 1 = 1 \\
F(3) &= F(2) + F(1) + 1 = 2 \\
F(4) &= F(3) + F(2) + 1 = 4 \\
F(5) &= F(4) + F(3) + 1 = 7 \\
F(6) &= F(5) + F(4) + 1 = 12 \\
F(7) &= F(6) + F(5) + 1 = 20
\end{aligned}$$

No pattern is evident unless you add one to each of these. If you do, you will get $1, 1, 2, 3, 5, 8, 13, 21$, etc., which looks a lot like the Fibonacci sequence starting with f_1. So it appears $F(n) = f_{n+1} - 1$. To verify this, first notice that $F(0) = 0 = f_1 - 1$ and $F(1) = 0 = f_2 - 1$. Assume it holds for all values less than k. Then

$$\begin{aligned}
F(k) &= F(k-1) + F(k-2) + 1 \\
&= f_k - 1 + f_{k-1} - 1 + 1 \\
&= f_k + f_{k-1} - 1 \\
&= f_{k+1} - 1.
\end{aligned}$$

The result follows by induction.

So what does this mean? It means in order to compute f_n, `FibR(n)` performs $f_{n+1} + 1$ additions. In other words, it computes f_n by adding a bunch of 0s and 1s, which doesn't seem very efficient. Since f_n grows exponentially (we'll see this in Example 8.80), then $F(n)$ does as well. That pretty much explains what is wrong with the recursive algorithm.

[a] Alternatively, we could count the number of recursive calls made. This is reasonable since the amount of work done by the algorithm, aside from the recursive calls, is constant. Therefore, the time it takes to compute f_n is proportional to the number of recursive calls made. This would produce a slightly different answer, but they would be comparable.

8.3.2 Iteration Method

With the iteration method (sometimes called *backward substitution*, we expand the recurrence and express it as a summation dependent only on n and initial conditions. Then we evaluate the summation. Sometimes the closed form of the sum is obvious as we are iterating (so no actual summation appears in our work), while at other times it is not (in which case we *do* end up with an actual summation).

Our first example perhaps has too many steps of algebra, but it never hurts to be extra careful when doing so much algebra. We also don't provide a whole lot of justification or explanation for the steps. We will do that in the next example. It is easier to see the overall idea of the iteration method if we don't interrupt it with comments. If this example does not make sense, come back to it after reading the next example.

Example 8.56. Solve the recurrence

$$R(n) = \begin{cases} 1 & \text{when } n = 1 \\ 2R(n/2) + n/2 & \text{otherwise} \end{cases}$$

Proof: We have

$$\begin{aligned}
R(n) &= 2R(n/2) + n/2 \\
&= 2(2R(n/4) + n/4) + n/2 \\
&\quad = 2^2R(n/4) + n/2 + n/2 \\
&\quad = 2^2R(n/4) + n \\
&= 2^2(2R(n/8) + n/8) + n \\
&\quad = 2^3R(n/8) + n/2 + n \\
&\quad = 2^3R(n/8) + 3n/2 \\
&= 2^3(2R(n/16) + n/16) + 3n/2 \\
&\quad = 2^4R(n/16) + n/2 + 3n/2 \\
&\quad = 2^4R(n/16) + 2n \\
&\vdots \\
&= 2^kR(n/(2^k)) + kn/2 \\
&= 2^{\log_2 n}R(n/(2^{\log_2 n})) + (\log_2 n)n/2 \\
&= nR(n/n) + (\log_2 n)n/2 \\
&= nR(1) + (\log_2 n)n/2 \\
&= n + (\log_2 n)n/2
\end{aligned}$$

□

Using this method requires a little abstract thinking and pattern recognition. It also requires good algebra skills. Care must be taken when doing algebra, especially with the non-recursive terms. Sometimes you should add/multiply (depending on context) them all together, and other times you should leave them as is. The problem is that it takes experience (i.e. practice) to determine which one is better in a given situation. The key is flexibility. If you try doing it one way and don't see a pattern, try another way.

Here is my suggestion for using this method

1. Iterate enough times so you are certain of what the pattern is. Typically this means at least 3 or 4 iterations.

2. As you iterate, make adjustments to your algebra as necessary so you can see the pattern. For instance, whether you write 2^3 or 8 can make a difference in seeing the pattern.

3. Once you see the pattern, generalize it, writing what it should look like after k iterations.

4. Determine the value of k that will get you to the base case, and then plug it in.

5. Simplify.

⋆**Question 8.57.** The *iteration method* is probably not a good choice to solve the following recurrence relation. Explain why.

$$T(n) = T(n-1) + 3T(n-2) + n * T(n/3) + n^2, \;\; T(1) = 17$$

Answer ____

____

____

Here is an example that contains more of an explanation of the technique.

Example 8.58. Solve the recurrence relation $T(n) = 2T(n/2) + n^3$, $T(1) = 1$.

Solution: We start by backward substitution:

$$\begin{aligned} T(n) &= 2T(n/2) + n^3 \\ &= 2[2T(n/4) + (n/2)^3] + n^3 \\ &= 2[2T(n/4) + n^3/8)] + n^3 \\ &= 2^2T(n/4) + n^3/4 + n^3 \end{aligned}$$

Notice that in the second line we have $(n/2)^3$ and not n^3. This may be more clear if rewrite the formula using k: $T(k) = 2T(k/2) + k^3$. When applying the formula to $T(n/2)$, we have $k = n/2$, so we get

$$T(n/2) = 2T((n/2)/2) + (n/2)^3 = 2T(n/4) + n^3/8.$$

Back to the second line, also notice that the 2 is multiplied by both the $2T(n/4)$ and the $(n/2)^3$ terms. A common error is to lose one of the 2s on the $T(n/4)$ term or miss it on the $(n/2)^3$ term when simplifying. Also, $(n/2)^3 = n^3/8$, not $n^3/2$. This is another common mistake. Continuing,

$$\begin{aligned}
T(n) &= \ldots \\
&= 2^2T(n/4) + n^3/4 + n^3 \\
&= 2^2[2T(n/8) + (n/4)^3] + n^3/4 + n^3 \\
&= 2^2[2T(n/8) + n^3/4^3] + n^3/4 + n^3 \\
&= 2^3T(n/8) + n^3/4^2 + n^3/4 + n^3.
\end{aligned}$$

By now you should have noticed that I use 2 or more steps for every iteration–I do one substitution and then simplify it before moving on to the next substitution. This helps to ensure I don't make algebra mistakes and that I can write it out in a way that helps me see a pattern.

Next, notice that we can write the last line as

$$2^3T(n/2^3) + n^3/4^2 + n^3/4^1 + n^3/4^0,$$

so it appears that we can generalize this to

$$2^kT(n/2^k) + \sum_{i=0}^{k-1} n^3/4^i.$$

The sum starts at $i = 0$ (not 1) and goes to $k - 1$ (not k). It is easy to get either (or both) of these wrong if you aren't careful. We should be careful to make sure we have seen the correct pattern. Too often I have seen students make a pattern out of 2 iterations. Not only is this not enough iterations to be sure of anything, the pattern they usually come up with only holds for the last iteration they did. The pattern has to match *every* iteration. To be safe, go one more iteration after you identify the pattern to verify that it is correct.

Continuing (with a few more steps shown to make all of the algebra as clear as possible), we get

$$\begin{aligned}
T(n) &= \ldots \\
&= 2^3T(n/2^3) + n^3/4^2 + n^3/4^1 + n^3/4^0 \\
&= 2^3[2T(n/2^4) + (n/2^3)^3] + n^3/4^2 + n^3/4^1 + n^3/4^0 \\
&= 2^3[2T(n/2^4) + n^3/2^9] + n^3/4^2 + n^3/4^1 + n^3/4^0 \\
&= 2^4T(n/2^4) + n^3/2^6 + n^3/4^2 + n^3/4^1 + n^3/4^0 \\
&= 2^4T(n/2^4) + n^3/4^3 + n^3/4^2 + n^3/4^1 + n^3/4^0 \\
&= \ldots \\
&= 2^kT(n/2^k) + \sum_{i=0}^{k-1} n^3/4^i.
\end{aligned}$$

Notice that this *does* seem to match the pattern we saw above. We can evaluate the sum to simplify it a little more:

$$\begin{aligned}
T(n) &= \dots \\
&= 2^k T(n/2^k) + \sum_{i=0}^{k-1} n^3/4^i \\
&= 2^k T(n/2^k) + n^3 \sum_{i=0}^{k-1} 1/4^i \\
&= 2^k T(n/2^k) + n^3 \sum_{i=0}^{k-1} (1/4)^i \\
&= 2^k T(n/2^k) + n^3 \left(\frac{1-(1/4)^k}{1-1/4} \right) \\
&= 2^k T(n/2^k) + n^3 (4/3)(1-(1/4)^k)
\end{aligned}$$

We are almost done. We just need to find a k that allows us to get rid of the recursion. Thus, we need to determine what value of k makes $T(n/2^k) = T(1) = 1$. In other words, we need k such that

$$n/2^k = 1.$$

This is equivalent to

$$n = 2^k.$$

Taking log (base 2) of both sides, we obtain

$$\log_2 n = \log_2(2^k) = k \log_2 2 = k.$$

So $k = \log_2 n$. We plug in k and use the fact that $2^{\log_2 n} = n$ along with the exponent rules to obtain

$$\begin{aligned}
T(n) &= \dots \\
&= 2^k T(n/2^k) + n^3 (4/3)(1-(1/4)^k) \\
&= 2^{\log_2 n} T(n/2^{\log_2 n}) + n^3 (4/3)(1-(1/4)^{\log_2 n}) \\
&= nT(1) + n^3(4/3) \left(1 - \frac{1}{(2^2)^{\log_2 n}} \right) \\
&= n \cdot 1 + n^3(4/3) \left(1 - \frac{1}{(2^{\log_2 n})^2} \right) \\
&= n + n^3(4/3) \left(1 - \frac{1}{n^2} \right) \\
&= n + \frac{4}{3}n^3 - \frac{4}{3}n \\
&= \frac{4}{3}n^3 - \frac{1}{3}n.
\end{aligned}$$

So we have that $T(n) = \frac{4}{3}n^3 - \frac{1}{3}n$.

⋆**Exercise 8.59.** Use iteration to solve the recurrence

$$H(n) = \begin{cases} 1 & \text{when } n = 1 \\ 2H(n-1)+1 & \text{otherwise} \end{cases}$$

Example 8.60. Give a tight bound for the recurrence $T(n) = T(\sqrt{n}) + 1$, where $T(2) = 1$.

Solution: We can see that

$$\begin{aligned} T(n) &= T(n^{1/2}) + 1 \\ &= T(n^{1/4}) + 1 + 1 \\ &= T(n^{1/8}) + 1 + 1 + 1 \\ &= T(n^{1/2^k}) + k \end{aligned}$$

If we can determine when $n^{1/2^k} = 2$, we can obtain a solution. Taking logs (base 2) on both sides, we get

$$\log_2(n^{1/2^k}) = \log_2 2.$$

We apply the power-inside-a-log rule and the fact that $\log_2 2 = 1$ to get

$$(1/2^k) \log_2 n = 1.$$

Multiplying both sides by 2^k and flipping it around, we get

$$2^k = \log_2 n.$$

Again taking logs, we get

$$k = \log_2 \log_2 n.$$

Therefore,

$$\begin{aligned} T(n) &= T(n^{1/2^{\log_2 \log_2 n}}) + \log_2 \log_2 n \\ &= T(2) + \log_2 \log_2 n \ \ (\text{since } n^{1/2^{\log_2 \log_2 n}} = 2 \text{ by the way we chose } k) \\ &= 1 + \log_2 \log_2 n. \end{aligned}$$

Therefore, $T(n) = 1 + \log_2 \log_2 n$.

⋆**Exercise 8.61.** Use iteration to solve the recurrence relation that we developed in Example 8.49 for the complexity of `binarySearch`:

$$T(n) = T(n/2) + 1, T(1) = 1.$$

If you can do the following exercise correctly, then you have a firm grasp of the iteration method and your algebra skills are superb. If you have difficulty, keep working at it and/or get some assistance. I strongly recommend that you do your best to solve this one on your own.

⋆**Exercise 8.62.** Solve the recurrence relation $T(n) = 2T(n-1) + n$, $T(1) = 1$. (Hint: You will need the result from Exercise 8.16.)

8.3.3 Master Theorem

We will omit the proof of the following theorem which is particularly useful for solving recurrence relations that result from the analysis of certain types of recursive algorithms–especially divide-and-conquer algorithms.

Theorem 8.63 (Master Theorem)**.** *Let $T(n)$ be a monotonically increasing function satisfying*

$$\begin{aligned} T(n) &= aT(n/b) + f(n) \\ T(1) &= c \end{aligned}$$

where $a \geq 1$, $b > 1$, and $c > 0$. If $f(n) = \theta(n^d)$, where $d \geq 0$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Example 8.64. Use the Master Theorem to solve the recurrence

$$T(n) = 4T(n/2) + n, T(1) = 1.$$

Solution: We have $a = 4, b = 2$, and $d = 1$. Since $4 > 2^1$, $T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$ by the third case of the Master Theorem.

Example 8.65. Use the Master Theorem to solve the recurrence

$$T(n) = 4T(n/2) + n^2, T(1) = 1.$$

Solution: We have $a = 4, b = 2$, and $d = 2$. Since $4 = 2^2$, we have $T(n) = \Theta(n^2 \log n)$ by the second case of the Master Theorem.

Example 8.66. Use the Master Theorem to solve the recurrence

$$T(n) = 4T(n/2) + n^3, T(1) = 1.$$

Solution: Here, $a = 4, b = 2$, and $d = 3$. Since $4 < 2^3$, we have $T(n) = \Theta(n^3)$ by the first case of the Master Theorem.

Wow. That was easy.[4] But the ease of use of the Master Method comes with a cost. Well, two actually. First, notice that we do not get an *exact* solution, but only an *asymptotic bound* on the solution. Depending on the context, this may be good enough. If you need an exact numerical solution, the Master Method will do you no good. But when analyzing algorithms, typically we are more interested in the asymptotic behavior. In that case, it works great. Second, it only works for recurrences that have the exact form $T(n) = aT(n/b) + f(n)$. It won't even work on similar recurrence, such as $T(n) = T(n/b) + T(n/c) + f(n)$.

[4] Almost *too* easy.

⋆**Exercise 8.67.** Use the Master Theorem to solve the recurrence

$$T(n) = 2T(n/2) + 1, T(1) = 1.$$

Example 8.68. Let's redo one from a previous section. Use the Master Theorem to solve the recurrence

$$R(n) = \begin{cases} 1 & \text{when } n = 1 \\ 2R(n/2) + n/2 & \text{otherwise} \end{cases}$$

Solution: Here, we have $a = 2$, $b = 2$, and $d = 1$. Since $2 = 2^1$, $R(n) = \Theta(n^1 \log n) = \Theta(n \log n)$. Recall that in Example 8.56 we showed that $R(n) = n + (\log_2 n)n/2$. Since $n + (\log_2 n)n/2 = \Theta(n \log n)$, our solution is consistent.

⋆**Exercise 8.69.** Use the Master Theorem to solve the recurrence

$$T(n) = 7T(n/2) + 15n^2/4, T(1) = 1.$$

⋆**Question 8.70.** In the solution to the previous exercise, we stated that

'$T(n) = \Theta(n^{\log_2 7})$, which is about $\Theta(n^{2.8})$.'

Why didn't we just say '$T(n) = \Theta(n^{\log_2 7}) = \Theta(n^{2.8})$'?

Answer ____

____

⋆**Exercise 8.71.** We saw in Example 8.49 that the complexity of binary search is given by the recurrence relation $T(n) = T(n/2) + 1$, $T(0) = 1$ (and you may assume that $T(1) = 1$). Use the Master Theorem to solve this recurrence.

8.3.4 Linear Recurrence Relations

Although in my mind linear recurrence relations are of the least importance of these four methods for computer scientists, we will discuss them very briefly, both for completeness sake, and because we can talk about the Fibonacci numbers again.

Definition 8.72. *Let $c_1, c_2, \ldots, c_k$ be real constants and $f : \mathbb{N} \to \mathbb{R}$ a function. A recurrence relation of the form*

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k} + f(n) \tag{8.1}$$

is called a **linear recurrence relation** *(or* **linear difference equation***). If $f(n) = 0$ (that is, there is no non-recursive term), we say that the equation is* **homogeneous***, and otherwise we say the equation is* **nonhomogeneous***.*

The *order* of the recurrence is the difference between the highest and the lowest subscripts.

Example 8.73. $u_n = u_{n-1} + 2$ is of the first order, and $u_n = 9u_{n-4} + n^5$ is of the fourth order.

There is a general technique that can be used to solve linear homogeneous recurrence relations. However, we will restrict our discussion to certain first and second order recurrences.

First Order Recurrences

In this section we will learn a technique to solve some first-order recurrences. We won't go into detail about why the technique works.

Procedure 8.74. *Let $f(n)$ be a polynomial and $a \neq 1$. Then the following technique can be used to solve a first order linear recurrence relations of the form*

$$x_n = a x_{n-1} + f(n).$$

1. *First, ignore $f(n)$. That is, solve the homogeneous recurrence $x_n = a x_{n-1}$. This is done as follows:*

   (a) 'Raise the subscripts', so $x_n = a x_{n-1}$ becomes $x^n = a x^{n-1}$. This is called the

characteristic equation.

(b) *Canceling this gives $x = a$.*

(c) *The solution to the homogeneous equation $x_n = ax_{n-1}$ will be of the form $x_n = Aa^n$, where A is a constant to be determined.*

2. *Assume that the solution to the original recurrence relation, $x_n = ax_{n-1} + f(n)$, is of the form $x_n = Aa^n + g(n)$, where g is a polynomial of the same degree as $f(n)$.*

3. *Plug in enough values to determine the correct constants for the coefficients of $g(n)$.*

This procedure is a bit abstract, so let's just jump into seeing it in action.

Example 8.75. Let $x_0 = 7$ and $x_n = 2x_{n-1}, n \geq 1$. Find a closed form for x_n.

Solution: Raising subscripts we have the characteristic equation $x^n = 2x^{n-1}$. Canceling, $x = 2$. Thus we try a solution of the form $x_n = A2^n$, were A is a constant. But $7 = x_0 = A2^0 = A$ and so $A = 7$. The solution is thus $x_n = 7(2)^n$.

Example 8.76. Let $x_0 = 7$ and $x_n = 2x_{n-1} + 1, n \geq 1$. Find a closed form for x_n.

Solution: By raising the subscripts in the homogeneous equation we obtain $x^n = 2x^{n-1}$ or $x = 2$. A solution to the homogeneous equation will be of the form $x_n = A(2)^n$. Now $f(n) = 1$ is a polynomial of degree 0 (a constant) and so the general solution should have the form $x_n = A2^n + B$. Now, $7 = x_0 = A2^0 + B = A + B$. Also, $x_1 = 2x_0 + 1 = 15$ and so $15 = x_1 = 2A + B$. Solving the simultaneous equations

$$A + B = 7,$$

$$2A + B = 15,$$

Using these equations, we can see that $A = 7 - B$ and $B = 15 - 2A$. Plugging the latter into the former, we have $A = 7 - (15 - 2A) = -8 + 2A$, or $A = 8$. Plugging this back into either equation, we can see that $B = -1$. So the solution is $x_n = 8(2^n) - 1 = 2^{n+3} - 1$.

$\star$**Exercise 8.77.** Let $x_0 = 2, x_n = 9x_{n-1} - 56n + 63$. Find a closed form for this recurrence.

Second Order Recurrences

Let us now briefly examine how to solve some second order recursions.

Procedure 8.78. *Here is how to solve a second-order homogeneous linear recurrence relations of the form*

$$x_n = ax_{n-1} + bx_{n-2}.$$

1. *Find the characteristic equation by "raising the subscripts." We obtain* $x^n = ax^{n-1} + bx^{n-2}$.
2. *Canceling this gives* $x^2 - ax - b = 0$. *This equation has two roots* r_1 *and* r_2.
3. *If the roots are different, the solution will be of the form* $x_n = A(r_1)^n + B(r_2)^n$, *where* A, B *are constants.*
4. *If the roots are identical, the solution will be of the form* $x_n = A(r_1)^n + Bn(r_1)^n$.

Example 8.79. Let $x_0 = 1, x_1 = -1, x_{n+2} + 5x_{n+1} + 6x_n = 0$.

Solution: The characteristic equation is $x^2 + 5x + 6 = (x+3)(x+2) = 0$. Thus we test a solution of the form $x_n = A(-2)^n + B(-3)^n$. Since $1 = x_0 = A + B$, and $-1 = -2A - 3B$, we quickly find $A = 2$, and $B = -1$. Thus the solution is $x_n = 2(-2)^n - (-3)^n$.

Example 8.80. Find a closed form for the Fibonacci recurrence $f_0 = 0, f_1 = 1, f_n = f_{n-1} + f_{n-2}$.

Solution: The characteristic equation is $f^2 - f - 1 = 0$. This has roots $\frac{1\pm\sqrt{5}}{2}$. Therefore, a solution will have the form

$$f_n = A\left(\frac{1+\sqrt{5}}{2}\right)^n + B\left(\frac{1-\sqrt{5}}{2}\right)^n.$$

The initial conditions give

$$0 = A + B, \text{ and}$$

$$1 = A\left(\frac{1+\sqrt{5}}{2}\right) + B\left(\frac{1-\sqrt{5}}{2}\right) = \frac{1}{2}(A+B) + \frac{\sqrt{5}}{2}(A-B) = \frac{\sqrt{5}}{2}(A-B).$$

From these two equations, we obtain $A = \frac{1}{\sqrt{5}}, B = -\frac{1}{\sqrt{5}}$. We thus have

$$f_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n.$$

⋆**Exercise 8.81.** Find a closed form for the recurrence $x_0 = 1, x_1 = 4, x_n = 4x_{n-1} - 4x_{n-2}$.

8.4 Analyzing Recursive Algorithms

In Section 8.3 we already saw a few examples of analyzing recursive algorithms. We will provide a few more examples in this section. In case it isn't clear, the most common method to analyze a recursive algorithm is to develop and solve a recurrence relation for its running time. Let's see some examples.

Example 8.82. What is the worst-case running time of Mergesort?

Solution: The algorithm for `Mergesort` is below. Let $T(n)$ be the worst-case running time of `Mergesort` on an array of size $n = right - left$. Recall that `Merge` takes two sorted arrays and merges them into one sorted array in time $\Theta(n)$, where n is the number of elements in both arrays.[a] Since the two recursive calls to `Mergesort` are on arrays of half the size, they each require time $T(n/2)$ in the worst-case. The other operations take constant time. Below we annotate the `Mergesort` algorithm with these running times.

Algorithm	Time required
`Mergesort(int[] A,int left,int right) {`	$T(n)$
`    if (left < right) {`	C_1
`        int mid = (left + right)/2;`	C_2
`        Mergesort(A, left, mid);`	$T(n/2)$
`        Mergesort(A, mid + 1, right);`	$T(n/2)$
`        Merge(A, left, mid, right);`	$\Theta(n) \leq C_3 n$
`    }`	
`}`	

Given this, we can see that

$$\begin{aligned} T(n) &= C_1 + C_2 + T(n/2) + T(n/2) + \Theta(n) \\ &= 2T(n/2) + \Theta(n). \end{aligned}$$

Notice that we absorbed the constants C_1 and C_2 into the $\Theta(n)$ term. For simplicity, we will also replace the $\Theta(n)$ term with cn (where c is a constant) and rewrite this as

$$T(n) = 2T(n/2) + cn.$$

We could use the Master Theorem to prove that $T(n) = \Theta(n \log n)$, but that would be too easy. Instead, we will use induction to prove that $T(n) = O(n \log n)$, and leave the Ω-bound to the reader.

By definition, $T(n) = O(n \log n)$ if and only if there exists constants k and n_0 such that $T(n) \leq kn \log n$ for all $n \geq n_0$.

For the base case, notice that $T(2) = a$ for some constant a, and $a \leq k2 \log 2 = 2k$ as long as we pick $k \geq a/2$. Now, assume that $T(n/2) \leq k(n/2)\log(n/2)$. Then

$$\begin{aligned}
T(n) &= 2T(n/2) + cn \\
&\leq 2(k(n/2)\log(n/2) + cn \\
&= kn\log(n/2) + cn \\
&= kn\log n - kn\log 2 + cn \\
&= kn\log n + (c-k)n \\
&\leq kn\log n \quad \text{if } k \geq c
\end{aligned}$$

As long as we pick $k = \max\{a/2, c\}$, we have $T(n) \leq kn\log n$, so $T(n) = O(n\log n)$ as desired.

[a] Since our goal here is to analyze the algorithm, we won't provide a detailed implementation of `Merge`. All we need to know is its complexity.

⋆**Exercise 8.83.** We stated in the previous example that we could use the Master Theorem to prove that if $T(n) = 2T(n/2) + cn$, then $T(n) = \Theta(n\log n)$. Verify this.

⋆**Question 8.84.** Answer the following questions about points that were made in Example 8.82.

(a) Why were we allowed to absorb the constants C_1 and C_2 into the $\Theta(n)$ term?

Answer ____

(b) Why were we able to replace the $\Theta(n)$ term with cn?

Answer ____

Example 8.85 (Towers of Hanoi)**.** The following legend is attributed to French mathematician Edouard Lucas in 1883. In an Indian temple there are 64 gold disks resting on three pegs. At the beginning of time, God placed these disks on the first peg and ordained that a group of priests should transfer them to the third peg according to the following rules:

1. The disks are initially stacked on peg A, in decreasing order (from bottom to top).
2. The disks must be moved to another peg in such a way that only one disk is moved at a time and without stacking a larger disk onto a smaller disk.

When they finish, the Tower will crumble and the world will end. How many moves does it take to solve the *Towers of Hanoi* problem with n disks?

Solution: The usual (and best) algorithm to solve the *Towers of Hanoi* is:

- Move the top $n-1$ disk to from peg 1 to peg 2.
- Move the last disk from peg 1 to peg 3.
- Move the top $n-1$ disks from peg 2 to peg 3.

The only question is how to move the top $n-1$ disks. The answer is simple: use recursion but switch the peg numbers. Here is an implementation of this idea:

```
void solveHanoi(int N,int source,int dest,int spare) {
    if(N==1) {
        moveDisk(source,dest);
    } else {
        solveHanoi(N-1,source,spare,dest);
        moveDisk(source,dest);
        solveHanoi(N-1,spare,dest,source);
    }
}
```

Don't worry if you don't see why this algorithm works. Our main concern here is analyzing the algorithm.

The exact details of `moveDisk` depend on how the pegs/disks are implemented, so we won't provide an implementation of it. But it doesn't actually matter anyway since we just need to count the number of times `moveDisk` is called. As it turns out, any reasonable implementation of `moveDisk` will take constant time, so the complexity of the algorithm is essentially the same as the number of calls to `moveDisk`.

Let $H(n)$ be the number of moves it takes to solve the *Towers of Hanoi* problem with n disks. Then $H(n)$ is the number of times `moveDisk` is called when running `solveHanoi(n,1,2,3)`. It should be clear that $H(1) = 1$ since the algorithm simply makes a single call to `moveDisk` and quits. When $n > 1$, the algorithm makes two calls to `solveHanoi` with the first parameter being $n-1$ and one call to `moveDisk`. Therefore, we can see that

$$H(n) = 2H(n-1) + 1.$$

As with the first example, we want a closed form for $H(n)$. But we already showed that $H(n) = 2^n - 1$ in Examples 8.53 and 8.59.

⋆**Exercise 8.86.** Let $T(n)$ be the complexity of `blarg(n)`. Give a recurrence relation for $T(n)$.

```
int blarg(int n) {
     if(n>5) {
        return blarg(n-1)+blarg(n-1)+blarg(n-5)+blarg(sqrt(n));
     }
     else {
        return n;
     }
}
```

Answer ____________________

⋆**Exercise 8.87.** Give a recurrence relation for the running time of `stoogeSort(A,0,n-1)`. (Hint: Start by letting $T(n)$ be the running time of `stoogeSort` on an array of size n.)

```
void stoogeSort(int[] A,int L,int R){
    if(R<=L) return; // Array has at most one element
    if(A[R]<A[L]) {  // Swap first and last element
        Swap(A,L,R); // if they are out of order
    }
    if(R-L>1){ // If the list has at least 2 elements
        int third=(R-L+1)/3;
        stoogeSort(A,L,R-third); // Sort first two-thirds
        stoogeSort(A,L+third,R); // Sort last two-thirds
        stoogeSort(A,L,R-third); // Sort first two-thirds again
    }
}
```

Answer ____________________

⋆**Exercise 8.88.** Solve the recurrence relation you developed for `StoogeSort` in the previous exercise. (Make sure you verify your solution to the previous problem before you attempt to solve your recurrence relation).

⋆**Question 8.89.** Which sorting algorithm is faster, `Mergesort` or `StoogeSort`? Justify your answer.

Answer ____

____

____

⋆**Exercise 8.90.** Give and solve a recurrence relation for the running time of an algorithm that does as follows: The algorithm is given an input array of size n. If $n < 3$, the algorithm does nothing. If $n \geq 3$, create 5 separate arrays, each one-third of the size of the original array. This takes $\Theta(n)$ to accomplish. Then call the same algorithm on each of the 5 arrays.

8.4.1 Analyzing Quicksort

In this section we give a proof that the average case running time of randomized quicksort is $\Theta(n \log n)$. This proof gets its own section because the analysis is fairly involved. This proof is based on the one presented in Section 8.4 of the classic *Introduction to Algorithms* by Cormen, Leiserson, and Rivest. The algorithm they give is slightly different, and they include some interesting insights, so read their proof/discussion if you get a chance.

There are several slight variations of the quicksort algorithm, and although the exact running times are different for each, the asymptotic running times are all the same. Below is the version of `Quicksort` we will analyze.

Example 8.91. Here is one implementation of `Quicksort`:

```
Quicksort(int A[],int l,int r)
{
   if (r > l) {
     int p = RPartition(A,l,r);
     Quicksort(A,l,p-1);
     Quicksort(A,p+1,r);
     }
}
```

```
int RPartition(int A[],int l,int r)
{
    int piv=l+(rand()%(r-l+1));
    swap(A,l,piv);
    int i = l+1;
    int j = r;
    while (1) {
        while (A[i] <= A[l] && i<r)
            i++;
        while (A[j] >= A[l] && j>l)
            j--;
        if (i >= j) {
           swap(A,j,l);
           return j;
           }
        else swap(A,i,j);
        }
}
```

We will base our analysis on this version of `Quicksort`. It is straightforward to see that the runtime of `RPartition` is $\Theta(n)$ (Problem 8.11 asks you to prove this). We start by developing a recurrence relation for the average case runtime of `Quicksort`.

Theorem 8.92. *Let $T(n)$ be the average case runtime of* `Quicksort` *on an array of size n. Then*

$$T(n) = \frac{2}{n}\sum_{k=1}^{n-1} T(k) + \Theta(n).$$

Proof: *Since the pivot element is chosen randomly, it is equally likely that the pivot will end up at any position from l to r. That is, the probability that the pivot ends up at location $l+i$ is $1/n$ for each $i = 0, \ldots, r-l$. If we average over all of the possible pivot locations, we obtain*

$$\begin{aligned}
T(n) &= \frac{1}{n}\left(\sum_{k=0}^{n-1}\left(T(k)+T(n-k-1)\right)\right) + \Theta(n) \\
&= \frac{1}{n}\sum_{k=0}^{n-1} T(k) + \frac{1}{n}\sum_{k=0}^{n-1} T(n-k-1) + \Theta(n) \\
&= \frac{1}{n}\sum_{k=0}^{n-1} T(k) + \frac{1}{n}\sum_{k=0}^{n-1} T(k) + \Theta(n) \\
&= \frac{2}{n}\sum_{k=0}^{n-1} T(k) + \Theta(n) \\
&= \frac{2}{n}\sum_{k=1}^{n-1} T(k) + \Theta(n).
\end{aligned}$$

The last step holds since $T(0) = 0$. ☐

We will need the following result in order to solve the recurrence relation.

Lemma 8.93. *For any $n \geq 3$,*

$$\sum_{k=2}^{n-1} k \log k \leq \frac{1}{2}n^2 \log n - \frac{1}{8}n^2.$$

Proof: *We can write the sum as*

$$\sum_{k=2}^{n-1} k \log k = \sum_{k=2}^{\lceil n/2 \rceil - 1} k \log k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \log k$$

Then we can bound $(k \log k)$ by $(k \log(n/2)) = k(\log n - 1)$ in the first sum, and by $(k \log n)$ in the second sum. This gives

$$\begin{aligned}
\sum_{k=2}^{n-1} k \log k &= \sum_{k=2}^{\lceil n/2 \rceil - 1} k \log k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \log k \\
&\leq \sum_{k=2}^{\lceil n/2 \rceil - 1} k(\log n - 1) + \sum_{k=\lceil n/2 \rceil}^{n-1} k \log n \\
&= (\log n - 1) \sum_{k=2}^{\lceil n/2 \rceil - 1} k + \log n \sum_{k=\lceil n/2 \rceil}^{n-1} k \\
&= \log n \sum_{k=2}^{\lceil n/2 \rceil - 1} k - \sum_{k=2}^{\lceil n/2 \rceil - 1} k + \log n \sum_{k=\lceil n/2 \rceil}^{n-1} k \\
&= \log n \sum_{k=2}^{n-1} k - \sum_{k=2}^{\lceil n/2 \rceil - 1} k \\
&\leq \log n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k \\
&\leq (\log n)\frac{1}{2}(n-1)n - \frac{1}{2}(\frac{n}{2} - 1)\frac{n}{2} \\
&= \frac{1}{2}n^2 \log n - \frac{n}{2} \log n - \frac{1}{8}n^2 + \frac{n}{4} \\
&\leq \frac{1}{2}n^2 \log n - \frac{1}{8}n^2.
\end{aligned}$$

The last step holds since

$$\frac{n}{4} \leq \frac{n}{2} \log n,$$

when $n \geq 3$. □

Now we are ready for the final analysis.

Theorem 8.94. *Let $T(n)$ be the average case runtime of* `Quicksort` *on an array of size n. Then*

$$T(n) = \Theta(n \log n).$$

Proof: *We need to show that $T(n) = O(n \log n)$ and $T(n) = \Omega(n \log n)$. To prove that $T(n) = O(n \log n)$, we will show that for some constant a,*

$$T(n) \le an \log n \quad \text{for all } n \ge 2.$$ [a]

When $n = 2$,

$$an \log n = a2 \log 2 = 2a,$$

and a can be chosen large enough so that $T(2) \le 2a$. Thus, the inequality holds for the base case. Let $T(1) = C$, for some constant C. For $2 < k < n$, assume $T(k) \le ak \log k$. Then

$$\begin{aligned}
T(n) &= \frac{2}{n}\sum_{k=1}^{n-1} T(k) + \Theta(n) \\
&\le \frac{2}{n}\sum_{k=2}^{n-1} ak \log k + \frac{2}{n}T(1) + \Theta(n) && \text{(by assumption)} \\
&= \frac{2a}{n}\sum_{k=2}^{n-1} k \log k + \frac{2}{n}C + \Theta(n) \\
&\le \frac{2a}{n}\sum_{k=2}^{n-1} k \log k + C + \Theta(n) && (\text{since } \tfrac{2}{n} \le 1) \\
&\le \frac{2a}{n}\left(\frac{1}{2}n^2 \log n - \frac{1}{8}n^2\right) + C + \Theta(n) && \text{(by Lemma 2)} \\
&= an \log n - \frac{a}{4}n + C + \Theta(n) \\
&= an \log n + \left(\Theta(n) + C - \frac{a}{4}n\right) \\
&\le an \log n && (\text{choose } a \text{ so } \Theta(n) + C \le \tfrac{a}{4}n)
\end{aligned}$$

We have shown that with an appropriate choice of a, $T(n) \le an \log n$ for all $n \ge 2$, so $T(n) = O(n \log n)$.
We leave it to the reader to show that $T(n) = \Omega(n \log n)$. □

[a] We pick 2 for the base case since $n \log n = 0$ if $n = 1$, so we cannot make the inequality hold. Another solution would be to show that $T(n) \le an \log n + b$. In this case, b can be chosen so that the inequality holds for $n = 1$.

8.5 Problems

Problem 8.1. Use induction to prove that $\displaystyle\sum_{k=1}^{n} k^3 = \frac{n^2(n+1)^2}{4}$ for all $n \geq 1$.

Problem 8.2. Use induction to prove that for all $n \geq 2$,

$$\sum_{k=2}^{n} \frac{1}{(k-1)k} = \frac{1}{1\cdot 2} + \frac{1}{2\cdot 3} + \frac{1}{3\cdot 4} + \cdots + \frac{1}{(n-1)\cdot n} = \frac{n-1}{n}.$$

Problem 8.3. Prove that for all positive integers n, $f_1^2 + f_2^2 + \cdots + f_n^2 = f_n f_{n+1}$, where f_n is the nth Fibonacci number.

Problem 8.4. Prove that the number of binary palindromes of length $2k+1$ (odd length) is 2^{k+1} for all $k \geq 0$.

Problem 8.5. Explain why the following joke never ends: *Pete and Repete got in a boat. Pete fell off. Who's left?.*

Problem 8.6. Prove that the `FibR(n)` algorithm from Example 8.39 correctly computes f_n. (Hint: Use induction. How many base cases do you need? Do you need weak or strong induction?)

Problem 8.7. In Example 8.80 we gave a solution to the recurrence $f_n = f_{n-1} + f_{n-2}$, $f_0 = 0$, $f_1 = 1$. Use the substitution method to re-prove this. (Hint: Recall that the roots to the polynomial $x^2 - x - 1 = 0$ are $\frac{1\pm\sqrt{5}}{2}$. This is equivalent to $x^2 = x + 1$. You will find this helpful in the inductive step of the proof.

Problem 8.8. Find and prove a solution for each of the following recurrence relations using two different techniques (this will not only help you verify that your solutions are correct, but it will also give you more practice using each of the techniques). *At least one of the techniques must yield an exact formula if possible.*

(a) $T(n) = T(n/2) + n^2$, $T(1) = 1$.

(b) $T(n) = T(n/2) + n$, $T(1) = 1$.

(c) $T(n) = 2T(n/2) + n^2$, $T(1) = 1$.

(d) $T(n) = T(n-1) \cdot T(n-2)$, $T(0) = 1$, $T(1) = 2$.

(e) $T(n) = T(n-1) + n^2$, $T(1) = 1$.

(f) $T(n) = T(n-1) + 2n$, $T(1) = 2$.

Problem 8.9. Give an exact solution for each of the following recurrence relations.

(a) $a_n = 3a_{n-1}$, $a_1 = 5$.

(b) $a_n = 3a_{n-1} + 2n$, $a_1 = 5$.

(c) $a_n = a_{n-1} + 2a_{n-2}$, $a_0 = 2$, $a_1 = 5$.

(d) $a_n = 6a_{n-1} + 9a_{n-2}$, $a_0 = 1$, $a_1 = 2$.

(e) $a_n = -a_{n-1} + 6a_{n-2}$, $a_0 = 4$, $a_1 = 5$.

Problem 8.10. Use the Master Theorem to find a tight bound for each of the following recurrence relations.

(a) $T(n) = 8T(n/2) + 7n^3 + 6n^2 + 5n + 4$.

(b) $T(n) = 3T(n/5) + n^2 - 4n + 23$.

(c) $T(n) = 3T(n/2) + 3$.

(d) $T(n) = T(n/3) + n$.

(e) $T(n) = 2T(2n/5) + n$.

(f) $T(n) = 5T(2n/5) + n$.

Problem 8.11. Prove that the `RPartition` algorithm from Example 8.91 has complexity $\Theta(n)$.

Problem 8.12. Consider the classic *bubble sort* algorithm (see Example 7.132).

(a) Write a recursive version of the bubble sort algorithm. (Hint: The algorithm I have in mind should contain one recursive call and one loop.)

(b) Let $B(n)$ be the complexity of your recursive version of bubble sort. Give a recurrence relation for $B(n)$.

(c) Solve the recurrence relation for $B(n)$ that you developed in part (b).

(d) Is your recursive implementation better, worse, or the same as the iterative one given in Example 7.132? Justify your answer.

Problem 8.13. Consider the following algorithm (remember that integer division truncates):

```
int halfIt(int n) {
    if(n>0) {
        return 1 + halfIt(n/2);
    } else {
        return 0;
    }
}
```

(a) What does `halfIt(n)` return? Your answer should be a function of n.

(b) Prove that the algorithm is correct. That is, prove that it returns the answer you gave in part (a).

(c) What is the complexity of `halfIt(n)`? Give and prove an exact formula. (Hint: This will probably involve developing and solving a recurrence relation.)

Problem 8.14. This problem involves an algorithm to compute the sum of the first n squares $\left(\text{i.e. } \sum_{k=1}^{n} k^2\right)$ using recursion.

(a) Write an algorithm to compute $\sum_{k=1}^{n} k^2$ that uses recursion and only uses the increment operator for arithmetic (e.g., you cannot use addition or multiplication). (Hint: The algorithm I have in mind has one recursive call and one or two loops.)

(b) Let $S(n)$ be the complexity of your algorithm from part (a). Give a recurrence relation for $S(n)$.

(c) Solve the recurrence relation for $S(n)$ that you developed in part (b).

(d) Give a recursive linear-time algorithm to solve this same problem (with no restrictions on what operations you may use). Prove that the algorithm is linear.

(e) Give a constant-time algorithm to solve this same problem (with no restrictions on what you may use). Prove that the algorithm is constant.

(f) Discuss the relative merits of the three algorithms. Which algorithm is best? Worst? Justify.

Problem 8.15. Assuming the priests can move one disk per second, that they started moving disks 6000 years ago, and that the legend of the Towers of Hanoi is true, when will the world end?

Problem 8.16. Prove that the `stoogeSort` algorithm given in Exercise 8.87 correctly sorts an array of n integers.

Chapter 9

Counting

In this chapter we provide a very brief introduction to a field called *combinatorics*. We are actually only going to scratch the surface of this very broad and deep subfield of mathematics and theoretical computer science. We will focus on a subfield of combinatorics that is sometimes called *enumeration*. That is, we will mostly concern ourselves with how to count things.

It turns out that combinatorial problems are notoriously deceptive. Sometimes they can seem much harder than they are, and at other times they seem easier than they are. In fact, there are many cases in which one combinatorial problem will be relatively easy to solve, but a very closely related problem that seems almost identical will be very difficult to solve.

When solving combinatorial problems, you need to make sure you fully understand what is being asked and make sure you are taking everything into account appropriately. I used to tell students that combinatorics was easy. I don't say that anymore. In some sense it is easy. But it is also easy to make mistakes.

9.1 The Multiplication and Sum Rules

We begin our study of combinatorial methods with the following two fundamental principles. They are both pretty intuitive. The only difficulty is realizing which one applies to a given situation. If you have a good understanding of what you are counting, the choice is generally pretty clear.

Theorem 9.1 (Sum Rule). *Let $E_1, E_2, \ldots, E_k$, be pairwise finite disjoint sets. Then*

$$|E_1 \cup E_2 \cup \cdots \cup E_k| = |E_1| + |E_2| + \cdots + |E_k|.$$

Another way of putting the sum rule is this: If you have to accomplish some task and you can do it in one of n_1 ways, or one of n_2 ways, etc., up to one of n_k ways, and none of the ways of doing the task on any of the list are the same, then there are $n_1 + n_2 + \cdots + n_k$ ways of doing the task.

Example 9.2. I have 5 brown shirts, 4 green shirts, 10 red shirts, and 3 blue shirts. How many choices do I have if I intend to wear one shirt?

Solution: Since each list of shirts is independent of the others, I can use the sum rule. Therefore I can choose any of my $5 + 4 + 10 + 4 = 22$ shirts.

Example 9.3. How many ordered pairs of integers (x, y) are there such that $0 < |xy| \leq 5$?

Solution: Let $E_k = \{(x, y) \in \mathbb{Z}^2 : |xy| = k\}$ for $k = 1, \ldots, 5$. Then the desired number is

$$|E_1| + |E_2| + \cdots + |E_5|.$$

We can compute each of the these as follows:

$$\begin{array}{rcl} E_1 & = & \{(-1,-1),(-1,1),(1,-1),(1,1)\} \\ E_2 & = & \{(-2,-1),(-2,1),(-1,-2),(-1,2),(1,-2),(1,2),(2,-1),(2,1)\} \\ E_3 & = & \{(-3,-1),(-3,1),(-1,-3),(-1,3),(1,-3),(1,3),(3,-1),(3,1)\} \\ E_4 & = & \{(-4,-1),(-4,1),(-2,-2),(-2,2),(-1,-4),(-1,4),(1,-4), \\ & & (1,4),(2,-2),(2,2),(4,-1),(4,1)\} \\ E_5 & = & \{(-5,-1),(-5,1),(-1,-5),(-1,5),(1,-5),(1,5),(5,-1),(5,1)\} \end{array}$$

The desired number is therefore $4 + 8 + 8 + 12 + 8 = 40$.

⋆**Exercise 9.4.** For dessert you can have cake, ice cream or fruit. There are 3 kinds of cake, 8 kinds of ice cream and 5 different of fruits. How many choices do you have for dessert?

Answer ____________________

Theorem 9.5 (Product Rule). *Let $E_1, E_2, \ldots, E_k$, be finite sets. Then*

$$|E_1 \times E_2 \times \cdots \times E_k| = |E_1| \cdot |E_2| \cdots |E_k|.$$

Another way of putting the product rule is this: If you need to accomplish some task that takes k steps, and there are n_1 ways of accomplishing the first step, n_2 ways of accomplishing the second step, etc., and n_k ways of accomplishing the kth step, then there are $n_1 n_2 \cdots n_k$ ways of accomplishing the task.

Example 9.6. I have 5 pairs of socks, 10 pairs of shorts, and 8 t-shirts. How many choices do I have if I intend to wear one of each?

Solution: I can think of choosing what to wear as a task broken into 3 steps: I have to choose a pair of socks (5 ways), a pair of shorts (10 ways), and finally a t-shirt (8 ways). Thus I have $5 \times 10 \times 8 = 400$ choices.

⋆**Exercise 9.7.** If license plates are required to have 3 letters followed by 3 digits, how many license plates are possible?

Answer ____________________

Example 9.8. The positive divisors of 400 are written in increasing order

$$1, 2, 4, 5, 8, \dots, 200, 400.$$

How many integers are there in this sequence? How many of the divisors of 400 are perfect squares?

> **Solution:** Since $400 = 2^4 \cdot 5^2$, any positive divisor of 400 has the form $2^a 5^b$ where $0 \le a \le 4$ and $0 \le b \le 2$. Thus there are 5 choices for a and 3 choices for b for a total of $5 \cdot 3 = 15$ positive divisors.
>
> To be a perfect square, a positive divisor of 400 must be of the form $2^\alpha 5^\beta$ with $\alpha \in \{0, 2, 4\}$ and $\beta \in \{0, 2\}$. Thus there are $3 \cdot 2 = 6$ divisors of 400 which are also perfect squares.

It is easy to generalize Example 9.8 to obtain the following theorem.

Theorem 9.9. *Let the positive integer n have the prime factorization*

$$n = p_1^{a_1} p_2^{a_2} \cdots p_k^{a_k},$$

where the p_i are distinct primes, and the a_i are integers ≥ 1. If $d(n)$ denotes the number of positive divisors of n, then

$$d(n) = (a_1 + 1)(a_2 + 1) \cdots (a_k + 1).$$

⋆**Exercise 9.10.** Prove Theorem 9.9.

⋆**Question 9.11.** Whether or not you realize it, you used the fact that the p_i were distinct primes in your proof of Theorem 9.9 (assuming you did the proof correctly). Explain where that fact was used (perhaps implicitly).

Answer ____

____

Example 9.12. What is the value of *sum* after each of the following segments of code?

```
int sum=0;
for(int i=0;i<n;i++) {
    for(int i=0;i<m;i++) {
        sum = sum + 1;
    }
}
```

```
int sum=0;
for(int i=0;i<n;i++) {
    sum = sum + 1;
}
for(int i=0;i<m;i++) {
    sum = sum + 1;
}
```

Solution: In the code on the left, the inner loop executes m times, so every time the inner loop executes, *sum* gets m added to it. The outer loop executes n times, each time calling the inner loop. Therefore m is added to *sum* n times, so $sum = n \times m$ at the end.

In the code on the right, The first loop adds n to *sum*, and then the second loop adds m to *sum*. Therefore, $sum = n + m$ at the end.

The following problem can be solved using the product rule–you just need to figure out how.

⋆**Exercise 9.13.** The number 3 can be expressed as a sum of one or more positive integers in four ways, namely, as 3, $1+2$, $2+1$, and $1+1+1$. Show that any positive integer n can be so expressed in 2^{n-1} ways.

Answer ____

____

____

Example 9.14. Each day I need to decide between wearing a t-shirt or a polo shirt. I have 50 t-shirts and 5 polo shirts. I also have to decide whether to where jeans, shorts, or slacks. I have 5 pairs of jeans, 15 pairs of shorts, and 4 pairs of slacks. How many different choices do I have when I am getting dressed?

Solution: I have $50+5=55$ choices for a shirt and $5+15+4=24$ choices or pants. So the total number of choices if $55 \cdot 24 = 1320$.

⋆**Exercise 9.15.** If license plates are required to have 5 characters, each of which is either a digits or a letter, how many license plates are possible?

Answer ______________________________

____

⋆**Exercise 9.16.** How many bit strings are there of length n?

Answer ______________________________

Example 9.17. The integers from 1 to 1000 are written in succession. Find the sum of all the digits.

Solution: When writing the integers from 000 to 999 (with three digits), $3 \times 1000 = 3000$ digits are used. Each of the 10 digits is used an equal number of times, so each digit is used 300 times. The the sum of the digits in the interval 000 to 999 is thus

$$(0+1+2+3+4+5+6+7+8+9) \cdot 300 = 13500.$$

Therefore, the sum of the digits when writing the integers from 1 to 1000 is $13500 + 1 = 13501$.

⋆**Fill in the details 9.18.** In C++, identifiers (e.g. variable and function names) can contain only letters (upper and/or lower case), digits, and the underscore character. They may not begin with a digit.[a]

(a) There are $26 + 26 + 1 = 53$ possible identifiers that contain a single character.

(b) There are $53 \cdot (26 + 26 + 10 + 1) = 53 \cdot 63 = 3339$ possible identifiers with two characters.

(c) There are ____________ possible identifiers that contain a three characters.

(d) There are ____________ possible identifiers that contain a four characters.

(e) There are ____________ possible identifiers that contain a k characters.

[a] There are 84 reserved keywords that cannot be used, but we will ignore these for this exercise.

9.2 Pigeonhole Principle

The following theorem seems so obvious that it doesn't need to be stated. However, it often comes in handy in unexpected situations.

Theorem 9.19 (The Pigeonhole Principle)**.** *If n is a positive integer and $n+1$ or more objects are placed into n boxes, then one of the boxes contains at least two objects.*

Notice that the pigeonhole principle is saying that this is true *no matter how the objects are places in the boxes.* In other words, don't assume that $n-1$ boxes have one object and 1 box has 2 objects. It is possible that all $n+1$ objects are in the same box. But no matter how the objects are distributed in the boxes, we can be sure that there is some box with at least two objects.

Example 9.20. In any group of 13 people, there are always two who have their birthday on the same month. Similarly, if there are 32 people, at least two people were born on the same day of the month.

⋆**Exercise 9.21.** What can you say about the digits in a number that is 11 digits long?

Answer ____

The pigeonhole principle can be generalized.

Theorem 9.22 (The Generalized Pigeonhole Principle)**.** *If n objects are placed into k boxes, then there is at least one box that contains at least $\lceil n/k \rceil$ objects.*

> **Proof:** *Assume not. Then each of the k boxes contains no more than $\lceil n/k \rceil - 1$ objects. Notice that $\lceil n/k \rceil < n/k + 1$ (convince yourself that this is always true). Thus, the total number of objects in the k boxes is at most*
>
> $$k\left(\lceil n/k \rceil - 1\right) < k(n/k + 1 - 1) = n,$$
>
> *contradicting the fact that there are n objects in the boxes. Therefore, some box contains at least $\lceil n/k \rceil$ objects.* □

The tricky part about using the pigeonhole principle is identifying the *drawers* and *objects*. Once that is done, applying either form of the pigeonhole principle is straightforward. Actually, often the trickiest thing is identifying that the pigeonhole principle even applies to the problem you are trying to solve.

Example 9.23. A drawer contains an infinite supply of white, black, and blue socks. What is the smallest number of socks you must take from the drawer in order to be guaranteed that you have a matching pair?

> **Solution:** Clearly I could grab one of each color, so three is not enough. But according the the Pigeonhole Principle, if I take 4 socks, then I will get at least $\lceil 4/3 \rceil = 2$ of the same color (the colors correspond to the boxes). So 4 socks will guarantee a matched pair.
>
> Notice that I showed two things in this proof. I showed that 4 socks was enough,

but I also showed that 3 was not enough. This is important. For instance, 5 is enough, but it isn't the smallest number that works.

⋆**Exercise 9.24.** An urn contains 28 blue marbles, 20 red marbles, 12 white marbles, 10 yellow marbles, and 8 magenta marbles. How many marbles must be drawn from the urn in order to assure that there will be 15 marbles of the same color? Justify your answer.

Answer ______________________________

____

____

⋆**Exercise 9.25.** You are in line to get tickets to a concert. Each person can get at most 4 tickets. There are only 100 tickets available. The girl behind you in line says "I sure hope there are enough tickets for me. You're lucky, though. You will get as many as you want." What does she know, and under what circumstances will she get any tickets?

Answer ______________________________

____

____

The pigeonhole principle is useful in *existence* proofs–that is, proofs that show that something exists without actually identifying it concretely.

Example 9.26. Show that amongst any seven distinct positive integers not exceeding 126, one can find two of them, say a and b, which satisfy

$$b < a \leq 2b.$$

Solution: Split the numbers $\{1, 2, 3, \ldots, 126\}$ into the six sets

$$\{1, 2\}, \{3, 4, 5, 6\}, \{7, 8, \ldots, 13, 14\}, \{15, 16, \ldots, 29, 30\},$$

$$\{31, 32, \ldots, 61, 62\} \text{ and } \{63, 64, \ldots, 126\}.$$

By the Pigeonhole Principle, two of the seven numbers must lie in one of the six sets, and obviously, any such two will satisfy the stated inequality.

Example 9.27. Given any 9 integers whose prime factors lie in the set $\{3, 7, 11\}$ prove that there must be two whose product is a square.

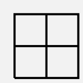
Solution: For an integer to be a square, all the exponents of its prime factorisation must be even. Any integer in the given set has a prime factorisation of the form $3^a 7^b 11^c$. Now each triplet (a, b, c) has one of the following 8 parity patterns: (even, even, even), (even, even, odd), (even, odd, even), (even, odd, odd), (odd, even, even), (odd, even, odd), (odd, odd, even), (odd, odd, odd). In a group of 9 such integers, there must be two with the same parity patterns in the exponents. Take these two. Their product is a square, since the sum of each corresponding exponent will be even.

⋆**Exercise 9.28.** The nine entries of a 3×3 grid are filled with -1, 0, or 1. Prove that among the eight resulting sums (three columns, three rows, or two diagonals) there will always be two that add to the same number.

Answer

Example 9.29. Prove that if five points are taken on or inside a unit square, there must always be two whose distance is no more than $\dfrac{\sqrt{2}}{2}$.

Solution: Split the square into four congruent squares as shown to the right. At least two of the points must fall into one of the smaller squares. The longest distance between two points in one of the smaller squares is, by the Pythagorean Theorem, $\sqrt{(\frac{1}{2})^2 + (\frac{1}{2})^2} = \dfrac{\sqrt{2}}{2}$. Thus, the result holds.



Example 9.30. Given any set of ten natural numbers between 1 and 99 inclusive, prove that there are two distinct nonempty subsets of the set with equal sums of their elements. (Hint: How many possible subsets are there, and what are the possible sums of the elements within the subsets?)

Solution: There are $2^{10} - 1 = 1023$ non-empty subsets that one can form with a given 10-element set. To each of these subsets we associate the sum of its elements. The minimum value that the sum can be for any subset is $1 + 2 + \cdots + 10 = 55$, and the maximum value is $90 + 91 + \cdots + 99 = 945$. Since the number of possible sums is no more than $945 - 55 + 1 = 891 < 1023$, there must be at least two different subsets that have the same sum.

Example 9.31. Prove that if 55 of the integers from 1 to 100 are selected, then two of them differ by 10.

Solution: First observe that if we choose $n+1$ integers from any set of $2n$ consecutive integers, there will always be some two that differ by n. This is because we can pair the $2n$ consecutive integers

$$\{a+1, a+2, a+3, \ldots, a+2n\}$$

into the n pairs

$$\{a+1, a+n+1\}, \{a+2, a+n+2\}, \ldots, \{a+n, a+2n\},$$

and if $n+1$ integers are chosen from this, there must be two that belong to the same group.

So now group the one hundred integers as follows:

$$\{1, 2, \ldots 20\}, \{21, 22, \ldots, 40\},$$

$$\{41, 42, \ldots, 60\}, \ \{61, 62, \ldots, 80\}$$

and

$$\{81, 82, \ldots, 100\}.$$

If we select fifty five integers, then we must have selected at least $\lceil 55/5 \rceil = 11$ from one of the groups. From that group, by the above observation (let $n = 10$), there must be two that differ by 10.

⋆**Exercise 9.32.** An eccentric old man has five cats. These cats have 16 kittens among themselves. What is the largest integer n for which one can say that at least one of the five cats has n kittens?

Answer ____

⋆**Evaluate 9.33.** Prove that at a party with at least two people, there are two people who have shaken hands with the same number of people.

Proof 1: There are $n-1$ people 1 person can shake hands with—4 others if there are 5 people at the party. At one given time two people cannot shake hands with 0 people and $n-1$ people simultaneously because there are 4 slots to fill and 5 people therefore by the pigeonhole principle at least two people shake hands with the same number of others.

Evaluation ____

Proof 2: Assume that at a gathering of $n \geq 2$ people, there are no two people who have shaken hands with the same number of people. If there are two people at the gathering they must either shake hands with each other or shake hands with nobody. However, this contradicts the assumption that no two people have shaken hands with the same number of people. Therefore, by contradiction, at a gathering of $n \geq 2$ people, there are at least two people who have shaken hands with the same number of people.

Evaluation ______________________________

____

Proof 3: Assume that at a gathering of $n \geq 2$ people, there are no two people who have shaken hands with the same number of people. Person n shakes hands with $n-1$ people because you can't shake your own hand. Person $n-1$ then shakes hands with $n-2$ people and so on...until you reach the last person. He shakes hands with no one which fulfills the contradiction.

Evaluation ______________________________

____

⋆**Exercise 9.34.** Give a correct proof of the problem stated in Evaluate 9.33.

⋆**Exercise 9.35.** There are seventeen friends from high school that all keep in touch by writing letters to each other.[a] To be clear, each person writes separate letters to each of the others. In their letters only three different topics are discussed. Each pair only corresponds about one of these topics. Prove that there at least three people who all write to each other about the same topic.

[a]You do know what letters are, right? They are like e-mail, only they are written on paper, are sent to just one person, and are delivered to your physical mail box.

9.3 Permutations and Combinations

Most of the counting problems we will be dealing with can be classified into one of four categories. The categories are determined by two factors: whether or not repetition is allowed and whether or not order matters. After presenting a brief example of each of these categories, we will go into more detail about each in the following four subsections.

Example 9.36. Consider the set $\{a, b, c, d\}$. Suppose we "select" two letters from these four. Depending on our interpretation, we may obtain the following answers.

(a) **Permutations with repetitions.** The *order* of listing the letters is important, and *repetition is* allowed. In this case there are $4 \cdot 4 = 16$ possible selections:

aa	ab	ac	ad
ba	bb	bc	bd
ca	cb	cc	cd
da	db	dc	dd

(b) **Permutations without repetitions.** The *order* of listing the letters is important, and *repetition is not* allowed. In this case there are $4 \cdot 3 = 12$ possible selections:

	ab	ac	ad
ba		bc	bd
ca	cb		cd
da	db	dc	

(c) **Combinations with repetitions.** The *order* of listing the letters is **not** important, and *repetition is* allowed. In this case there are $\frac{4 \cdot 3}{2} + 4 = 10$ possible selections:

aa	ab	ac	ad
	bb	bc	bd
		cc	cd
			dd

(d) **Combinations without repetitions.** The *order* of listing the letters is **not** important, and *repetition is not* allowed. In this case there are $\frac{4 \cdot 3}{2} = 6$ possible selections:

	ab	ac	ad
		bc	bd
			cd

Although most of the simple types of counting problems we want to solve can be reduced to one of these four, care must be taken. The previous example assumed that we had a set of *distinguishable* objects. When objects are not distinguishable, the situation is a more complicated.

9.3.1 Permutations without Repetitions

Definition 9.37. *Let $x_1, x_2, \ldots, x_n$ be n distinct objects. A* **permutation** *of these objects is simply a rearrangement of them.*

Example 9.38. There are 24 permutations of the letters in $MATH$, namely

$$\begin{array}{cccccc} MATH & MAHT & MTAH & MTHA & MHTA & MHAT \\ AMTH & AMHT & ATMH & ATHM & AHTM & AHMT \\ TAMH & TAHM & TMAH & TMHA & THMA & THAM \\ HATM & HAMT & HTAM & HTMA & HMTA & HMAT \end{array}$$

$\star$**Exercise 9.39.** List all of the permutations of EAT

Answer ____

____

Theorem 9.40. *Let $x_1, x_2, \ldots, x_n$ be n distinct objects. Then there are $n!$ permutations of them.*

Proof: *The first position can be chosen in n ways, the second object in $n-1$ ways, the third in $n-2$, etc. This gives*

$$n(n-1)(n-2)\cdots 2 \cdot 1 = n!.$$

$\square$

Example 9.41. Previously we saw that there are $24 = 4!$ permutations of the letters in $MATH$ and $6 = 3!$ permutations of the letters in EAT.

$\star$**Exercise 9.42.** How many permutations are there of the letters in UNCOPYRIGHTABLE?

Answer ____

Let's see some slightly more complicated examples.

Example 9.43. A bookshelf contains 5 German books, 7 Spanish books and 8 French books. Each book is different from one another. How many different arrangements can be done of these books if

(a) we put no restrictions on how they can be arranged?

(b) books of each language must be next to each other?

(c) all the French books must be next to each other?

(d) no two French books must be next to each other?

Solution:

(a) We are permuting $5+7+8=20$ objects. Thus the number of arrangements sought is $20! = 2432902008176640000$.

(b) "Glue" the books by language, this will assure that books of the same language are together. We permute the 3 languages in $3!$ ways. We permute the German books in $5!$ ways, the Spanish books in $7!$ ways and the French books in $8!$ ways. Hence the total number of ways is $3! \cdot 5! \cdot 7! \cdot 8! = 146313216000$.

(c) Align the German books and the Spanish books first. Putting these $5+7=12$ books creates $12+1=13$ spaces (we count the space before the first book, the spaces between books and the space after the last book). To assure that all the French books are next each other, we "glue" them together and put them in one of these spaces. Now, the French books can be permuted in $8!$ ways and the non-French books can be permuted in $12!$ ways. Thus the total number of permutations is

$$13 \cdot 8! \cdot 12! = 251073478656000.$$

(d) As with (c), we align the 12 German and Spanish books first, creating 13 spaces. To assure that no two French books are next to each other, we put them into these spaces. The first French book can be put into any of 13 spaces, the second into any of 12 remaining spaces, etc., and the eighth French book can be put into any 6 remaining spaces. Now, the non-French books can be permuted in $12!$ ways. Thus the total number of permutations is

$$13 \cdot 12 \cdot 11 \cdot 10 \cdot 9 \cdot 8 \cdot 7 \cdot 6 \cdot 12! = 24856274386944000.$$

⋆**Exercise 9.44.** Telephone numbers in *Land of the Flying Camels* have 7 digits, and the only digits available are $\{0, 1, 2, 3, 4, 5, 7, 8\}$. No telephone number may begin in 0, 1 or 5. Find the number of telephone numbers possible that meet the following criteria:

(a) You may not repeat any of the digits.

Answer ______________________________

(b) You may not repeat the digits and the phone numbers must be odd.

Answer ______________________________

The previous example and exercise should demonstrate that counting often requires thinking about things in different ways depending on the exact situation. This can be tricky, and it is very easy to make mistakes that lead to under or over counting possibilities. As you are solving problems, think very carefully about what you are counting so you don't fall into this trap.

9.3.2 Permutations with Repetitions

We now consider permutations with repeated objects.

Example 9.45. In how many ways may the letters of the word

$$MASSACHUSETTS$$

be permuted to form different strings?

Solution: We put subscripts on the repeats forming

$$MA_1S_1S_2A_2CHUS_3ET_1T_2S_4.$$

There are now 13 distinguishable objects, which can be permuted in 13! different ways by Theorem 9.40. But this counts some arrangements multiple times since in reality the duplicated letters are not distinguishable. Consider a single permutation of all of the distinguishable letters. If I permute the letters A_1A_2, I get the same permutation when ignoring the subscripts. The same thing is true of T_1T_2. Similarly, there are 4! permutations of $S_1S_2S_3S_4$, so there are 4! permutations that look the same (without the subscripts). Since I can do all of these independently, there are 2!2!4! permutations that look identical when the subscripts are removed. This is true of every permutation. Therefore, the actual number of permutations is $\dfrac{13!}{2!\cdot 4!\cdot 2!} = 64864800$.

The following exercises should help the technique used in the previous example to sink in.

⋆**Exercise 9.46.** Use an argument similar to that in Example 9.45 to determine the number of permutations in the letters in $TALL$.

Answer ____________________

⋆**Exercise 9.47.** List all of the permutations of the letters $TALL$.

Answer ____________________

⋆**Exercise 9.48.** How many permutations are there in the letters of $AEEEI$?

Answer ____________________

$\star$**Exercise 9.49.** List all of the permutations of the letters $AEEEI$.

Answer

The arguments from the previous examples and exercises can be generalized to prove the following.

Theorem 9.50. *Let there be k types of objects: n_1 of type 1; n_2 of type 2; etc. Then the number of ways in which these $n_1 + n_2 + \cdots + n_k$ objects can be rearranged is*

$$\frac{(n_1 + n_2 + \cdots + n_k)!}{n_1! \cdot n_2! \cdots n_k!}.$$

Example 9.51. How many permutations of the letters from $MASSACHUSETTS$ contain $MASS$?

> **Solution:** We can consider $MASS$ as one block along with the remaining 9 letters A, C, H, U, S, E, T, T, S. Thus, we are permuting 10 'letters'. There are two S's[a] and two T's and so the total number of permutations sought is
>
> $$\frac{10!}{2! \cdot 2!} = 907200.$$

[a] Remember, the other two S's are part of $MASS$, which we are now treating as a single object.

$\star$**Exercise 9.52.** How many permutations of the letters from the word $ALGORITHMS$ contain $SMITH$?

Answer

Example 9.53. In how many ways may we write the number 9 as the sum of three positive integer summands? Here order counts, so, for example, $1 + 7 + 1$ is to be regarded different from $7 + 1 + 1$.

> **Solution:** We need to find the values of a, b, and c such that $a + b + c = 9$, where $a, b, c \in \mathbb{Z}^+$. We will consider triples (a, b, c) listed smallest to largest and

determine how many ways each triple can be reordered. The possibilities are:

(a,b,c)	Number of permutations
$(1,1,7)$	$3!/2! = 3$
$(1,2,6)$	$3! = 6$
$(1,3,5)$	$3! = 6$
$(1,4,4)$	$3!/2! = 3$
$(2,2,5)$	$3!/2! = 3$
$(2,3,4)$	$3! = 6$
$(3,3,3)$	$3!/3! = 1$

Thus the number desired is $3+6+6+3+3+6+1 = 28$.

Example 9.54. In how many ways can the letters of the word **MURMUR** be arranged without allowing two of the same letters next to each other?

Solution: If we started with, say , **MU** then the **R** could be arranged in one of the following three ways:

M	**U**	**R**		**R**	

M	**U**	**R**			**R**

M	**U**		**R**		**R**

In the first case there are $2! = 2$ ways of putting the remaining **M** and **U**, in the second there are $2! = 2$ ways and in the third there is only $1!$ way. Thus starting the word with **MU** gives $2+2+1 = 5$ possible arrangements. In the general case, we can choose the first letter of the word in 3 ways, and the second in 2 ways. Thus the number of ways sought is $3 \cdot 2 \cdot 5 = 30$.[a]

[a]It should be noted that this analysis worked because the three letters each occurred twice. If this was not the case we would have had to work harder to solve the problem.

⋆**Exercise 9.55.** Telephone numbers in *Land of the Flying Camels* have 7 digits, and the only digits available are $\{0, 1, 2, 3, 4, 5, 7, 8\}$. No telephone number may begin with 0, 1 or 5. Find the number of telephone numbers possible that meet the following criteria:

(a) You may repeat all digits.

Answer ________________________________

(b) You may repeat digits, but the last digit must be even.

Answer ________________________________

(c) You may repeat digits, but the last digit must be odd.

Answer ________________________________

Example 9.56. In how many ways can the letters of the word **AFFECTION** be arranged, keeping the vowels in their natural order and not letting the two **F**'s come together?

Solution: There are $\dfrac{9!}{2!}$ ways of permuting the letters of **AFFECTION**. The 4 vowels can be permuted in 4! ways, and in only one of these will they be in their natural order. Thus there are $\dfrac{9!}{2! \cdot 4!}$ ways of permuting the letters of **AFFECTION** in which their vowels keep their natural order. If we treat FF as a single letter, there are 8! ways of permuting the letters so that the F's stay together. Hence there are $\dfrac{8!}{4!}$ permutations of **AFFECTION** where the vowels occur in their natural order and the FF's are together. In conclusion, the number of permutations sought is

$$\frac{9!}{2! \cdot 4!} - \frac{8!}{4!} = \frac{8!}{4!}\left(\frac{9}{2} - 1\right) = 8 \cdot 7 \cdot 6 \cdot 5 \cdot \frac{7}{2} = 5880.$$

9.3.3 Combinations without Repetitions

Let's begin with some important notation.

Definition 9.57. *Let n, k be non-negative integers with $0 \leq k \leq n$. The* **binomial coefficient** $\binom{n}{k}$ *(read "n choose k") is defined by*

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n \cdot (n-1) \cdot (n-2) \cdots (n-k+1)}{1 \cdot 2 \cdot 3 \cdots k}.$$

An alternative notation is $C(n, k)$. This notation is particularly useful when you want to express a binomial coefficient in the middle of text since it doesn't take up two lines.

Note: *Observe that in the last fraction, there are k factors in both the numerator and denominator. Also, observe the boundary conditions*

$$\binom{n}{0} = \binom{n}{n} = 1, \qquad \binom{n}{1} = \binom{n}{n-1} = n.$$

Example 9.58. We have

$$\begin{aligned} \binom{6}{3} &= \frac{6 \cdot 5 \cdot 4}{1 \cdot 2 \cdot 3} = 20, \\ \binom{11}{2} &= \frac{11 \cdot 10}{1 \cdot 2} = 55, \\ \binom{12}{7} &= \frac{12 \cdot 11 \cdot 10 \cdot 9 \cdot 8 \cdot 7 \cdot 6}{1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 7} = 792, \\ \binom{110}{0} &= 1. \end{aligned}$$

⋆**Exercise 9.59.** Compute each of the following

(a) $\binom{7}{5} =$ ________________________

(b) $\binom{12}{2} =$ ________________________

(c) $\binom{10}{5} =$ ________________________

(d) $\binom{200}{4} =$ ________________________

(e) $\binom{67}{0} =$ ____________________

If there are n kittens and you decide to take k of them home, you also decided *not* to take $n-k$ of them home. This idea leads to the following important theorem.

Theorem 9.60. *If $n, k \in \mathbb{Z}$, with $0 \leq k \leq n$, then*

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n!}{(n-k)!(n-(n-k))!} = \binom{n}{n-k}$$

Proof: *Since $k = n-(n-k)$, the result is obvious.* □

Example 9.61.

$$\binom{11}{9} = \binom{11}{2} = 55.$$

$$\binom{12}{5} = \binom{12}{7} = 792.$$

$$\binom{110}{109} = \binom{110}{1} = 110$$

⋆**Exercise 9.62.** Compute each of the following

(a) $\binom{17}{15} =$ ____________________

(b) $\binom{12}{10} =$ ____________________

(c) $\binom{10}{6} =$ ____________________

(d) $\binom{200}{196} =$ ____________________

(e) $\binom{67}{66} =$ ____________________

Definition 9.63. *Let there be n distinguishable objects. A k-**combination** is a selection of k, ($0 \leq k \leq n$) objects from the n made without regards to order.*

Example 9.64. The 2-combinations from the list $\{X, Y, Z, W\}$ are

$$XY, XZ, XW, YZ, YW, WZ.$$

Notice that YX (for instance) is not on the list because XY is already on the list and order does not matter.

Example 9.65. The 3-combinations from the list $\{X, Y, Z, W\}$ are

$$XYZ, XYW, XZW, YWZ.$$

⋆**Exercise 9.66.** List the 2-combinations from the list $\{1, 2, 3, 4, 5\}$

Answer ____

____

Theorem 9.67. *Let there be n distinguishable objects, and let k, $0 \leq k \leq n$. Then the numbers of k-combinations of these n objects is $\binom{n}{k}$.*

> **Proof:** *The number of ways of picking k objects if the order matters is $n(n-1)(n-2)\cdots(n-k+1)$ since there are n ways of choosing the first object, $n-1$ ways of choosing the second object, etc.. Since each k-combination can be ordered in $k!$ ways, the number of ordered lists of size k is $k!$ times the number of k-combinations. Put another way, the number of k-combinations is the number above divided by $k!$. That is, the total number of k-combinations is*
>
> $$\frac{n(n-1)(n-2)\cdots(n-k+1)}{k!} = \binom{n}{k}.$$
>
> □

Example 9.68. From a group of 10 people, we may choose a committee of 4 in $\binom{10}{4} = 210$ ways.

⋆**Evaluate 9.69.** A family has seven women and nine men. They need five of them to get together to plan a party. If at least one of the five must be a woman, how many ways are there to select the five?

Solution 1: Since one has to be a woman, this is equivalent to selecting four people from a pool of 15, so the answer is $\binom{15}{4}$.

Evaluation ______________________________

____

Solution 2: There are 7 women to choose from to ensure there is one woman, and then 4 more need to be selected from the remaining 15. There are $\binom{15}{4}$ ways of doing that. Therefore the total number of ways is $\binom{15}{4}+7$.

Evaluation ______________________________

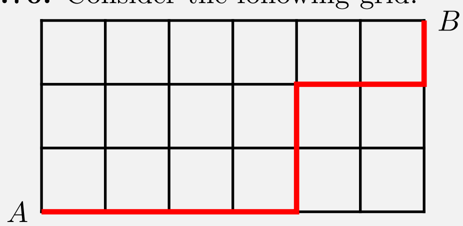
____

Solution 3: There are $\binom{16}{5}$ possible committees, $\binom{9}{5}$ of which contain only men. Thus, there are $\binom{16}{5}-\binom{9}{5}$ committees that contain at least one woman.

Evaluation ______________________________

____

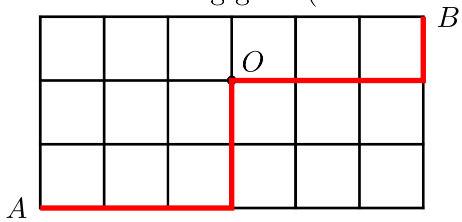
Example 9.70. Consider the following grid:



To count the number of shortest routes from A to B (one of which is given), observe that any shortest path must consist of 6 horizontal moves and 3 vertical ones for a total of $6 + 3 = 9$ moves. Once we choose which 6 of these 9 moves are horizontal the 3 vertical ones are determined. For instance, if I choose to go horizontal on moves 1, 2, 4, 6, 7, and 8, then moves 3, 5 and 9 must be vertical. Since there are 9 moves, I just need to choose which 6 of these are the horizontal moves. Thus there are $\binom{9}{6} = 84$ paths.

Another way to think about it is that we need to compute the number of permutations of $EEEEEENNN$, where E means move east, and N means move north. The number of permutations is $9!/(6! \cdot 3!) = \binom{9}{6}$.

⋆**Exercise 9.71.** Count the number of shortest routes from A to B that pass through point O in the following grid. (Hint: Break it into two subproblems and combine the solutions.)



⋆**Evaluate 9.72.** A family has seven women and nine men. How many ways are there to select five of them to plan a party if at least one man and one woman must be selected?

Solution 1: There are 7 choices for the first woman, 9 choices for the first man, and $\binom{14}{3}$ choices for the rest of the committee. Thus, there are $\binom{14}{3} \cdot 7 \cdot 9$ possible committees.

Evaluation ______________________________

____

Solution 2: Since one has to be a woman and one has to be a man, then they really just need to select 3 more member from the remaining 14 people, so the answer is $\binom{14}{3}$.

Evaluation ______________________________

____

Now it's your turn to give a correct solution to the previous problem.

⋆**Exercise 9.73.** A family has seven women and nine men. How many ways are there to select five of them to plan a party if at least one man and one woman must be selected?

⋆**Question 9.74.** In the answer to the previous problem, we pointed out that two sets of committees did not overlap. Why was that important?

Answer ______________________________

____

Example 9.75. Three different integers are drawn from the set $\{1, 2, \ldots, 20\}$. In how many ways may they be drawn so that their sum is divisible by 3?

Solution: In $\{1, 2, \ldots, 20\}$ there are

6	numbers leaving remainder 0
7	numbers leaving remainder 1
7	numbers leaving remainder 2

The sum of three numbers will be divisible by 3 when (a) the three numbers are divisible by 3; (b) one of the numbers is divisible by 3, one leaves remainder 1 and the third leaves remainder 2 upon division by 3; (c) all three leave remainder 1 upon division by 3; (d) all three leave remainder 2 upon division by 3. Hence the number of ways is

$$\binom{6}{3} + \binom{6}{1}\binom{7}{1}\binom{7}{1} + \binom{7}{3} + \binom{7}{3} = 384.$$

⋆**Evaluate 9.76.** The 300-level courses in the CS department are split into three groups: Foundations (361, 385), Applications (321, 342, 392), and Systems (335, 354, 376). In order to get a BS in computer science at Hope you need to take at least one course from each group. If you take four 300-level courses, how many different possibilities do you have that satisfy the requirements?

Solution 1: You have to take one from each group and then you can take any of the remaining 5 courses. So the total is $2 * 3 * 3 * 5 = 90$.

Evaluation ______________________________

____

Solution 2: $\binom{8}{4} = 70$

Evaluation ______________________________

____

⋆**Evaluate 9.77.** Using the same requirements from Evaluate 9.76, how many total ways are there to take 300-level courses that satisfy the requirements?

Solution 1: Take one from each group and then choose between 0 and 5 of the remaining 5. The total is therefore $2*3*3*\sum_{k=0}^{5}\binom{5}{k}$.

Evaluation ______________________________

____

Solution 2: Since you can take anywhere between 3 and 8 courses, the number of possibilities is $\binom{8}{3}+\binom{8}{4}+\binom{8}{5}+\binom{8}{6}+\binom{8}{7}+\binom{8}{8}$.

Evaluation ______________________________

____

9.3.4 Combinations with Repetitions

Example 9.78. How many ways are there to put 10 ping pong balls into 4 buckets?

Solution: We will solve this using a technique sometimes called *bars and stars*. We will represent the drawers with bars and the balls with stars. We will use 10 stars and 3 bars. To see why this is 3 and not 4, let's see how we represent the situation of having 3 balls in the first bucket, 5 in the second, none in the third, and 2 in the fourth:

|**||**

Do you see it? The bars act as separators between the buckets, which is why we have one less bar than the number of buckets.

Given this formulation, aren't we just trying to find all possible orderings of bars and stars? Indeed. To do so, all we need to do is determine where to put the stars, and the bars 'fall into place'. Alternatively, we can determine where to put the bars and let the stars fall into place. There are 13 spots and we need to choose 10 spots for the balls (the 'stars') or 3 spots for the bucket separators (the 'bars'). So the solution is

$$\binom{13}{10}=\binom{13}{3}=286.$$

Notice that Theorem 9.60 implies that these two methods of solving the problem will always be the same, which is a really good thing.

Example 9.79. How many ways are there to choose 10 pieces of fruit if you can take any number of bananas, oranges, apples, or pears and the order I select them does not matter?

Solution: Again we can use stars and bars so solve this problem. We need 10 stars to represent the chosen fruits and 3 bars to divide the four fruits we can choose from. The stars before the first bar represent bananas, those between the first and second bar are oranges, between the second and third are apples, and after the third are pears. Thus, we need to count the number of ways we can arrange 10 stars and 3 bars. Notice that this is exactly the same thing we did in the previous example, so the answer is

$$\binom{13}{10} = \binom{13}{3} = 286.$$

⋆**Exercise 9.80.** I want to make a sandwich that has 3 slices of meat. My refrigerator is well stocked because I have 11 different meats to choose from. How many choices do I have for my sandwich if I allow myself to have multiple slices of the same meat and the order the slices appear on the sandwich does not matter?

Answer ____

We can generalize the previous examples as follows.

Theorem 9.81. *There are* $\binom{n+k-1}{k} = \binom{n+k-1}{n-1}$ *ways of placing* k *indistinguishable objects into* n *distinguishable bins.*

This is also the number of ways of selecting k *objects from a collection of* n *objects if repetition is allowed.*

The previous theorem can be applied to various situations. As with the pigeonhole principle, the trickiest part is recognizing when and how to apply it.

Let's see another sort of counting problem.

Theorem 9.82 (De Moivre)**.** *Let* n *be a positive integer. The number of positive integer solutions to*

$$x_1 + x_2 + \cdots + x_r = n$$

is

$$\binom{n-1}{r-1}.$$

Proof: *Write* n *as*

$$n = 1 + 1 + \cdots + 1 + 1,$$

where there are n *1s and* $n-1$ $+$*s. To decompose* n *in* r *summands we only need to choose* $r-1$ *pluses from the* $n-1$*. For instance, writing* $n=7$ *as* $7 = 2+3+2$ *is equivalent to* $7 = (1+1)+(1+1+1)+(1+1)$*, where the* $+$*'s outside of the* *parentheses are the ones we chose (we apply the ones inside the parentheses to obtain the x_is). This proves the theorem.* □

Example 9.83. In how many ways may we write the number 9 as the sum of three positive integer summands? Here order counts, so, for example, $1+7+1$ is to be regarded different from $7+1+1$.

Solution: Notice that this is the same problem as Example 9.53. We are seeking integral solutions to

$$a+b+c=9, \quad a>0, b>0, c>0.$$

By Theorem 9.82 this is

$$\binom{9-1}{3-1} = \binom{8}{2} = 28.$$

Note: *The solution in Example 9.83 was much easier than the solution in Example 9.53, demonstrating the fact that choosing the right tool for the job can make a huge difference. Sometimes recognizing the best tool for the job can be tricky. Of course, the more problems of this type you solve, the easier it gets. Similarly, having more tools in your bag gives you more options.*

This also demonstrates something that is true of a lot of combinatorial problems: There are often several valid ways of approaching them. But there are also a lot of invalid approaches, so be careful!

⋆**Exercise 9.84.** In how many ways can 100 be written as the sum of four positive integer summands?

Answer ______________________________

____

The following corollary is similar to Theorem 9.82 except that the numbers are allowed to be 0.

Corollary 9.85. *Let n be a positive integer. The number of non-negative integer solutions to*

$$y_1+y_2+\cdots+y_k=n$$

is

$$\binom{n+k-1}{k-1}.$$

Proof: *Set $x_i-1=y_i$ for $i=1,\ldots,k$. Then $x_i\geq 1$, and equation*

$$y_1+y_2+\cdots+y_k=n$$

is equivalent to

$$x_1-1+x_2-1+\cdots+x_k-1=n$$

which is equivalent to

$$x_1 + x_2 + \cdots + x_k = n + k,$$

which has $\binom{n+k-1}{k-1}$ *solutions by Theorem 9.82.* $\square$

Notice that the previous corollary is very similar to Theorem 9.81. We leave it for the interested reader to determine whether or not there is a deeper connection between these two. The next example demonstrates that the technique used in the proof of Corollary 9.85 can be extended.

Example 9.86. Find the number of quadruples (a, b, c, d) of integers satisfying

$$a + b + c + d = 100, \;\; a \geq 30, b > 21, c \geq 1, d \geq 1.$$

Solution: Put $a' + 29 = a, b' + 20 = b$. Then we want the number of positive integer solutions to

$$a' + 29 + b' + 21 + c + d = 100,$$

or

$$a' + b' + c + d = 50.$$

By Theorem 9.82 this number is

$$\binom{50-1}{4-1} = \binom{49}{3} = 18424.$$

⋆**Exercise 9.87.** In how many ways may 1024 be written as the *product* of three positive integers? (Hint: Find the prime factorization of 1024 and then figure out why that helps.)

Answer ______________________________

____

____

9.4 Binomial Theorem

It is well known that

$$(a+b)^2 = a^2 + 2ab + b^2 \tag{9.1}$$

Multiplying this last equality by $a+b$ one obtains

$$(a+b)^3 = (a+b)^2(a+b) = a^3 + 3a^2b + 3ab^2 + b^3$$

Again, multiplying

$$(a+b)^3 = a^3 + 3a^2b + 3ab^2 + b^3 \tag{9.2}$$

by $a+b$ one obtains

$$(a+b)^4 = (a+b)^3(a+b) = a^4 + 4a^3b + 6a^2b^2 + 4ab^3 + b^4$$

This generalizes, as we see in the next theorem.

Theorem 9.88 (Binomial Theorem)**.** *Let x and y be variables and n be a nonnegative integer. Then*

$$(x+y)^n = \sum_{i=0}^{n} \binom{n}{i} x^{n-i} y^i.$$

Example 9.89. Expand $(4x+5)^3$, simplifying as much as possible.

Solution:

$$\begin{aligned}
(4x+5)^3 &= \binom{3}{0}(4x)^3 5^0 + \binom{3}{1}(4x)^2(5)^1 + \binom{3}{2}(4x)^1(5)^2 + \binom{3}{3}(4x)^0 5^3 \\
&= (4x)^3 + 3(4x)^2(5) + 3(4x)(5)^2 + 5^3 \\
&= 64x^3 + 240x^2 + 300x + 125
\end{aligned}$$

Example 9.90. In the following, $i = \sqrt{-1}$, so that $i^2 = -1$.

$$\begin{aligned}
(2+i)^5 &= 2^5 + 5(2)^4(i) + 10(2)^3(i)^2 + 10(2)^2(i)^3 + 5(2)(i)^4 + i^5 \\
&= 32 + 80i - 80 - 40i + 10 + i \\
&= -38 + 39i
\end{aligned}$$

Notice that we skipped the step of explicitly writing out the binomial coefficient for this example. You can do it either way–just make sure you aren't forgetting anything or making algebra mistakes by taking shortcuts.

⋆**Exercise 9.91.** Expand $(2x - y^2)^4$, simplifying as much as possible.

The most important things to remember when using the binomial theorem are not to forget the binomial coefficients, and not to forget that the powers (i.e. x^{n-i} and y^i) apply to the whole term, including any coefficients. A specific case that is easy to forget is a negative sign on the coefficient. Did you make any of these mistakes when doing the last exercise? Be sure to identify your errors so you can avoid them in the future.

⋆**Exercise 9.92.** Expand $(\sqrt{3} + \sqrt{5})^4$, simplifying as much as possible.

Example 9.93. Let $n \geq 1$. Find a closed form for $\displaystyle\sum_{k=0}^{n} \binom{n}{k} (-1)^k$.

Solution: Using a little algebra and the binomial theorem, we can see that

$$\sum_{k=0}^{n}\binom{n}{k}(-1)^k = \sum_{k=0}^{n}\binom{n}{k}1^{n-k}(-1)^k = (1-1)^n = 0.$$

⋆**Exercise 9.94.** Find a closed form for $\displaystyle\sum_{k=1}^{n}\binom{n}{k}3^k$.

If we ignore the variables in the Binomial Theorem and write down the coefficients for increasing values of n, a pattern, called *Pascal's Triangle*, emerges (see Figure 9.1).

$$\begin{array}{c}
1\\
1 \quad 1\\
1 \quad 2 \quad 1\\
1 \quad 3 \quad 3 \quad 1\\
1 \quad 4 \quad 6 \quad 4 \quad 1\\
1 \quad 5 \quad 10 \quad 10 \quad 5 \quad 1\\
1 \quad 6 \quad 15 \quad 20 \quad 15 \quad 6 \quad 1\\
1 \quad 7 \quad 21 \quad 35 \quad 35 \quad 21 \quad 7 \quad 1\\
1 \quad 8 \quad 28 \quad 56 \quad 70 \quad 56 \quad 28 \quad 8 \quad 1\\
1 \quad 9 \quad 36 \quad 84 \quad 126 \quad 126 \quad 84 \quad 36 \quad 9 \quad 1\\
1 \quad 10 \quad 45 \quad 120 \quad 210 \quad 252 \quad 210 \quad 120 \quad 45 \quad 10 \quad 1\\
\vdots
\end{array}$$

Figure 9.1: Pascal's Triangle

Notice that each entry (except for the 1s) is the sum of the two neighbors just above it. This observation leads to the Pascal's Identity.

Theorem 9.95 (Pascal's Identity)**.** *Let n and k be positive integers with $k \leq n$. Then*

$$\binom{n+1}{k} = \binom{n}{k-1} + \binom{n}{k}.$$

Proof: *See Problem 9.21.* □

9.5 Inclusion-Exclusion

The Sum Rule (Theorem 9.1) gives us the cardinality for unions of finite sets that are mutually disjoint. In this section we will drop the disjointness requirement and obtain a formula for the cardinality of unions of general finite sets.

The Principle of *Inclusion-Exclusion* is attributed to both Sylvester and to Poincaré. We will only consider the cases involving two and three sets, although the principle easily generalizes to k sets.

This section is rough. I combined material from 2 places. I still need to smooth it out.

Theorem 9.96 (Inclusion-Exclusion for Two Sets)**.** *Let A and B be sets. Then*

$$|A \cup B| = |A| + |B| - |A \cap B|$$

Proof: *Clearly there are $|A \cap B|$ elements that are in both A and B. Therefore, $|A| + |B|$ is the number of element in A and B, where the elements in $|A \cap B|$ are counted twice. From this it is clear that $|A \cup B| = |A| + |B| - |A \cap B|$.* □

Example 9.97. Of 40 people, 28 smoke and 16 chew tobacco. It is also known that 10 both smoke and chew. How many among the 40 neither smoke nor chew?

Solution: Let A denote the set of smokers and B the set of chewers. Then

$$|A \cup B| = |A| + |B| - |A \cap B| = 28 + 16 - 10 = 34,$$

meaning that there are 34 people that either smoke or chew (or possibly both). Therefore the number of people that neither smoke nor chew is $40 - 34 = 6$.

⋆**Exercise 9.98.** In a group of 100 camels, 46 eat wheat, 57 eat barley, and 10 eat neither. How many camels eat both wheat and barley?

Example 9.99. Consider the set A that are multiples of 2 no greater than 114. That is,

$$A = \{2, 4, 6, \ldots, 114\}.$$

(a) How many elements are there in A?

(b) How many are divisible by 3?

(c) How many are divisible by 5?

(d) How many are divisible by 15?

(e) How many are divisible by either 3, 5 or both?

(f) How many are neither divisible by 3 nor 5?

(g) How many are divisible by exactly one of 3 or 5?

Solution: Let $A_k \subset A$ be the set of those integers divisible by k.

(a) Notice that the elements are $2 = 2(1)$, $4 = 2(2)$, $\ldots$, $114 = 2(57)$. Thus $|A| = 57$.

(b) Notice that

$$A_3 = \{6, 12, 18, \ldots, 114\} = \{1 \cdot 6, 2 \cdot 6, 3 \cdot 6, \ldots, 19 \cdot 6\},$$

so $|A_3| = 19$.

(c) Notice that

$$A_5 = \{10, 20, 30, \ldots, 110\} = \{1 \cdot 10, 2 \cdot 10, 3 \cdot 10, \ldots, 11 \cdot 10\},$$

so $|A_5| = 11$.

(d) Notice that $A_{15} = \{30, 60, 90\}$, so $|A_{15}| = 3$.

(e) First notice that $A_3 \cap A_5 = A_{15}$. Then it is clear that the answer is $|A_3 \cup A_5| = |A_3| + |A_5| - |A_{15}| = 19 + 11 - 3 = 27$.

(f) We want

$$|A \setminus (A_3 \cup A_5)| = |A| - |A_3 \cup A_5| = 57 - 27 = 30.$$

(g) We want

$$\begin{aligned} |(A_3 \cup A_5) \setminus (A_3 \cap A_5)| &= |(A_3 \cup A_5)| - |A_3 \cap A_5| \\ &= 27 - 3 \\ &= 24. \end{aligned}$$

We will use the following somewhat intuitive result in several examples.

Theorem 9.100. *Let k and n be a positive integers with $k < n$. Then there are $\lfloor \frac{n}{k} \rfloor$ numbers between 1 and n, inclusive, that are divisible by k.*

⋆**Exercise 9.101.** How many integers between 1 and 1000 inclusive, do not share a common factor with 1000, that is, are relatively prime to 1000? (Hint: 1000 only has 2 prime factors. Start by using inclusion/exclusion to count the number of numbers that have either of these as a factor.)

We now derive a three-set version of the Principle of Inclusion-Exclusion.

Theorem 9.102 (Inclusion-Exclusion for Three Sets)**.** *Let A, B, and C be sets. Then*

$$\begin{aligned} |A \cup B \cup C| &= |A| + |B| + |C| \\ &\quad -|A \cap B| - |B \cap C| - |C \cap A| \\ &\quad +|A \cap B \cap C| \end{aligned}$$

Proof: *Using the associativity and distributivity of unions of sets, we see that*

$$\begin{aligned} |A \cup B \cup C| &= |A \cup (B \cup C)| \\ &= |A| + |B \cup C| - |A \cap (B \cup C)| \\ &= |A| + |B \cup C| - |(A \cap B) \cup (A \cap C)| \\ &= |A| + |B| + |C| - |B \cap C| - |A \cap B| - |A \cap C| + |(A \cap B) \cap (A \cap C)| \\ &= |A| + |B| + |C| - |B \cap C| - (|A \cap B| + |A \cap C| - |A \cap B \cap C|) \\ &= |A| + |B| + |C| - |A \cap B| - |B \cap C| - |C \cap A| + |A \cap B \cap C|. \end{aligned}$$

□

Example 9.103. At *Medieval High* there are forty students. Amongst them, fourteen like Mathematics, sixteen like theology, and eleven like alchemy. It is also known that seven like Mathematics and theology, eight like theology and alchemy and five like Mathematics and alchemy. All three subjects are favored by four students. How many students like neither Mathematics, nor theology, nor alchemy?

Solution: Let A be the set of students liking Mathematics, B the set of students liking theology, and C be the set of students liking alchemy. We are given that

$$|A| = 14, |B| = 16, |C| = 11, |A \cap B| = 7, |B \cap C| = 8, |A \cap C| = 5,$$

and

$$|A \cap B \cap C| = 4.$$

Using Theorem 9.102, along with some set identities, we can see that

$$\begin{aligned} |\overline{A} \cap \overline{B} \cap \overline{C}| &= |\overline{A \cup B \cup C}| \\ &= |U| - |A \cup B \cup C| \\ &= |U| - |A| - |B| - |C| + |A \cap B| + |A \cap C| + |B \cap C| - |A \cap B \cap C| \\ &= 40 - 14 - 16 - 11 + 7 + 5 + 8 - 4 \\ &= 15. \end{aligned}$$

$\star$**Exercise 9.104.** A survey of a group's viewing habits revealed the percentages that watch a given sports. The results are given below. Calculate the percentage of the group that watched none of the three sports.

28% gymnastics	14% gymnastics & baseball	8% all three sports
29% baseball	10% gymnastics & soccer	
19% soccer	12% baseball & soccer	

Example 9.105. How many integers between 1 and 600 inclusive are not divisible by 3, nor 5, nor 7?

Solution: Let A_k denote the numbers in $[1, 600]$ which are divisible by $k = 3, 5, 7$. Then

$$\begin{aligned}
|A_3| &= \left\lfloor \frac{600}{3} \right\rfloor = 200, \\
|A_5| &= \left\lfloor \frac{600}{5} \right\rfloor = 120, \\
|A_7| &= \left\lfloor \frac{600}{7} \right\rfloor = 85, \\
|A_{15}| &= \left\lfloor \frac{600}{15} \right\rfloor = 40, \\
|A_{21}| &= \left\lfloor \frac{600}{21} \right\rfloor = 28, \\
|A_{35}| &= \left\lfloor \frac{600}{35} \right\rfloor = 17, \text{ and} \\
|A_{105}| &= \left\lfloor \frac{600}{105} \right\rfloor = 5.
\end{aligned}$$

By Inclusion-Exclusion there are $200+120+85-40-28-17+5 = 325$ integers in $[1, 600]$ divisible by at least one of 3, 5, or 7. Those not divisible by these numbers are a total of $600 - 325 = 275$.

$\star$**Exercise 9.106.** Would you believe a market investigator that reports that of 1000 people, 816 like candy, 723 like ice cream, 645 like cake, while 562 like both candy and ice cream, 463 like both candy and cake, 470 like both ice cream and cake, while 310 like all three? State your reasons!

Example 9.107. An auto insurance company has $10,000$ policyholders. Each policy holder is classified as

- young or old,
- male or female, and
- married or single.

Of these policyholders, 3000 are young, 4600 are male, and 7000 are married. The policyholders can also be classified as 1320 young males, 3010 married males, and 1400 young married persons. Finally, 600 of the policyholders are young married males.

How many of the company's policyholders are young, female, and single?

Solution: Let Y, F, S, M stand for young, female, single, male, respectively, and let Ma stand for married. We have

$$\begin{aligned} |Y \cap F \cap S| &= |Y \cap F| - |Y \cap F \cap Ma| \\ &= |Y| - |Y \cap M| \\ &\qquad -(|Y \cap Ma| - |Y \cap Ma \cap M|) \\ &= 3000 - 1320 - (1400 - 600) \\ &= 880. \end{aligned}$$

The following problem is a little more challenging than the others we have seen, but you have all of the tools you need to tackle it.

$\star$**Exercise 9.108** (Lewis Carroll in *A Tangled Tale.*)**.** In a very hotly fought battle, at least 70% of the combatants lost an eye, at least 75% an ear, at least 80% an arm, and at least 85% a leg. What can be said about the percentage who lost all four members?

9.6 Problems

Problem 9.1. How many license plates can be made using either three letters followed by three digits or four letters followed by two digits?

Problem 9.2. How many license plates can be made using 4 letters and 3 numbers if the letters cannot be repeated?

Problem 9.3. How many bit strings of length 8 either begin with three 1s or end with four 0s?

Problem 9.4. How many alphabetic strings are there whose length is at most 5?

Problem 9.5. How many bit strings are there of length at least 4 and at most 6?

Problem 9.6. How many subsets with 4 or more elements does a set of size 30 have?

Problem 9.7. Given a group of ten people, prove that at least 4 are male or at least 7 are female.

Problem 9.8. My family wants to take a group picture. There are 7 men and 5 women, and we want none of the women to stand next to each other. How many different ways are there for us to line up?

Problem 9.9. My family (7 men and 5 women) wants to select a group of 5 of us to plan Christmas. We want at least 1 man and 1 woman in the group. How many ways are there for us to select the members of this group?

Problem 9.10. Compute each of the following: $\binom{8}{4}$, $\binom{9}{9}$, $\binom{7}{3}$, 8!, and 5!

Problem 9.11. For what value(s) of k is $\binom{18}{k}$ largest? smallest?

Problem 9.12. For what value(s) of k is $\binom{19}{k}$ largest? smallest?

Problem 9.13. A computer network consists of 10 computers. Each computer is directly connected to zero or more of the other computers.

(a) Prove that there are at least two computers in the network that are directly connected to the same number of other computers.

(b) Prove that there are an even number of computers that are connected to an odd number of other computers.

Problem 9.14. Simplify the following expression so it does not involve any factorials or binomial coefficients: $\binom{x}{y}/\binom{x+1}{y-1}$.

Problem 9.15. Prove that amongst six people in a room there are at least three who know one another, or at least three who do not know one another.

Problem 9.16. Suppose that the letters of the English alphabet are listed in an arbitrary order.

(a) Prove that there must be four consecutive consonants.

(b) Give a list to show that there need not be five consecutive consonants.

(c) Suppose that all the letters are arranged in a circle. Prove that there must be five consecutive consonants.

Problem 9.17. Bob has ten pockets and forty four silver dollars. He wants to put his dollars into his pockets so distributed that each pocket contains a different number of dollars.

(a) Can he do so?

(b) Generalize the problem, considering p pockets and n dollars. Why is the problem most interesting when $n = \dfrac{(p-1)(p-2)}{2}$?

Problem 9.18. Expand and simplify the following.

(a) $(x-4y)^3$

(b) $(x^3+y^2)^4$

(c) $(2+3x)^6$

(d) $(2i-3)^5$

(e) $(2i+3)^4+(2i-3)^4$

(f) $(2i+3)^4-(2i-3)^4$

(g) $(\sqrt{3}-\sqrt{2})^3$

(h) $(\sqrt{3}+\sqrt{2})^3+(\sqrt{3}-\sqrt{2})^3$

(i) $(\sqrt{3}+\sqrt{2})^3-(\sqrt{3}-\sqrt{2})^3$

Problem 9.19. What is the coefficient of x^6y^9 in $(3x-2y)^{15}$?

Problem 9.20. What is the coefficient of x^4y^6 in $(x\sqrt{2}-y)^{10}$?

Problem 9.21. Prove Pascal's Identity (Theorem 9.95). (Hint: Just use the definition of the binomial coefficient and do a little algebra.)

Problem 9.22. Prove that for any positive integer n, $\displaystyle\sum_{k=0}^{n}(-2)^k\binom{n}{k} = (-1)^n$. (Hint: *Don't* use induction.)

Problem 9.23. Expand and simplify

$$(\sqrt{1-x^2}+1)^7-(\sqrt{1-x^2}-1)^7.$$

Problem 9.24. There are approximately 7,000,000,000 people on the planet. Assume that everyone has a name that consists of exactly k lower-case letters from the English alphabet.

(a) If $k=8$, is it guaranteed that two people have the same name? Explain.

(b) What is the maximum value of k that would guarantee that at least two people have the same name?

(c) What is the maximum value of k that would guarantee that at least 100 people have the same name?

(d) Now assume that names can be between 1 and k characters long. What is the maximum value of k that would guarantee that at least two people have the same name?

Problem 9.25. Password cracking is the process of determining someone's password, typically using a computer. One way to crack passwords is to perform an exhaustive search that tries every possible string of a given length until it (hopefully) finds it. Assume your computer can test 10,000,000 passwords per second. How long would it take to crack passwords with the following restrictions? Give answers in seconds, minutes, hours, days, or years depending on how large the answer is (e.g. 12,344,440 seconds isn't very helpful). Start by determining how many possible passwords there are in each case.

(a) 8 lower-case alphabetic characters.

(b) 8 alphabetic characters (upper or lower).

(c) 8 alphabetic (upper or lower) and numeric characters.

(d) 8 alphabetic (upper or lower), numeric characters, and special characters (assume there are 32 allowable special characters).

(e) 8 or fewer alphabetic (upper or lower) and numeric characters.

(f) 10 alphabetic (upper or lower), numeric characters, and special characters (assume there are 32 allowable special characters).

(g) 8 characters, with at least one upper-case, one lower-case, one number, and one special character.

Problem 9.26. IP addresses are used to identify computers on a network. In IPv4, IP addresses are 32 bits long. They are usually written using dotted-decimal notation, where the 32 bits are split up into 4 8-bit segments, and each 8-bit segment is represented in decimal. So the IP address 10000001 11000000 00011011 00000100 is represented as $129.192.27.4$. The *subnet mask* of a network is a string of k ones followed by $32-k$ zeros, where the value of k can be different on different networks. For instance, the subnet mask might be 11111111111111111111111100000000, which is $255.255.255.0$ in dotted decimal. To determine the *netid*, an IP address is bitwise ANDed with the subnet mask. To determine the *hostid*, an IP address is bitwise ANDed with the bitwise complement of the subnet mask. Since every computer on a network needs to have a different *hostid*, the number of possible *hostid*s determines the maximum number of computers that can be on a network.

Assume that the subnet mask on my computer is currently $255.255.255.0$ and my IP address is $209.140.209.27$.

(a) What are the *netid* and *hostid* of my computer?

(b) How many computers can be on the network that my computer is on?

(c) In 2010, Hope College's network was not split into subnetworks like it is currently, so all of the computers were on a single network that had a subnet mask of $255.255.240.0$. How many computers could be on Hope's network in 2010?

Problem 9.27. Prove that $\displaystyle\sum_{k=0}^{n}\binom{n}{k} = 2^n$ by counting the number of binary strings of length n in two ways.

Problem 9.28. In March of every year people fill out brackets for the NCAA Basketball Tournament. They pick the winner of each game in each round. We will assume the tournament starts with 64 teams (it has become a little more complicated than this recently). The first round of the tournament consists of 32 games, the second 16 games, the third 8, the fourth 4, the fifth 2, and the final 1. So the total number of games is $32 + 16 + 8 + 4 + 2 + 1 = 63$. You can arrive at the number of games in a different way. Every game has a loser who is out of the tournament. Since only 1 of the 64 teams remains at the end, there must be 63 losers, so there must be 63 games. Notice that we can also write $1 + 2 + 4 + 8 + 16 + 32 = 63$ as $\displaystyle\sum_{k=0}^{5} 2^k = 2^6 - 1$.

(a) Use a combinatorial proof to show that for any $n > 0$, $\displaystyle\sum_{k=0}^{n} 2^k = 2^{n+1} - 1$. (That is, define an appropriate set and count the cardinality of the set in two ways to obtain the identity.)

(b) When you fill out a bracket you are picking who you think the winner will be of each game. How many different ways are there to fill out a bracket? (Hint: If you think about this in the proper way, this is pretty easy.)

(c) If everyone on the planet (7,000,000,000) filled out a bracket, is it guaranteed that two people will have the same bracket? Explain.

(d) Assume that everyone on the planet fills out k different brackets and that no brackets are repeated (either by an individual or by anybody else). How large would k have to be before it is guaranteed that somebody has a bracket that correctly predicts the winner of every game?

(e) Assume every pair of people on the planet gets together to fill out a bracket (so everyone has 6,999,999 brackets, one with every other person on the planet). What is the smallest and largest number of possible repeated brackets?

Problem 9.29. Mega Millions has 56 white balls, numbered 1 through 56, and 46 red balls, numbered 1 through 46. To play you pick 5 numbers between 1 and 56 (corresponding to white balls) and 1 number between 1 and 46 (corresponding to a red ball). Then 5 of the 56 balls and 1 of the 46 balls are drawn randomly (or so they would have us believe). You win if your numbers match all 6 balls.

(a) How many different draws are possible?

(b) If everyone in the U.S.A. bought a ticket (about 314,000,000), is it guaranteed that two people have the same numbers? Three people?

(c) If everyone in the U.S.A. bought a ticket, what is the maximum number of people that are guaranteed to share the jackpot?

(d) Which is more likely: Winning Mega Millions or picking every winner in the NCAA Basketball Tournament (see previous question)? How many more times likely is one than the other?

(e) I purchased a ticket last week and was surprised when *none* of my six numbers matched. Should I have been surprised? What are the chances that a randomly selected ticket will match none of the numbers?

(f) (hard) What is the largest value of k such that you are more likely to pick at least k winners in the NCAA Basketball Tournament than you are to win Mega Millions?

Problem 9.30. You get a new job and your boss gives you 2 choices for your salary. You can either make \$100 per day or you can start at \$.01 on the first day and have your salary doubled every day. You know that you will work for k days. For what values of k should you take the first offer and for which should you take the second offer? Explain.

Problem 9.31. The 300-level courses in the CS department are split into three groups: Foundations (361, 385), Applications (321, 342, 392), and Systems (335, 354, 376). In order to get a BS in computer science at Hope you need to take at least one course from each group.

(a) How many different ways are there of satisfying this requirement by taking exactly 3 courses?

(b) If you take four 300-level courses, how many different possibilities do you have that satisfy the requirements?

(c) How many total ways are there to take 300-level courses that satisfy the requirements?

(d) What is the smallest k such that no matter which k 300-level courses you choose, it is guaranteed that you will satisfy the requirement?

Problem 9.32. I am implementing a data structure that consists of k lists. I want to store a total of n objects in this data structure, with each item being stored on one of the lists. All of the lists will have the same capacity (e.g. perhaps each list can hold up to 10 elements).
Write a method `minimumCapacity(int n, int k)` that computes the minimum capacity each of the k lists must have to accommodate n objects. In other words, if the capacity is less than this, then there is no way the objects can all be stored on the lists. You may assume integer arithmetic truncates (essentially giving you the *floor* function), but that there is no *ceiling* function available.

Problem 9.33. Write a method `choose(int n, int k)` (in a Java-like language) that computes $\binom{n}{k}$. Your implementation should be as efficient as possible.

Chapter 10

Graph Theory

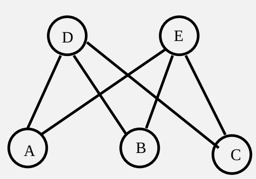
In this chapter we will provide a *very brief* and *very selective* introduction to graphs. Graph theory is a very wide field and there are many thick textbooks on the subject. The main point of this chapter is to provide you with the basic notion of what a graph is, some of the terminology used, a few applications, and a few interesting and/or important results.

10.1 Types of Graphs

Definition 10.1. *A* **(simple) graph** $G = (V, E)$ *consists of*

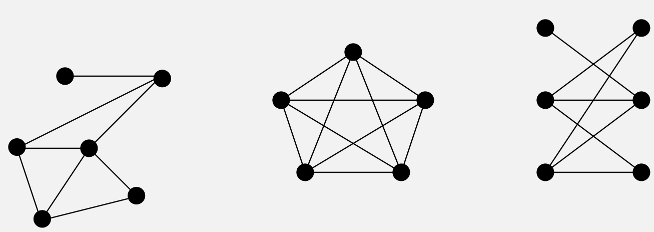
- V*, a nonempty set of* **vertices** *and*
- E*, a set of* unordered pairs *of distinct vertices called* **edges***.*

Example 10.2. Here is an example of a graph with the set of vertices and edges listed on the right. Vertices are usually represented by means of dots on the plane, and the edges by means of lines connecting these dots.



V={A,B,C,D,E}
E={ (A,D),(A,E),(B,D),
(B,E),(C,D),(C,E)}

Example 10.3. Sometimes we just care about the visual representation of a graph. Here are three examples.

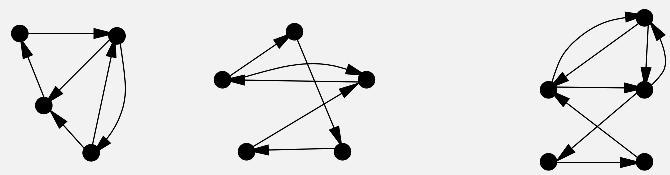


There are several variations of graphs. We will provide definitions and examples of the most common ones.

Definition 10.4. *A* **directed graph** *(or* **digraph***)* $G = (V, E)$ *consists of*

- V*, a nonempty set of* **vertices** *and*
- E*, a set of* ordered pairs *of distinct vertices called* **directed edges** *(or sometimes just* **edges***).*

Example 10.5. Here are three examples of directed graphs.



As you would probably suspect, the only difference between simple graphs and directed graphs is that the edges in directed graphs have a direction. We should note that simple graphs are sometimes called **undirected graphs** to make it clear that the graphs are not directed.

Example 10.6. In a simple graph, $\{u, v)\}$ and $(\{v, u\}$ are just two different ways of talking about the same edge–the edge between u and v. In a directed graph, (u, v) and (v, u) are different edges, and they may or may not both be present.

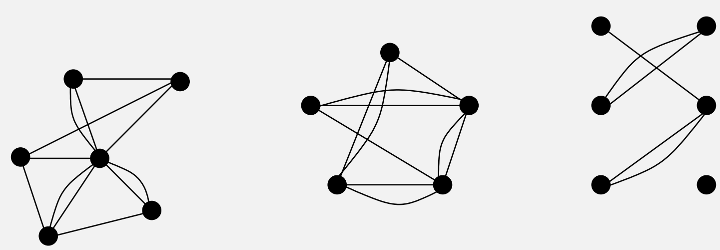
Definition 10.7. *A* **multigraph (directed multigraph)** $G = (V, E)$ *consists of*

- V*, a set of vertices,*
- E*, a set of edges, and*
- *a function* f *from* E *to* $\{\{u, v\} : u \neq v \in V\}$
  (function f *from* E *to* $\{(u, v) : u \neq v \in V\}$*.)*

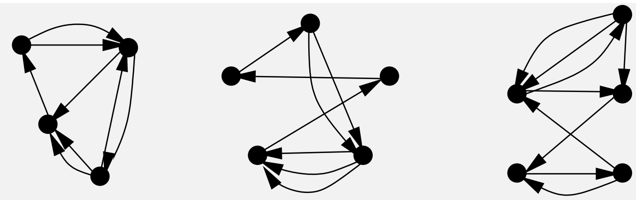
Two edges e_1 *and* e_2 *with* $f(e_1) = f(e_2)$ *are called* **multiple edges***.*

Although the definition looks a bit complicated, a **multigraph** $G = (V, E)$ is just a graph in which multiple edges are allowed between a pair of vertices.

Example 10.8. Here are a few examples of multigraphs.

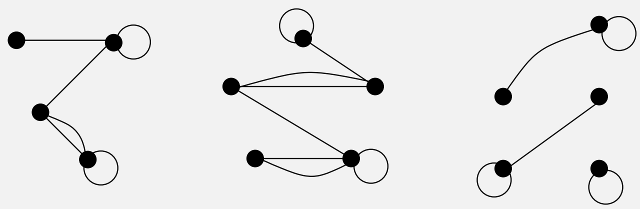


Here are some examples of directed multigraphs.

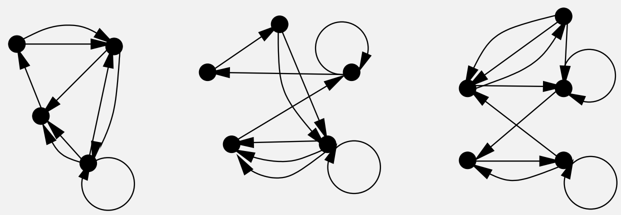


Definition 10.9. *A* **pseudograph** $G = (V, E)$ *is a graph in which we allow* **loops**–*that is, edges from a vertex to itself. As you might imagine, a* **pseudo-multigraph** *allows both loops and multiple edges.*

Example 10.10. Here are some pseudographs.



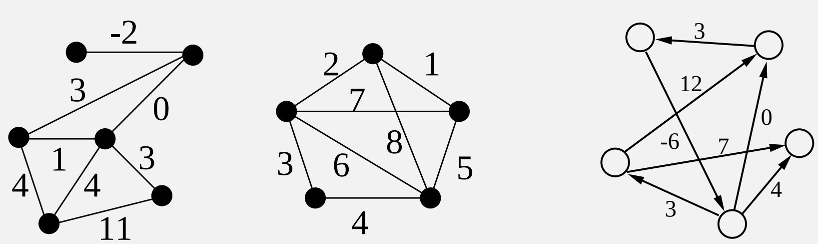
Here are a few directed pseudographs.



Definition 10.11. *A* **weighted graph** *is a graph (or digraph) with the additional property that each edge e has associated with it a real number $w(e)$ called it's* weight.

A **weighted digraph** *is often called a* **network**.

Example 10.12. Here are two examples of weighted graphs and one weighted directed graph.



As we have seen, there are several ways of categorizing graphs:

- Directed or undirected edges.
- Weighted or unweighted edges.

- Allow multiple edges or not.
- Allow loops or not.

Unless specified, you can usually assume a graph does not allow multiple edges or loops since these aren't that common. Generally speaking, you can assume that if a graph is not specified as weighted or directed, it isn't. The most common graphs we'll use are graphs, digraphs, weighted graphs, and networks.

Note: *When writing graph algorithms, it is important to know what characteristics the graphs have. For instance, if a graph might have loops, the algorithm should be able to handle it. Some algorithms do not work if a graph has loops and/or multiple edges, and some only apply to directed (or undirected) graphs.*

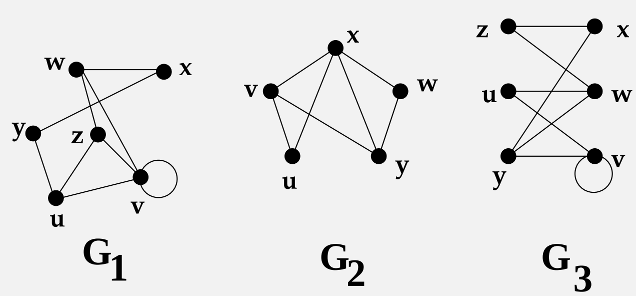
10.2 Graph Terminology

Definition 10.13. *Given a graph $G = (V, E)$, we denote the number of vertices in G by $|V|$ and the number of edges by $|E|$ (a notation that makes perfect sense since V and E are sets).*

Definition 10.14. *Let u and v be vertices and $e = \{u, v\}$ be an edge in undirected graph G.*

- *The vertices u and v are said to be* **adjacent**
- *The vertices u and v are called the* **endpoints** *of the edge e.*
- *The edge e is said to be* **incident with** *u and v.*
- *The edge e is said to* **connect** *u and v.*
- *The* **degree** *of a vertex, denoted $deg(v)$, is the number of edges incident with it.*

Example 10.15. Consider the following graphs.



In graph G_1, we can say:

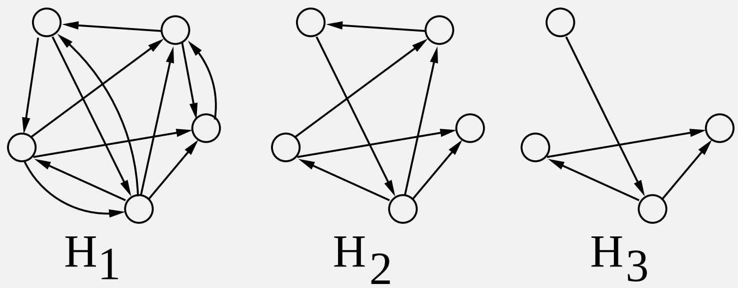
- w is adjacent to x.
- w and x are the endpoints of the edge (w, x).
- (w, x) is incident with both w and x.
- (w, x) connects vertices w and x.

The following table gives the degree of each of the vertices in the graphs above.

G_1	G_2	G_3
deg(u)=3	deg(u)=2	deg(u)=2
deg(v)=5	deg(v)=3	deg(v)=4
deg(w)=3	deg(w)=2	deg(w)=3
deg(x)=2	deg(x)=4	deg(x)=2
deg(y)=2	deg(y)=3	deg(y)=3
deg(z)=3		deg(z)=2

Definition 10.16. *A* **subgraph** *of a graph $G = (V, E)$ is a graph $G' = (V', E')$ such that $V' \subset V$ and $E' \subset E$.*

Example 10.17. Consider the following three graphs:



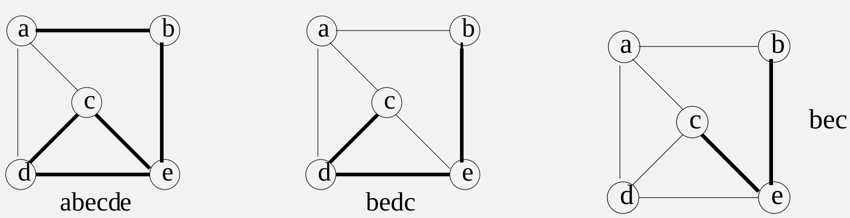
Notice that H_2 is a subgraph of H_1 and that H_3 is a subgraph of both H_1 and H_2.

Definition 10.18. *A $u - v$* **walk** *is an alternating sequence of vertices and edges in G with starting vertex u and ending vertex v such that every edge joins the vertices immediately preceding it and immediately following it.*

Definition 10.19. *A $u - v$* **trail** *is a $u - v$ walk that does not repeat an edge.*

Definition 10.20. *A $u - v$* **path** *is a walk that does not repeat any vertex.*

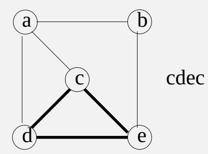
Example 10.21. In the first example below, the trail *abecde* is indicated with the dark lines. It is not a path since it repeats the vertex *e*. The second and third graphs show examples of paths.



Definition 10.22. *A* **cycle** *(or* **simple cycle***) is a path to which we append the first vertex. In other words, it is a path that ends where it started.*

The number of vertices in a cycle is called its **length***.*

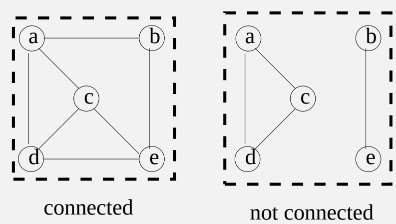
Example 10.23. Here is a graph with a cycle of length 3.



Definition 10.24. *A graph is called* **connected** *if there is a path between every pair of distinct vertices.*

A **connected component** *of a graph is a maximal connected subgraph.*

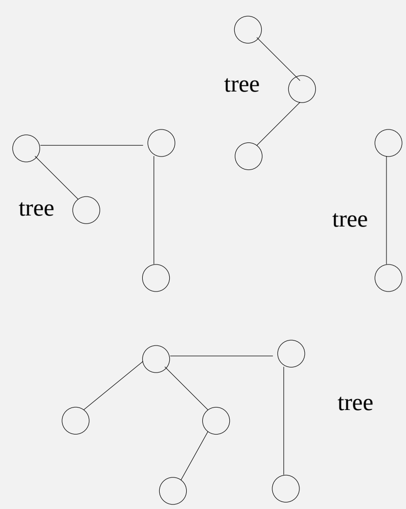
Example 10.25. Below are two graphs, each drawn inside dashed boxes. The graph on the left is connected. The one on the right is not connected. It has two connected components.



Definition 10.26. *A* **tree** *(or* **unrooted tree***) is a connected acyclic graph. That is, a graph with no cycles.*

A **forest** *is a collection of trees.*

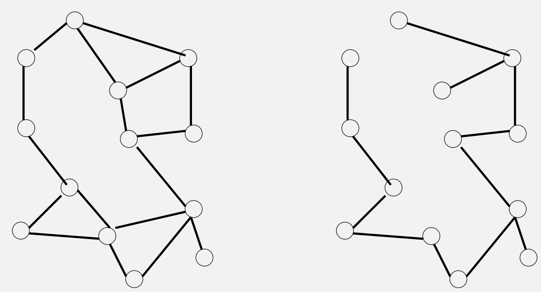
Example 10.27. Here are four trees. If they were all part of the same graph, we could consider the graph a forest.



Note: *These trees are not to be confused with* rooted trees *(e.g.* binary trees*). When computer scientists use the term* tree*, they usually mean rooted trees, not the trees we are discussing here. When you see/hear the term 'tree,' it is important to be clear about which one the writer/speaker has in mind.*

Definition 10.28. *A* **spanning tree** *of G is a subgraph which is a tree and contains all of the vertices of G.*

Example 10.29. Below is a graph (on the left) and one of several possible spanning trees (on the right).



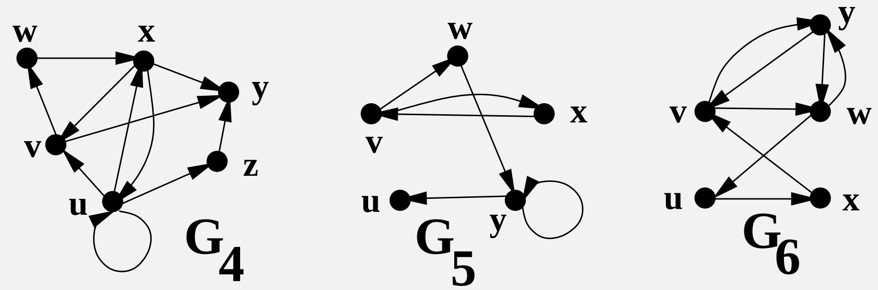
G spanning tree of G

Here is some terminology related to directed graphs.

Definition 10.30. *Let u and v be vertices in a directed graph G, and let $e = (u, v)$ be an edge in G.*

- *u is said to be* **adjacent to** *v.*
- *v is said to be* **adjacent from** *u.*
- *u is called the* **initial vertex** *of (u, v).*
- *v is called the* **terminal** *or* **end vertex** *of (u, v).*
- *The* **in-degree** *of u, denoted by $\deg^-(u)$, is the number of edges in G which have u as their terminal vertex.*
- *The* **out-degree** *of u, denoted by $\deg^+(u)$, is the number of edges in G which have u as their initial vertex.*

Example 10.31. Consider the three graphs below.



Consider the edge (w, x) in G_4.

- w is adjacent to x and x is adjacent from w.
- w is the initial vertex and x is the terminal vertex of the edge (w, x).

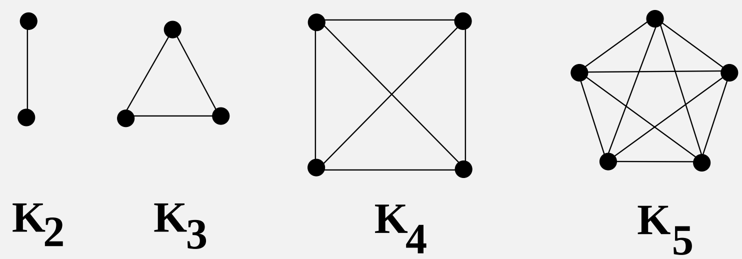
The following table gives the in-degree and out-degree for the vertices in graphs G_4, G_5, and G_6.

G_4		G_5		G_6	
$\deg^-$(u)=2	$\deg^+$(u)=4	$\deg^-$(u)=1	$\deg^+$(u)=0	$\deg^-$(u)=1	$\deg^+$(u)=1
$\deg^-$(v)=2	$\deg^+$(v)=2	$\deg^-$(v)=1	$\deg^+$(v)=2	$\deg^-$(v)=2	$\deg^+$(v)=2
$\deg^-$(w)=1	$\deg^+$(w)=1	$\deg^-$(w)=1	$\deg^+$(w)=1	$\deg^-$(w)=2	$\deg^+$(w)=2
$\deg^-$(x)=2	$\deg^+$(x)=3	$\deg^-$(x)=1	$\deg^+$(x)=1	$\deg^-$(x)=1	$\deg^+$(x)=1
$\deg^-$(y)=3	$\deg^+$(y)=0	$\deg^-$(y)=2	$\deg^+$(y)=2	$\deg^-$(y)=2	$\deg^+$(y)=2
$\deg^-$(z)=1	$\deg^+$(z)=1				

10.3 Some Special Graphs

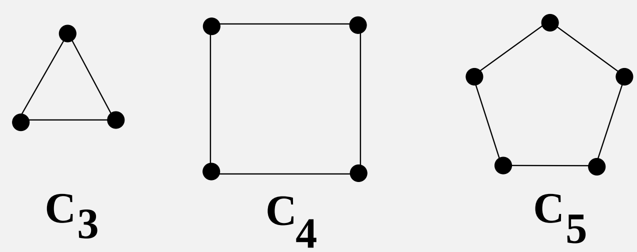
Definition 10.32. *The* **complete graph** *with n vertices K_n is the graph where every pair of vertices is adjacent. Thus K_n has $\binom{n}{2}$ edges.*

Example 10.33. Here are the complete graphs with $n = 2, 3, 4, 5$.



Definition 10.34. *C_n denotes a* **cycle** *of length n. It is a graph with n edges, and n vertices $v_1 \cdots v_n$, where v_i is adjacent to v_{i+1} for $n = 1, \ldots, n-1$, and v_1 is adjacent to v_n.*

Example 10.35. Here are the cycles of length 3, 4, and 5.

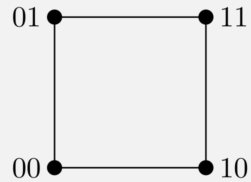


Definition 10.36. *P_n denotes a* path *of length n. It is a graph with n edges, and $n+1$ vertices $v_0 v_1 \cdots v_n$, where v_i is adjacent to v_{i+1} for $n = 0, 1, \ldots, n-1$.*

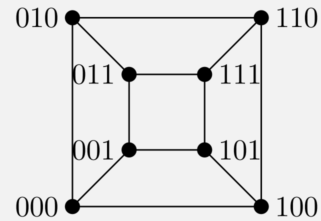
We won't provide an example of the paths, but they are pretty easy to visualize. For instance, P_3 is simply C_4 with one edge removed.

Definition 10.37. *Q_n denotes the n-***dimensional cube** *(or* **hypercube***). One way to define Q_n is that it is a simple graph with 2^n vertices, which we label with n-tuples of 0's and 1's. Vertices of Q_n are connected by an edge if and only if they differ by exactly one coordinate. Observe that Q_n has $n2^{n-1}$ edges.*

Example 10.38. Here are Q_2 and Q_3 labeled as mentioned in the definition.



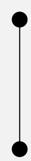
$\mathbf{Q_2}$



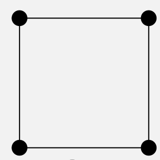
$\mathbf{Q_3}$

Notice that in Q_2, the vertex labeled 11 is adjacent to the vertices labeled 10 and 01 since each of these differ in one bit. Similarly, the vertex labeled 101 in Q_3 is adjacent to the vertices labeled 001, 111, and 100 for the same reason.

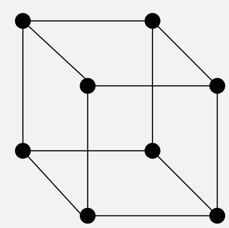
Example 10.39. Here are the (unlabeled) hypercubes with dimensions 1, 2, 3, and 4.



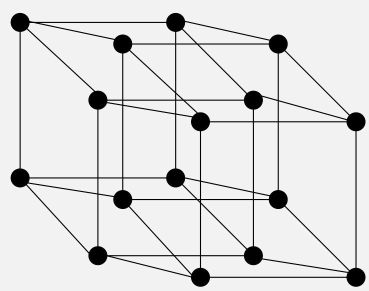
$\mathbf{Q_1}$



$\mathbf{Q_2}$



$\mathbf{Q_3}$



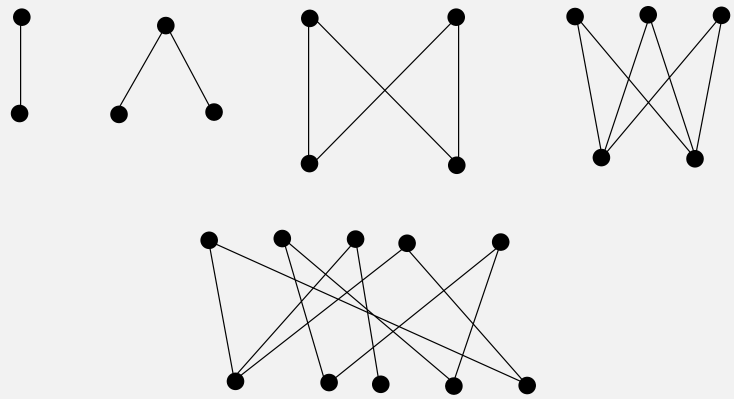
$\mathbf{Q_4}$

Definition 10.40. *A simple graph G is called* **bipartite** *if the vertex set V can be partitioned into two disjoint nonempty sets V_1 and V_2 such that every edge connects a vertex in V_1 to a vertex in V_2.*

Put another way, no vertices in V_1 are connected to each other, and no vertices in V_2 are connected to each other.

Note that there may be different ways of assigning the vertices to V_1 and V_2. That is not important. As long as there is at least one way to do so such that all edges go between V_1 and V_2, then a graph is bipartite.

Example 10.41. Here are a few bipartite graphs.



Notice that although these are drawn to make it clear what the partition is (i.e. V_1 is the top row of vertices and V_2 is the bottom row), a graph does not have to be drawn as such in order to be bipartite. They are often drawn this way out of convenience. For instance, the hypercubes are all bipartite even though they are not drawn this way.

Definition 10.42. *$K_{m,n}$ denotes the* **complete bipartite graph** *with $m+n$ vertices. That is, it is the graph with $m+n$ vertices that is partitioned into two sets, one of size n and the other of size m such that every possible edge between the two sets is in the graph.*

Example 10.43. The first four graphs from Example 10.41 are complete bipartite graphs. The first is $K_{1,1}$, the second is $K_{1,2}$ (or $K_{2,1}$), the third is $K_{2,2}$, and the fourth is $K_{3,2}$ (or $K_{2,3}$).

10.4 Handshaking Lemma

The following theorem is valid not only for simple graphs, but also for multigraphs and pseudographs.

Theorem 10.44 (Handshake Lemma)**.** *Let $G = (V, E)$ be a graph. Then*

$$\sum_{v \in V} \deg(v) = 2|E|.$$

Proof: *Let $X = \{(e, v) : e \in E, v \in V,$ and e and v are incident$\}$. We will compute $|X|$ in two ways. Each edge $e \in E$ is incident with exactly 2 vertices. Thus,*

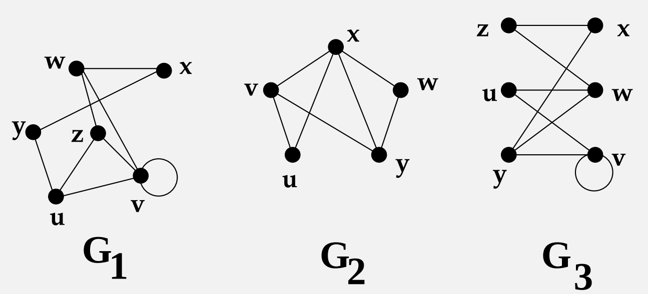
$$|X| = 2|E|.$$

Also, each vertex $v \in V$ is incident with deg(v) edges. Thus, we have that

$$|X| = \sum_{v \in V} deg(v).$$

Setting these equal, we have the result . □

Example 10.45. Consider the following graphs.



A quick tabulation of the degrees of the vertices and the number of edges reveals the following:

Graph	G_1	G_2	G_3
$\|E\|$	9	7	8
$\sum_{v \in V} deg(v)$	18	14	16

These results are certainly consistent with Theorem 10.44.

Corollary 10.46. *Every graph has an even number of vertices of odd degree.*

Proof: *The sum of an odd number of odd numbers is odd. Since the sum of the degrees of the vertices in a simple graph is always even, one cannot have an odd number of odd degree vertices.* □

The situation is slightly different, but not too surprising, for directed graphs.

Theorem 10.47. *Let $G = (V, E)$ be a directed graph. Then*

$$\sum_{v \in V} deg^{-}(v) = \sum_{v \in V} deg^{+}(v) = |E|.$$

We won't provide a proof of this theorem (it's almost obvious), but you should verify it for the graphs in Example 10.31 by adding up the degrees in each column and comparing the appropriate sums.

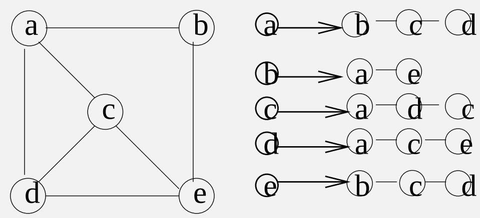
10.5 Graph Representation

Much could be said about representing graphs. We provide only a very brief discussion of the topic. Consult your favorite data structure book for more details.

Let $G = (V, E)$ be a graph with n vertices and m edges. There are two common ways of representing G.

Definition 10.48. *The* adjacency list *representation of a graph maintains, for each vertex, a list of all of the vertices adjacent to that vertex. This can be implemented in many ways, but often an array of linked lists is used.*

Example 10.49. The following shows a graph on the left with the adjacency list representation on the right.



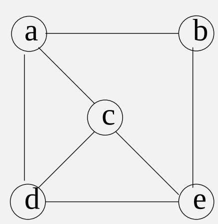
It should not be difficult to see that the space required for storage is approximately $n + 2m = \Theta(n + m)$ for graphs (since each edge occurs on two lists), and about $n + m = \Theta(n + m)$ for digraphs. For weighted graphs, an additional field can be stored in each node. Of course, this requires about m more space.

Definition 10.50. *We assume that the vertices are numbered* $0, 1, \ldots, n-1$. *The* adjacency matrix M *of a graph* G *is the* n *by* n *matrix* M *defined as*

$$M(i,j) = \begin{cases} 1 & \textit{if } (i,j) \textit{ is an edge} \\ 0 & \textit{if } (i,j) \textit{ is not an edge} \end{cases}$$

It should be clear that this representation requires about $n^2 = \Theta(n^2)$ space. If G is weighted, we can store the weights in the matrix instead of just 0 or 1. For non-adjacent vertices, we store ∞, or MAX_INT (or -1 if only positive weights are valid). Depending on the data type of the matrix, weighted graphs may take no more space to store than unweighted graphs.

Example 10.51. Here we see a graph on the left with the adjacency matrix representation on the right.



	a	b	c	d	e
a	0	1	1	1	0
b	1	0	0	0	1
c	1	0	0	1	1
d	1	0	1	0	1
e	0	1	1	1	0

10.6 Problem Solving with Graphs

There are many problems on graphs that are of interest for various reasons. The following very short list contains some of the more common ones.

- PATH: Is there a path from A to B?
- CYCLES: Does the graph contain a cycle?
- CONNECTIVITY (SPANNING TREE): Is there a way to connect the vertices?
- BICONNECTIVITY: Will the graph become disconnected if one vertex is removed?
- PLANARITY: Is there a way to draw the graph without edges crossing?
- SHORTEST PATH: What is the shortest way from A to B?
- LONGEST PATH: What is the longest way from A to B?
- MINIMUM SPANNING TREE: What is the best way to connect the vertices?
- TRAVELING SALESMAN: What is the shortest route to connect the vertices without visiting the same vertex twice?

Knowing what graph problems have been studied and what is known about each is very important. Many problems can be modeled using graphs, and once a problem has been mapped to a particular graph problem, it can be helpful to know the best way to solve it.

We will now give a few examples of problems whose solutions become simpler when using a graph-theoretic model.

Example 10.52. A wolf, a goat, and a cabbage are on one bank of a river. The ferryman wants to take them across, but his boat is too small to accommodate more than one of them. Evidently, he can neither leave the wolf and the goat, or the cabbage and the goat behind. Can the ferryman still get all of them across the river?

Solution: Represent the position of a single item by 0 for one bank of the river and 1 for the other bank. The position of the three items can now be given as an ordered triplet, say (W, G, C). For example, $(0, 0, 0)$ means that the three items are on one bank of the river, $(1, 0, 0)$ means that the wolf is on one bank of the river while the goat and the cabbage are on the other bank. The object of the puzzle is now seen to be to move from $(0, 0, 0)$ to $(1, 1, 1)$, that is, traversing Q_3 while avoiding certain edges. One answer is

$$000 \to 010 \to 011 \to 001 \to 101 \to 111.$$

This means that the ferryman (i) takes the goat across, (ii) returns and takes the cabbage over bringing back the goat, (iii) takes the wolf over, (iv) returns and takes the goat over. Another one is

$$000 \to 010 \to 110 \to 100 \to 101 \to 111.$$

This means that the ferryman (i) takes the goat across, (ii) returns and takes the wolf over bringing back the goat, (iii) takes the cabbage over, (iv) returns and takes the goat over. The graph depicting both answers can be seen in figure 10.1. Go to `http://www.cut-the-knot.org/ctk/GoatCabbageWolf.shtml` to see a pictorial representation of this problem.

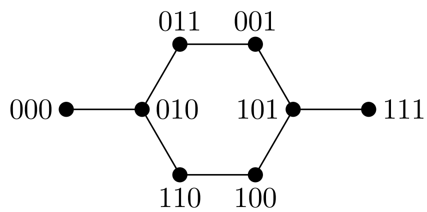


Figure 10.1: Example 10.52.

Example 10.53. Prove that amongst six people in a room there are at least three who know one another, or at least three who do not know one another.

Solution: In graph-theoretic terms, we need to show that every colouring of the edges of K_6 into two different colours, say red and blue, contains a monochromatic triangle (that is, the edges of the triangle have all the same colour). Consider an arbitrary person of this group (call him Peter). There are five other people, and of these, either three of them know Peter or else, three of them do not know Peter. Let us assume three do know Peter, as the alternative is argued similarly. If two of these three people know one another, then we have a triangle (Peter and these two, see figure 10.2, where the acquaintances are marked by solid lines). If no two of these three people know one another, then we have three mutual strangers, giving another triangle (see figure 10.3).

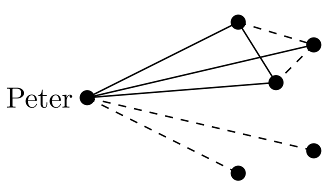


Figure 10.2: Example 10.53.

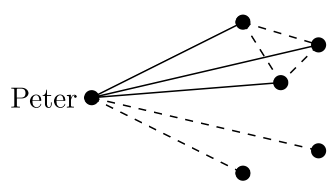


Figure 10.3: Example 10.53.

Example 10.54. Mr. and Mrs. Landau invite four other married couples for dinner. Some people shook hands with some others, and the following rules were noted: (i) a person did not shake hands with himself, (ii) no one shook hands with his spouse, (iii) no one shook hands more than once with the same person. After the introductions, Mr. Landau asks the nine people how many hands they shook. Each of the nine people asked gives a different number. How many hands did Mrs. Landau shake?

Solution: The given numbers can either be $0, 1, 2, \ldots, 8$, or $1, 2, \ldots, 9$. Now, the sequence $1, 2, \ldots, 9$ must be ruled out, since if a person shook hands nine times, then he must have shaken hands with his spouse, which is not allowed. The only permissible sequence is thus $0, 1, 2, \ldots, 8$. Consider the person who shook hands 8 times, as in figure 10.4. Discounting himself and his spouse, he must have shaken hands with everybody else. This means that he is married to the person who shook 0 hands! We now consider the person that shook 7 hands, as in figure 10.5. He didn't shake hands with himself, his spouse, or with the person that shook 0 hands. But the person that shook hands only once did so with the person shaking 8 hands. Thus the person that shook hands 7 times is married to the person that shook hands once. Continuing this argument, we see the following pairs: $(8, 0)$, $(7, 1)$, $(6, 2)$, $(5, 3)$. This leaves the person that shook hands 4 times without a partner, meaning that this person's partner did not give a number, hence this person must be Mrs. Landau! Conclusion: Mrs. Landau shook hands four times. A graph of the situation appears in figure 10.6.

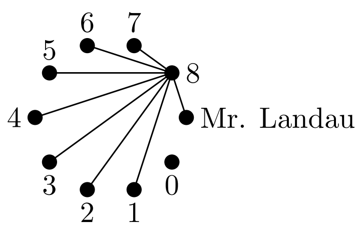


Figure 10.4: Example 10.54.

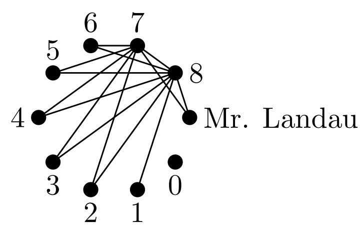


Figure 10.5: Example 10.54.

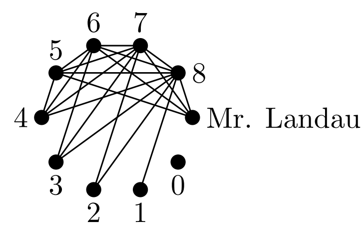


Figure 10.6: Example 10.54.

10.7 Traversability

Definition 10.55. *Recall that a* **trail** *is a walk where all the edges are distinct. An* **Eulerian trail** *on a graph G is a trail that traverses every edge of G. A* **tour** *of G is a closed walk that traverses each edge of G at least once. An* **Euler tour** *(or* **Euler cycle***) on G is a tour traversing each edge of G exactly once, that is, a closed Euler trail. A graph is* **Eulerian** *if it contains an Euler tour.*

Theorem 10.56. *A nonempty connected graph is Eulerian if and only if it has no vertices of odd degree.*

Proof: *Assume first that G is Eulerian, and let C be an Euler tour of G starting and ending at vertex u. Each time a vertex v is encountered along C, two of the edges incident to v are accounted for. Since C contains every edge of G, $d(v)$ is then even for all $v \neq u$. Also, since C begins and ends in u, $d(u)$ must also be even.*

Conversely, assume that G is a connected nonEulerian graph with at least one edge and no vertices of odd degree. Let W be the longest walk in G that traverses every edge at most once:

$$W = v_0, v_0v_1, v_1, v_1v_2, v_2, \ldots, v_{n-1}, v_{n-1}v_n, v_n.$$

Then W must traverse every edge incident to v_n, otherwise, W could be extended into a longer walk. In particular, W traverses two of these edges each time it passes through v_n and traverses $v_{n-1}v_n$ at the end of the walk. This accounts for an odd number of edges, but the degree of v_n is even by assumption. Hence, W must also begin at v_n, that is, $v_0 = v_n$. If W were not an Euler tour, we could find an edge not in W but incident to some vertex in W since G is connected. Call this edge uv_i. But then we can construct a longer walk:

$$u, uv_i, v_i, v_iv_{i+1}, \ldots, v_{n-1}v_n, v_n, v_0v_1, \ldots, v_{i-1}v_i, v_i.$$

This contradicts the definition of W, so W must be an Euler tour. $\square$

The following problem is perhaps the originator of graph theory.

Example 10.57 (Königsberg Bridge Problem)**.** The town of Königsberg (now called Kaliningrad) was built on an island in the Pregel River. The island sat near where two branches of the river join, and the borders of the town spread over to the banks of the river as well as a nearby promontory. Between these four land masses, seven bridges had been erected. The townsfolk used to amuse themselves by crossing over the bridges and asked whether it was possible to find a trail starting and ending in the same location allowing one to traverse each of the bridges exactly once. Figure 10.7 has a graph-theoretic model of the town, with the seven edges of the graph representing the seven bridges. By Theorem 10.56, this graph is not Eulerian so it is impossible to find a trail as the townsfolk asked.

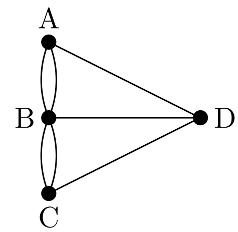


Figure 10.7: Model of the bridges in Königsberg from Example 10.57.

Definition 10.58. *A* **Hamiltonian cycle** *in a graph is a cycle passing through every vertex. G is* **Hamiltonian** *if it contains a Hamiltonian cycle.*

Unlike Theorem 10.56, there is no simple characterisation of all graphs with a Hamiltonian cycle. We have the following one-way result, however.

Theorem 10.59 (Dirac's Theorem, 1952)**.** *Let $G = (V, E)$ be a graph with $n = |V| \geq 3$ vertices where each vertex has degree $\geq \frac{n}{2}$. Then G is Hamiltonian.*

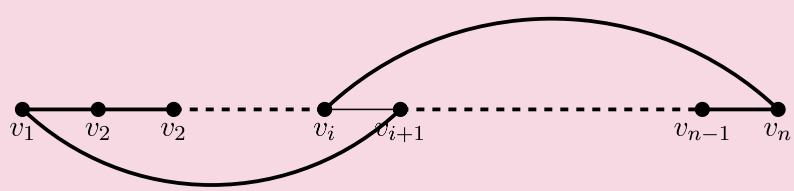
Proof: *Arguing by contradiction, suppose G is a maximal non-Hamiltonian graph with $n \geq 3$, and that G has more than 3 vertices. Then G cannot be complete. Let a and b be two non-adjacent vertices of G. By definition of G, $G + ab$ is Hamiltonian, and each of its Hamiltonian cycles must contain the edge ab. Hence, there is a Hamiltonian path $v_1v_2 \ldots v_n$ in G beginning at $v_1 = a$ and ending at $v_n = b$. Put*

$$S = \{v_i : av_{i+1} \in E\} \qquad \text{and} \qquad \{v_j : v_jb \in E\}.$$

As $v_n \in S \cap T$, we must have $|S \cup T| = n$. Moreover, $S \cap T = \varnothing$, since if $v_i \S \cap T$ then G would have the Hamiltonian cycle

$$v_1v_2 \cdots v_iv_nv_{n-1} \cdots v_{i+1}v_1,$$

as in the following figure, contrary to the assumption that G is non-Hamiltonian.



But then

$$d(a) + d(b) = |S| + |T| = |S \cup T| + |S \cap T| < n.$$

But since we are assuming that $d(a) \geq \dfrac{n}{2}$ and $d(b) \geq \dfrac{n}{2}$, we have arrived at a contradiction. □

10.8 Planarity

Definition 10.60. *A graph is* **planar** *if it can be drawn in a plane with no intersecting edges. Such a drawing is called a* **planar embedding** *of the graph.*

Example 10.61. K_4 is planar, as shown in figure 10.8.

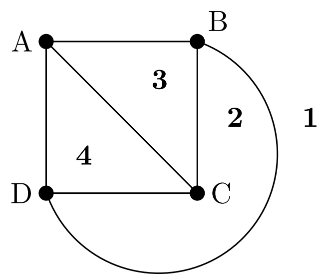


Figure 10.8: A planar embedding of K_4.

Definition 10.62. *A* **face** *of a planar graph is a region bounded by the edges of the graph.*

Example 10.63. From figure 10.8, K_4 has 4 faces. Face **1** which extends indefinitely, is called the *outside face*.

Theorem 10.64 (Euler's Formula)**.** *For every drawing of a connected planar graph with v vertices, e edges, and f faces the following formula holds:*

$$v - e + f = 2.$$

Proof: *The proof is by induction on e. Let $P(e)$ be the proposition that $v - e + f = 2$ for every drawing of a graph G with e edges. If $e = 0$ and it is connected, then we must have $v = 1$ and hence $f = 1$, since there is only the outside face. Therefore, $v - e + f = 1 - 0 + 1 = 2$, establishing $P(0)$.*

Assume now $P(e)$ is true, and consider a connected graph G with $e + 1$ edges. Either

❶ *G has no cycles. Then there is only the outside face, and so $f = 1$. Since there are $e+1$ edges and G is connected, we must have $v = e+2$. This gives $(e+2) - (e+1) + 1 = 2 - 1 + 1 = 2$, establishing $P(e+1)$.*

❷ *or G has at least one cycle. Consider a spanning tree of G and an edge uv in the cycle, but not in the tree. Such an edge is guaranteed by the fact that a tree has no cycles. Deleting uv merges the two faces on either side of the edge and leaves a graph G' with only e edges, v vertices, and f faces. G'*

is connected since there is a path between every pair of vertices within the spanning tree. So $v - e + f = 2$ by the induction assumption $P(e)$. But then

$$v - e + f = 2 \implies (v) - (e+1) + (f+1) = 2 \implies v - e + f = 2,$$

establishing $P(e+1)$.

This finishes the proof. ☐

Theorem 10.65. *Every simple planar graph with $v \geq 3$ vertices has $e \leq 3v - 6$ edges. Every simple planar graph with $v \geq 3$ vertices and which does not have C_3 as a subgraph has $e \leq 2v - 4$ edges.*

Proof: *If $v = 3$, both statements are plainly true so assume that G is a maximal planar graph with $v \geq 4$. We may also assume that G is connected, otherwise, we may add an edge to G. Since G is simple, every face has at least 3 edges in its boundary. If there are f faces, let F_k denote the number of edges on the k-th face, for $1 \leq k \leq f$. We then have*

$$F_1 + F_2 \cdots + F_f \geq 3f.$$

Also, every edge lies in the boundary of at most two faces. Hence if E_j denotes the number of faces that the j-th edge has, then

$$2e \geq E_1 + E_2 + \cdots + E_e.$$

Since $E_1 + E_2 + \cdots + E_e = F_1 + F_2 \cdots + F_f$, we deduce that $2e \geq 3f$. By Euler's Formula we then have $e \leq 3v - 6$.

The second statement follows for $v = 4$ by inspecting all graphs G with $v = 4$. Assume then that $v \geq 5$ and that G has no cycle of length 3. Then each face has at least four edges on its boundary. This gives $2e \geq 4f$ and by Euler's Formula, $e \leq 2v - 4$. ☐

Example 10.66. K_5 is not planar by Theorem 10.65 since K_5 has $\binom{5}{2} = 10$ edges and $10 > 9 = 3(5) - 6$.

Example 10.67. $K_{3,3}$ is not planar by Theorem 10.65 since $K_{3,3}$ has $3 \cdot 3 = 9$ edges and $9 > 8 = 2(6) - 4$.

10.9 Problems

Problem 10.1. Give the degrees of the vertices of each of the following graphs. Assume m and n are positive integers. For instance, for P_n, $n-1$ of the vertices have degree 2, and 2 vertices have degree 1.

(a) C_n

(b) Q_n

(c) K_n

(d) $K_{m,n}$

Problem 10.2. Can a graph with 6 vertices have vertices with the following degrees: $3, 4, 1, 5, 4, 2$? If so, draw it. If not, prove it.

Problem 10.3. Prove or disprove that Q_n is bipartite for $n \geq 1$.

Problem 10.4. For what values of n is K_n bipartite?

Problem 10.5. Give the adjacency matrix representation of Q_3, numbering the vertices in the obvious order.

Problem 10.6. Give the adjacency matrix for K_4.

Problem 10.7. Describe what the adjacency matrix looks like for K_n for $n > 1$.

Problem 10.8. What property does the adjacency matrix of every undirected graph have that is not necessarily true of directed graphs?

Problem 10.9. Let G be a graph and let u and v be vertices of G.

(a) If G is *undirected* and there is a path from u to v, is their necessarily a path from v to u? Explain, giving an example if possible.

(b) If G is *directed* and there is a path from u to v, is their necessarily a path from v to u? Explain, giving an example if possible.

Problem 10.10. For what values of n is Q_n Eulerian? Prove your claim.

Problem 10.11. Is C_n Eulerian for all $n \geq 3$? Prove it or give a counter example.

Problem 10.12. Prove that K_n is Hamiltonian for all $n \geq 3$.

Problem 10.13. Prove that $K_{n,n}$ is Hamiltonian for all $n \geq 3$.

Problem 10.14. For what values of m and n is $K_{m,n}$ Eulerian?

Problem 10.15. A graph is Eulerian if and only if its adjacency matrix has what property?

Problem 10.16. What properties does an adjacency matrix for graph G need in order to use Theorem 10.59 to prove it is Hamiltonian?

Problem 10.17. Let G be a bipartite graph with v vertices and e edges. Prove that if $v > 2v-4$, then G is not planar.

Problem 10.18. For each of the following, either give a planar embedding or prove the graph is not planar.

(a) Q_3

(b) Q_4 (Hint: Q_4 does not contain C_3 as a subgraph.)

(c) $K_{2,3}$

(d) K_5

Problem 10.19. If a graph has very few edges, which is better: an adjacency matrix or an adjacency list? Explain.

Problem 10.20. Let G be a graph with n vertices and m edges. and let u and v be arbitrary vertices of G. Describe an algorithm that accomplishes each of the following assuming G is represented using an *adjacency matrix*. Then give a tight bound on the worst-case complexity of the algorithm. Your bounds might be based on n, m, $deg(u)$, and/or $deg(v)$.

(a) Determine the degree of u.

(b) Determine whether or not edge (u, v) is in the graph.

(c) Iterate over the neighbors of u (and doing something for each neighbor, but don't worry about what).

(d) Add an edge between u and v.

Problem 10.21. Repeat Problem 10.20, but this time assume that G is represented using *adjacency lists*.

(a) Determine the degree of u.

(b) Determine whether or not edge (u, v) is in the graph.

(c) Iterate over the neighbors of u (and doing something for each neighbor, but don't worry about what).

(d) Add an edge between u and v.

Problem 10.22. (a) List several advantages that the adjacency matrix representation has over the adjacency list representation.

(b) List several advantages that the adjacency list representation has over the adjacency matrix representation.

Chapter 11

Selected Solutions

2.4 $2d+1$; $c+d+1$; even

2.6 $2n$; $2o+1$; some integers n and o; $4no+2n = 2(2no+n)$ or $2(n(2o+1))$. [Your steps might vary slightly, but you should end up with either $2(2no+n)$ or $2(n(2o+1))$ in the final step]; $2no+1$ or $n(2o+1)$; 'an even integer' or 'even'.

2.7 Let a and b be even integers. Then $a = 2m$ and $b = 2n$ for some integers m and n. Their product is $ab = (2m)(2n) = 2(2mn)$ which is even since $2mn$ is an integer.

2.8 Here are my comments on the proof.

- The first sentence is phrased weird–we are not letting a and b be odd *by* the definition of odd. We are *using* the definition.
- It does not state that n and q need to be integers.
- Although it is not incorrect, using n and q is just weird in this context. It is customary to use adjacent letters, like n and m, or q and r.
- Given the above problems, I would rephrase the first sentence as *'Let a and b be an odd numbers. Then $a = 2n+1$ and $b = 2m+1$ for some integers n and m.'*
- There is an algebra mistake. The product should be $2(2nq+q+n)$.
- If you replace $2nq+1$ with $2nq+q+n$ (twice) in the last sentence (see the previous item) it would be a perfect finish to the proof.

2.9 Hopefully it is clear to you that the proof *can't* be correct since the sum of an even and an odd number is odd, not even. The algebra is correct. *The problem is that $n+m+1/2$ is not an integer.* In order to be even, a number must be expressed in the form $2k$ ***where k is an integer***. *Any* number can be written as $2x$ if we don't require that x be an integer, so you *cannot* say that a number is even because it is of the form $2x$ unless x is an integer.

2.13 a an integer; $(3x+2)$; $(5x-7)$;7; 7 divides $15x^2-11x-14$.

2.15 This proof is correct. Not all of the Evaluate problems have an error!

2.17 The number 2 is positive and even but is clearly not composite since it is prime. Since the statement is false the proof must be incorrect. So where is the error? It is in the final statement. Although a can be written as the product of 2 and k, what if $k=1$ (that is, $a=2$). In that case we have not demonstrated that a has a factor other than a or 1, so we can't be sure that it is composite.

2.18 Let $a > 2$ be an even integer. Then $a = 2k$ for some integer k. Since $a \neq 2$, a has a factor other than a or 1. Therefore a is not prime. Therefore 2 is the only even prime number.

2.19 It was O.K. because according to the definition of prime, only positive integers can be prime. Therefore we only needed to consider positive even integers.

2.23 This one has a combination of two subtle errors. First of all, if $a|c$ and $b|c$, that does not necessarily imply that $ab|c$. For instance, $6|12$ and $4|12$, but it should be clear that $6 \cdot 4 \nmid 12$. Second, what if $a = b$? We'll see how to fix the proof in the next example.

2.25 Since n is not a perfect square, we know that $a \neq b$. Therefore $a < b$ or $b < a$. Since a and b are just labels for two factors of n, it doesn't matter which one is larger. So we can just assume a is the smaller one without any loss of generality. By definition of composite, we know that $a > 1$. Finally, it should be pretty clear that $b < n - 1$ since if $b = n - 1$, then $n = ab = a(n-1) \geq 2(n-1) = 2n - 2 = n + (n-2) > n$ since $n > 4$. But clearly $n > n$ is impossible.

2.26 We assumed that $n = a^2 > 4$, so clearly $a > 2$.

2.28

1. **Experiment**. If you aren't sure what to do, don't be afraid to try things.

2. **Read Examples**. But don't just read. Make sure you *understand* them.

3. **Practice**. It makes perfect!

2.33 Only when you read *xkcd* and you don't laugh.

2.34 If you build it and they don't come, the proposition is false. This is the only case where it is false. To see this, notice that if you build it and they do come, it is true. If you don't build it, then it doesn't matter whether or not they come–it is true.

2.38 If you don't know a programming language, then you don't know Java.

2.40 true; $\neg p$; false; p; p is true; q is false (the last two can be in either order).

2.42 If you don't know Java, then you don't know a programming language.

2.43 They are *not* equivalent. Since Java is a programming language, the proposition seems obviously true. However, what if someone knows C++ but not Java? Then they know a programming language but they don't know Java. Thus, the inverse is false. Since one is true and the other is false, the proposition and its inverse are clearly not equivalent.

2.45 If you know a programming language, then you know Java.

2.46 They are *not* equivalent. Since Java is a programming language, the proposition seems obviously true. However, what if someone knows C++ but not Java? Then they know a programming language but they don't know Java. Thus, the converse is false. Since one is true and the other is false, the proposition and its converse are clearly not equivalent.

2.48 (a) The implication states that if I get to watch "The Army of Darkness" that I will be happy. However, it doesn't say that it is the only thing that will make me happy. For instance, if I get to see "Iron Man" instead, that would also make me happy. Thus, the inverse statement is false.
(b) I will use fact that $p \to q$ is true unless p is true and q is false. The implication is true unless I watch "The Army of Darkness" and I am not happy. The contrapositive is "If I am not happy, then I didn't get to watch 'The Army of Darkness.' " This is true unless I am not happy and I watched "The Army of Darkness." Since this is exactly the same cases in which the implication are true, the implication and its contrapositive are equivalent.

2.51 $\sqrt{35}$; $10\sqrt{35}$; $3481 \geq 3500$; *nonsense* or *false* or *a contradiction.*

2.52

Evaluation of Proof 1: Here are my comments on this proof:

- It is proving the wrong thing. This proves that the product of an even number and an odd number is even. But it doesn't even do that quite correctly as we will see next.
- The first sentence is phrased weird–we are not letting a be even *by* the definition of even. We are *using* the definition.
- It does not state that n and q need to be integers.
- Although it is not incorrect, using n and q is just weird. It is customary to use adjacent letters, like n and m, or q and r.
- Given the above problems, I would rephrase the first sentence as *'Let a be an even number and b be an odd number. Then $a = 2n$ and $b = 2m + 1$ for some integers n and m.'*
- There is an algebra mistake. The product should be $2(2nq + n)$.
- The last sentence is actually perfect (again, except for the fact that it isn't proving the right thing).

Evaluation of Proof 2: This proof is incorrect. It actually proves the *converse* of the statement. (We'll learn more about converse statements later.) In other words, it proves that if at least of one of a or b is even, then ab is even. This is *not* the same thing. It is a pretty good proof of the wrong thing, but it can be improved in at least 4 ways.

- It defines a and b but never really uses them. They should be used at the beginning of the algebra steps (i.e. $a \cdot b = \cdots$) to make it clear that the algebra is related to the product of these two numbers.
- It needs to state that k and x are integers.
- As above, using k and x is weird (but not wrong). It would be better to use k and l, or x and y.
- It needs a few words to bring the steps together. In particular, sentences should not generally begin with algebra.

Taking into account these things, the second part could be rephrased as follows.

> Let $a = 2n$ and $b = 2m+1$, where n and m are integers. Then $ab = (2n)(2m+1) = 4nm + 2n = 2(2nm + n)$, which is even since $2nm + n$ is an integer.

Evaluation of Proof 3: This proof is correct.

2.56 $(1,2,3)$, $(1,3,2)$, $(2,1,3)$, $(2,3,1)$, $(3,1,2)$, and $(3,2,1)$.
2.58 Since it wasn't obvious how to do a direct proof of the fact, proof by contradiction seemed like the way to go. So we begin by assuming what we want to prove (that the product is even) is false. The short answer: *Because contradiction proofs generally begin by assuming the negation of what you want to prove.*
2.59 The proof gives the justification for this, but you may have to think about it for it to entirely sink in. Consider carefully the definition of S: $S = (a_1 - 1) + (a_2 - 2) + \cdots + (a_n - n)$. Notice it adds and subtracts terms. If $S = 0$, then the amount added and subtracted must be the same. And if you think about it for a few minutes, especially in light of the justification given in the proof, you should see why. If you can't see it right away, go back to how the a_k's are defined and think a little more. If you get totally stuck, try an example with $n = 3$ or 4.

2.62 Because $a^2 = a \cdot a$, so to list the factors of a^2 you can list the factors of a twice. Thus, a^2 has twice as many factors as a, so it must be an even number.

2.65 (1) No. (2) Yes. (3) No. (4) No. (5) Statements of the form "p implies q" are false precisely when p is true and q is false. (6) No. Whether or not you are 21, you aren't breaking the rule. (7) No. If p is false, whether or not q is true or false doesn't matter–the statement is true. Let's consider the previous question–if you do not drink alcohol, you are following the rule regardless of whether or not the statement "you are 21" is true or false.

2.66 $a > b$; $\frac{a-b}{2}$; $\frac{a-b}{2}$; $b + \frac{a}{2} - \frac{b}{2} = \frac{a}{2} + \frac{b}{2}$; subtract $\frac{a}{2}$ from both sides and multiple both sides by 2; $a > b$; contradiction; $a \leq b$.

2.68 $a\left(\frac{p}{q}\right)^2 + b\left(\frac{p}{q}\right) + c$; multiple both sides by q^2; odd; 0; $ap^2 + bpq$ is even and cq^2 is odd, so $ap^2 + bpq + cq^2$ is odd; $bpq + cq^2$ is even and ap^2 is odd, so $ap^2 + bpq + cq^2$ is odd; $ax^2 + bx + c = 0$ does not have a rational solution if a, b, and c are odd.

2.72

Evaluation of Proof 1: This is attempting to prove the *converse*, not the contrapositive. Since the converse of a statement is not equivalent to the original statement, this is not a valid proof. Further, the proof contains an algebra mistake. Finally, it uses the property that the sum of two even integers is even. Although this is true, the problem specifically asked to prove it using the definition of even/odd.

Evaluation of Proof 2: This proof starts out correctly by using the contrapositive statement and the definition of odd. Unfortunately, the writer claims that $5(\frac{6}{5}k + 1)$ is 'clearly odd.' This is not at all clear. What about this number makes it odd? Is it expressed as $2a + 1$ for some integer a? No. Even worse, there is a fraction in it, obscuring the fact that the number is even an integer.

Evaluation of Proof 3: This proof is *really close*. The only problem is that we don't know that $6k + 5$ is odd *using the definition of odd*. All the writer needed to do is take their algebra a little further to obtain $2(3k + 2) + 1$, which is odd by the definition of odd since $3k + 2$ is an integer.

2.78 Answers will vary greatly, but one proof is: 3 and 5 are prime but $3 + 5 = 8 = 2^3$ is clearly not prime.

2.81 $2s$ is a power of two that is in the closed interval.; $2^r = 2 \cdot 2^{r-1} < 2s < 2 \cdot 2^r = 2^{r+1}$, so $s < 2^r < 2s < 2^{r+1}$, and so the interval $[s, 2s]$ contains 2^r, a power of 2.

2.82 Because these statements are contrapositives of each other. In other words, they are equivalent. Therefore you can prove either form of the statement.

2.84 If x is odd, then $x = 2k+1$ for some integer k. Then $x+20 = 2k+1+20 = 2(k+10)+1$, which is odd since $k + 10$ is an integer. If $x + 20$ is odd, then $x + 20 = 2k + 1$ for some integer k. Then $x = (x+20) - 20 = 2k+1-20 = 2(k-10)+1$, which is odd since $k-10$ is odd. Therefore x is odd iff $x + 20$ is odd.

2.85 If x is odd, then $x = 2k+1$ for some integer k. Then $x+20 = 2k+1+20 = 2(k+10)+1$, which is odd since $k + 10$ is an integer. If x is even, then $x = 2k$ for some integer k. Then $x + 20 = 2k + 20 = 2(k + 10)$. Since $k + 10$ is an integer, then $x + 20$ is even. Therefore x is odd iff $x + 20$ is odd.

2.86 p implies q; q implies p; p implies q; $\neg p$ implies $\neg q$

2.87

Evaluation of Proof 1: For the forward direction, they didn't use the definition of odd. Otherwise, that part is fine. For the backward direction, their proof is nonsense. They *assumed* that $x = 2k + 1$ when they wrote $(2k + 1) - 4$ in the second sentence. This need to be *proven.*

Evaluation of Proof 2: For the forward direction, they didn't specify that k was an integer. Otherwise it is correct. The second part of the proof is *not* proving the converse. It is proving the forward direction a second time using a proof by contraposition. In other words, this proof just proves the forward direction twice and does not prove the backward direction.

2.89 The problem is that this is actually a proof that $x+x$ is even if x is even since $x = 2a = y$ was assumed.

2.90 Notice that 4 and 6 are even, but $4+6 = 10$ is not divisible by 4. So clearly the statement is incorrect. Therefore, there must be something wrong with the proof. The problem is the same as in the previous example–the proof assumed $x = y$, even if that was not the intent of the writer. So what was proven was that if x is even, then $x + x$ is divisible by 4.

2.91 Since it should be clear that the result $(-1 = 1)$ is false, the proof can't possibly be correct.

2.92 No! Example 2.91 should have made it clear that this approach is flawed.

2.93 No, you should not be convinced. As we just mentioned, whether or not the equation is true, sometimes you can work both sides to get the same thing. Thus the technique of working both sides is not valid. It doesn't guarantee anything.

2.94 Since p and q are odd, we know that $p + q$ is even, and so $\dfrac{p+q}{2}$ is an integer. But $p < q$ gives $2p < p + q < 2q$ and so $p < \dfrac{p+q}{2} < q$, that is, the average of p and q lies between them. Since p and q are consecutive primes, any number between them is composite, and so divisible by at least two primes. So $p + q = 2\left(\dfrac{p+q}{2}\right)$ is divisible by the prime 2 and by at least two other primes dividing $\dfrac{p+q}{2}$.

2.95

Evaluation of Proof 1: This is not correct. It needs to be shown that x^y can be written as c/d, where c and d are integers with $d \neq 0$. Ask yourself this: Are a^y and b^y necessarily integers?

Evaluation of Proof 2: This is not correct. If $y = 3/2$, what does it mean to multiple x by itself one and a half times?

2.96 The statement is false. There are many counterexamples, but here is an easy one: Let $x = 2$ and $y = 1/2$. Then $x^y = 2^{1/2} = \sqrt{2}$, which is irrational.

2.97

Evaluation of Proof 1: This solution has two serious flaws. First, we absolutely cannot assume x is an integer. The only thing we can assume about x is that it is rational, and not every rational number is an integer. The other problem is that the writer proved the *inverse*, not the *contrapositive*. What they needed to prove was that if $1/x$ is rational, then x is rational. So in actuality, we know is that $1/x$ is rational, not x. We need to prove that x is rational based on the assumption that $1/x$ is rational.

Evaluation of Proof 2: This is not really a proof. It just takes the statement of the problem one step further. Is the writer *sure* that $1/x$ can't be expressed as an integer over an integer? Why? There are just too many details omitted.

Evaluation of Proof 3: The biggest flaw is that this is a proof of the *inverse* statement, not the *contrapositive*. So even it the rest of the proof were correct, it would be proving the wrong thing since the *inverse* and *contrapositive* are not equivalent. But the rest is not even entirely correct because the inverse statement is not quite true. If $x = 0$, then $p = 0$ as well and the statement and proof falls apart for the same reason–you can't divide by 0.

Evaluation of Proof 4: This proof is *almost correct*. It does correctly try to prove the contrapositive, and if it had done so correctly, that would imply the original statement is true. But there is one small problem: If $a = 0$ the proof would fall apart because it would divide by 0. This possibility needs to be dealt with. This is actually not too difficult to fix. We just need to add the following sentence before the last sentence: *"Since $0 \neq 1/x$ for any value of x, we know that $a \neq 0$."*.

Evaluation of Proof 5: This proof is correct.

2.98

Evaluation of Proof 1: As you will prove next, the statement is actually false. Therefore the proof has to be incorrect. But where did it go wrong? It turns out they they tried to prove the wrong thing. What needed to be proved was "If p is prime then $2^p - 1$ is prime." They attempted to prove the *converse* statement, which is not equivalent. We can still learn something by evaluating their proof. It turns out that the converse is actually true, and the proof has a lot of correct elements. Unfortunately, they are not put together properly. First of all, the proof seems to be a combination of a contradiction proof and a proof by contrapositive. They needed to pick one and stick with it. Second, the arrows ($\rightarrow$) are confusing. What do they mean? I think they are supposed to be read as "implies", but a few more words are needed to make the connections between these phrases. Finally, the final statement is incorrect. This does *not* prove that all numbers of the form $2^p - 1$ are prime when p is prime.

Evaluation of Proof 2: This proof is not even close. This is a case of "I wasn't sure how to prove it so I just said stuff that sounded good." You can't argue anything about the factors of $2^p - 1$ based on the factors of 2^p. Further, although $2^p - 1$ being odd means 2 is not a factor, it doesn't tell us whether or not the number might have *other* factors.

2.99 Notice that 11 is prime but that $2^{11} - 1 = 23 \cdot 89$ is not. Therefore, not all numbers of the form $2^p - 1$, where p is prime, are prime.

3.2

```
double areaSquare(double w) { return w*w; }
```

3.7 It does not work. To see why, notice that if we pass in a and b, then $x = a$ and $y = b$ at the beginning. After the first line, $x = b$ and $y = b$. After the second line $x = b$ and $y = b$. The problem is that the first line overwrites the value stored in x (a), and we can't recover it.

3.12 (a) 45; (b) 8; (c) 3; (d) 6; (e) 0; (f) 7; (g) 7; (h) 7; (i) 11.

3.21 -15; -7; 9; 13; 21. Notice that it is every 4th number along the number line, both in the positive and negative directions.

3.22 Either -1 or 3 are possible answers if we are uncertain whether it will return a positive or negative answer. But we know it is one of these. It won't be -5, for instance.

3.23

Evaluation of Solution 1: This solution is both incorrect and a bit confusing. The phrase 'both sides' is confusing–both sides of what? We don't have an equation in this problem. But there is a more serious problem. If you thought it was correct, go back and try to figure out why it is incorrect before you continue reading this solution. The main problem is that although this may return a value in the correct range, it doesn't always return the correct value. In fact, what if $(a \pmod b + b - 1)$ is odd? Mathematically, this would result in a non-integer result which is clearly incorrect. In most programming languages it would at least truncate and return an integer–but again, not always the correct one. This person focused on the wrong thing–getting the number in a particular range. Although that is important, they needed to think more about how to get the *correct* number. They should have plugged in a few more values to double-check their logic.

Evaluation of Solution 2: Incorrect. Generally speaking, $a \bmod b \neq -a \bmod b$. In other words, returning the absolute value when the result is negative is almost always incorrect.

Evaluation of Solution 3: Incorrect. If you think about if for a few minutes you should see that this is just one way of implementing the idea from the previous solution.

Evaluation of Solution 4: Incorrect. If $(a \bmod b)$ is negative, performing another mod will still leave it negative.

3.24 There are several possible answers, but the slickest is probably: `(b+(a mod b)) mod b`. Try it with both positive and negative numbers for a and convince yourself that it is correct.

3.27 (1) 9; (2) 10; (3) 9; (4) 10; (5) 9; (6) 9.

3.29

Evaluation of Solution 1: This does not work. What happens when $x = 3.508$, for instance?

Evaluation of Solution 2: This is incorrect for two reasons. First, $1/2 = 0$ in most programming languages, so this will *always* round down. Second, even if we replaced this with $.5$ or $1.0/2.0$, it would round *up* at $.5$.

Evaluation of Solution 3: Nice try, but still no good. What if $x = 2.5$? This will round *up* to 3. Worse, what if $x = 2.0001$? Again, it rounds *up* to 3 which is really bad.

Evaluation of Solution 4: This one is correct. Plug in values like 2, 2.1, and 2.5 to see that it rounds down to 2 and values like 2.51, 2.7, and 2.9 to see that it rounds up to 3.

3.31 (a) 0; (b) 1; (c) 1; (d) 1; (e) 1 (f) 1; (g) 2; (h) 9; (i) 0; (j) -1; (k) -1; (l) -2.

3.32

Evaluation of Solution 1: 0.5 is not an integer, and the `floor` function is not allowed.

Evaluation of Solution 2: The `floor` function is not allowed. Even if it were, this solution doesn't work. $1/2$ is evaluated to 0 so it doesn't help.

Evaluation of Solution 3: This one works, but 0.5 is not allowed so it does not follow the directions.

3.33 Two reasonable solutions include `(n+m/2)/m` and `(2n+m)/(2m)`.

3.36 Here is one possibility:

```
int max(int x, int y, int z) {
    int w = max(x,y);
    return max(w,z);
}
```

We will use a proof by cases.

- If x is the maximum, then $w =$`max`$(x, y) = x$. so it returns `max`$(w, z) = x$, which is correct.
- If y is the maximum, the argument is essentially the same the previous case.
- If z is the maximum, then w is either x or y, but in either case $w \leq z$, so it returns `max`$(w, z) = z$.

In each case the algorithm returns the correct answer.

3.37 Here is a possible answer.

```
void HelloGoodbye(int x) {
    if(x >= 4) {
        if(x <= 6) {
            print("Hello");
        } else {
            print("Goodbye");
        }
    } else {
        print("Goodbye");
    }
}
```

3.38 It is possible. if you thought it wasn't, go back and try to write the algorithm before reading any further.

Here is one way to do it using an extra variable and an additional conditional statement.

```
void HelloGoodbye(int x) {
    boolean sayGoodbye = true;
    if(x >= 4) {
        if(x <= 6) {
            sayGoodbye = false;
        }
    }
    if(sayGoodbye) {
        print("Goodbye");
    } else {
        print("Hello");
    }
}
```

It can also be done by using a return statement:

```
void HelloGoodbye(int x) {
    if(x >= 4) {
        if(x <= 6) {
            print("Hello");
            return;
        }
    }
    print("Goodbye");
}
```

The second solution is simpler, but this sort of code (with somewhat random `return` statements in the middle of them) can be tricky to debug if it is changed later. Did you come up with a better solution than these?

3.41 It will loop forever if $n < 0$. This should be fixed in two ways. Since $n!$ is undefined when $n < 0$, it can't return the correct answer for negative values. So the first change is that instead of looping forever, it should check and return return -1 if $n < 0$. Some other value can be used, but -1 works well because $n!$ can't be negative, so if it returns -1, you know something is up. Second, this behavior should be clearly documented so that it clearly states that it returns $n!$ for $n \geq 0$, and -1 for $n < 0$.

3.42

Evaluation of Solution 1: This algorithm always returns 0. If you don't see it right away, carefully work through the algorithm with a few values of n.

Evaluation of Solution 2: This is correct. It doesn't multiply by 1, but that doesn't change the answer.

Evaluation of Solution 3: This is also correct. It is just multiplying the values in the reverse order of the other examples.

Evaluation of Solution 4: This is incorrect. It actually computes $(n-1)!$. To fix this, does i have to start at 0, or go to n (instead of $n-1$)? We'll leave it to you to work out which is correct.

3.43 This isn't really that much different than the algorithms to compute $n!$. Here is one algorithm that does the job:

```
double power(double x, int n) {
    power = 1;
    for(int i=0;i<n;i++) {
        power = power*x;
    }
    return power;
}
```

The loop could also have been `for(int i=1;i<=n;i++)`, or any loop that executes n times since the loop index, i, is not used as part of the calculation.

3.48 As we already discussed, many languages truncate when performing integer division. When the numbers are positive (as they are here), that is the same thing as taking the floor. Even if a language does not do this, Theorem 3.28 implies that it would still work.

3.49 It is easiest to see that this is correct by comparing with the previous solution. The only difference is that the condition went from $i <= (n-2)/2$ to $i < n/2$. Notice that $(n-2)/2+1 = n/2 - 2/2 + 1 = n/2$. But since we also replaced $<=$ with $<$, it still stops at the same point.

3.51 It is correct. To convince yourself (but this is *not* a proof), plug the numbers 1 through 5 (or some other set of both even and odd values) into both sides to see that you get the same number.

3.57 In the first iteration in the loop, $n = 5, i = 1$, $n * i > 4$ and thus $x = 10$. Next, $n = 3$, $i = 2$, and we go through the loop again. Since $n * i > 4$, $x = 10 + 2 * 3 = 16$. Finally, $n = 1, i = 3$, and the loop stops. Hence $x = 16$ is returned.

3.60 $\lfloor\sqrt{101}\rfloor = 10$. The only primes less than 10 are 2, 3, 5, and 7. Since $101 \bmod 2 = 1$, $101 \bmod 3 = 2$, $101 \bmod 5 = 1$, and $101 \bmod 7 = 3$, none of which are 0, Theorem 3.58 tells us that 101 is prime.

3.61 $323 = 17 * 19$, so it is not prime. I determined this by seeing if any of the primes no greater than $\lfloor\sqrt{323}\rfloor = 17$ were factors. Although 2, 3, 5, 7, 11, and 13, are not, 17 is.
3.63 It allows the loop to increment by 2 each time instead of by one, making it about twice as fast. This is sensible since if 2 is not a factor, no even number is, so why check them all?

A bonus thought: If you think about it, the same thing could be said of 3, 5, etc. That is, once we know that a number is not divisible by 3, we don't really need to ask if it is divisible by 6, 9, 12, etc. But doing this in general (not just for 2) complicates the algorithm quite a bit. So we'll just settle for an algorithm that is about twice as fast.
3.64 The following algorithm does the job.

```
int reverseDigits(int n) {
        x=0;
        while(n!=0) {
            x = x*10+n%10;
            n=n/10;
        }
        return x;
}
```

4.3 (a) false. (b) true. (c) true. (d) false. If you don't know the story behind this, Google it.
4.5 (a) Not a proposition. (b) I would like to think this is true. However, this is not a proposition since not everyone agrees with me. (c) Also not a proposition. (d) true. (e) false. This one is a bit tricky to think about it, so the next example will ask you to prove it.
4.9 "I am not learning discrete mathematics." You could also have "It is not the case that I am learning discrete mathematics," although it is better to smooth out the English when possible.; False. Since you are currently reading the book, you *are* learning discrete mathematics.
4.10 (a) The statement "This is not a proposition" is just the negation of the proposition "This is a proposition," so it is a proposition. Since "This is a proposition" is true, "This is not a proposition" is false.
(b) Assume that "This is not a proposition" is not a proposition. Then the statement "This is not a proposition" is true, which means it is a proposition. But we assumed it wasn't a proposition. Since we have a contradiction, our assumption that "This is not a proposition" is not a proposition was incorrect, so it is a proposition. Since it is a proposition that says is isn't, its truth value is false.
4.11 `list.size()!=0` and `!(list.size()==0)` are the most obvious solutions.
4.15 Either "I like cake and I like ice cream," or "I like cake and ice cream" are correct.
4.19 "$x > 0$ or $x < 10$"; true; true; $x < 10$; true.
4.20 (a) "You do not have to be at least 48 inches tall to ride the roller coaster." It is *not* "You must be at most 48 inches tall to ride the roller coaster." (b) "You must be at least 48 inches tall or 18 years old to ride the roller coaster." (c) "You must be at least 48 inches tall and at least 18 years old to ride the roller coaster."
4.21 Here is one possible solution. Note that the parentheses are necessary.

```
boolean startsOrEndsWithZero(int[] a, int n) {
    if( n>0 && (a[0]==0 || a[n-1]==0) ) {
        return true;
    else {
        return false;
    }
}
```

4.22 The solution uses the expression `n>0 && (a[0]==0 || a[n-1]==0)`. If $n = 0$, the expression is false because of the `&&`, so the algorithm returns false as it should since an array with no elements certainly does not begin or end with a 0. If $n = 1$, first note that $n - 1 = 0$, so $a[0]$ and $a[n-1]$ refer to the same element. Although this is redundant, it isn't a problem. If $a[0] = 0$, the expression evaluates to $T \wedge (T \vee T) = T$, and the algorithm returns true as expected. If $a[0] \neq 0$, the expression evaluates to $T \wedge (F \vee F) = F$, and the algorithm returns false as expected.

4.25 (a) XOR; (b) OR. This one is a little tricky because parts can't be simultaneously true so it sounds like an XOR. But since the point of the statement is not to *prevent* both from being true, it is an OR. (c) Without more context, this one is difficult to answer. I would suspect that most of the time this is probably OR. The purpose of this example is to demonstrate that sometimes life contains ambiguities. This is particularly true with software specifications. Generally speaking, you should *not* just assume one of the alternative possibilities. Instead, get the ambiguity clarified. (d) When course prerequisites are involved, OR is almost certainly in mind. (e) The way this is phrased, it is almost certainly an XOR.

4.26 $p \vee q$ is "either list 1 or list 2 is empty." To be completely unambiguous, you could rephrase it as "at least one of list 1 or list 2 is empty." $p \oplus q$ is "either list 1 or list 2 is empty, but not both," or "precisely one of list 1 or list 2 is empty." They are different because if both lists are empty, $p \vee q$ is true, but $p \oplus q$ is false.

4.27 (a) No. If p and q are both true, then $p \vee q$ is true, but $p \oplus q$ is false, so they do not mean the same thing.
(b) We have to be very careful here. In general, the answer to this would be absolutely not (we'll discuss this more next). However, for *this particular p and q*, they actually essentially are the same. But the reason is that it is impossible for x to be less than 5 and greater than 15 at the same time. In other words, p and q can't both be true at the same time. The only other way for $p \oplus q$ to be false is if both p and q are false, which is exactly when $p \vee q$ is false.

4.30 (a) An A. (b) We can't be sure. We know that earning 90% is enough for an A, but we don't know whether or not there are other ways of earning an A. (c) We can't be sure. If the premise is false, we don't know anything about conclusion.

4.33 (a) An A. (b) Yes. Because it is a biconditional statement that we assumed to be true, the statements "you will receive an A in the course" and "you earn 90%" have the same truth value. Since the former is true, the latter has to be true. (c) Yes. Notice that $p \leftrightarrow q$ is equivalent to $\neg p \leftrightarrow \neg q$ (You should convince yourself that this is true). Thus the statements "you don't earn 90%" and "you didn't get an A" have the same truth value.

4.35 The answers are in **bold**.

With Variables/Operators	In English
$p \to q$	**If *Iron Man* is on TV, then I will watch it.**
$\boldsymbol{(\neg r \wedge p) \to q}$	If I don't own *Iron Man* on DVD and it is on TV, I will watch it.
$p \wedge r \wedge \neg q$	***Iron Man* is on TV and I own the DVD, but I won't watch it.**
$\boldsymbol{q \leftrightarrow p}$	I will watch *Iron Man* every time it is on TV, and that is the only time I watch it.
$\boldsymbol{r \to q}$	I will watch *Iron Man* if I own the DVD.

4.38 Here is the truth table with one (optional) intermediate column.

p	q	$p \to q$	$(p \to q) \wedge q$
T	T	T	T
T	F	F	F
F	T	T	T
F	F	T	F

4.40 Here is the truth table with two intermediate columns. For consistency, your table should have the rows in the same order.

a	b	c	$\neg b$	$a \vee \neg b$	$(a \vee \neg b) \wedge c)$
T	T	T	F	T	T
T	T	F	F	T	F
T	F	T	T	T	T
T	F	F	T	T	F
F	T	T	F	F	F
F	T	F	F	F	F
F	F	T	T	T	T
F	F	F	T	T	F

4.43 $(a \wedge b) \vee c$

4.44 They are not equivalent. For instance, when $a = F$, $b = F$, and $c = T$, $(a \wedge b) \vee c$ is true but $a \wedge (b \vee c)$ is false.

4.45 Since $(a \to b) \to c$ is how it should be interpreted, the first statement is correct. The second statement is incorrect. We'll leave it to you to find true values for a, b, and c that result in these two parenthesizations having different truth values.

4.48 (a) tautology (b) contradiction; p and $\neg p$ cannot both be true. (c) contingency; it can be either true or false depending on the truth values of p and q.

4.50

Evaluation of Proof 1: Nice truth table, but what does it *mean*? It is just a bunch of symbols on a page. Why does this truth table prove that the proposition is a tautology? The proof needs to include a sentence or two to make the connection between the truth table and the proposition being a tautology.

Evaluation of Proof 2: This is mostly correct, but the phrasing could be improved. For instance, the phrase 'they all return true' is problematic. Who/what are 'they'? And what does it mean that they 'return' true? Propositions don't 'return' anything. Replace 'Since they all return true' with 'Since every row of the table is true' and the proof would be good.

Evaluation of Proof 3: While I applaud the attempt at completeness, this proof is way too complicated. It is hard to understand because of the incredibly long sentences and the mathematical statements written in English in the middle of sentences. But I suppose that technically speaking it is correct. Here are a few specific examples of problems with the proof (not exhaustive). The first three sentences are confusing as stated. The point that the author is trying to make is that whenever q is true, the statement must be true regardless of the value of p, so there is nothing further to verify. Thus the only case left is when q is false. This point could be made with far few words and more clearly. The phrase 'we would have true and (true implies false), which is false,' is very confusing, as are a few similar statements in the proof. The problem is that the writer is trying to express mathematical statements in sentence form instead of using mathematical notation. There is a reason we learn mathematical notation–to use it!

Evaluation of Proof 4: This proof is correct and is not too difficult to understand. It is a lot better than the previous proof for a few reasons. First of all, it starts off in a better place–focusing in on the single case of importance. Second, it uses the appropriate mathematical notation and refers to definitions and previous facts to clarify the argument.

Evaluation of Proof 5: While I appreciate the patriotism (in case you don't know, some people use 'merica as a shorthand for America), this has nothing to do with the question. Sorry, no points for you! By the way, I did *not* make this solution up. Although it wasn't really used on this particular problem, one student was in the habit of giving answers like this if he didn't know how to do a problem.

4.54 Below is the truth table for $\neg(p \wedge q)$ and $\neg p \vee \neg q$ (the gray columns).

p	q	$p \wedge q$	$\neg(p \wedge q)$	$\neg p$	$\neg q$	$\neg p \vee \neg q$
T	T	T	F	F	F	F
T	F	F	T	F	T	T
F	T	F	T	T	F	T
F	F	F	T	T	T	T

Since they are the same for every row of the table, $\neg(p \wedge q) = \neg p \vee \neg q$.

4.55 For the first part: negation; distribution; $p \vee p$. For the second part:

$$\begin{aligned} p &= \underline{p \wedge T} && \text{(identity)} \\ &= \underline{p \wedge (p \vee \neg p)} && \text{(negation)} \\ &= \underline{(p \wedge p) \vee (p \wedge \neg p)} && \text{(distributive)} \\ &= \underline{(p \wedge p) \vee F} && \text{(negation)} \\ &= p \wedge p && \text{(\underline{identity})} \end{aligned}$$

Thus, $\underline{p \wedge p = p}$.

4.57 (a) We can use the identity, distributive, and dominations laws to see that

$$p \vee (p \wedge q) = (p \wedge T) \vee (p \wedge q) = p \wedge (T \vee q) = p \wedge T = p.$$

(b) We can prove this similarly to the previous one, or we can use the previous one along with distribution and idempotent laws:

$$p \wedge (p \vee q) = (p \wedge p) \vee (p \wedge q) = p \vee (p \wedge q) = p.$$

4.60 Let $p =$"$x > 0$" and $q =$"$x < y$". Then the conditional above can be expressed as $(p \wedge q) \vee (p \wedge \neg q)$. According to Example 4.56, this is just p. Therefore the code simplifies to:

```
if (x>0) {
    x=y;
}
```

4.61 This is not equivalent to the original code. Consider the case when $x = -1$ and $y = 1$, for instance.

4.62 Since both conditions need to be true when if statements are nested, it is the same thing as a conjunction. In other words, the two ifs are equivalent to `if( x>0 && (x<y || x>0) )`. By absorption, this is equivalent to `if(x>0)`. So the simplified code is:

```
if(x>0) {
    x=y;
}
```

You can also think about it this way. The assignment `x=y` cannot happen unless `x>0` due to the outer if.[1] But the inner if has a disjunction, one part of which is `x>0`, which we already know is true. In other words, it doesn't matter whether or not `x<y`. This argument also leads to the solution we just gave.

4.64

Evaluation of Solution 1: This solution is incorrect. There are a few problems. The obvious one is that the first statement actually prevents the program from crashing so it is certainly not unnecessary! Also, the second and third statements may be equivalent, but how are they connected? For instance, given the expression $\neg(A \wedge B) \vee \neg A$, I cannot simply remove the 'redundant' $\neg A$ to obtain an "equivalent" expression of $\neg(A \wedge B)$ (if necessary, plug in different truth values for A and B to convince yourself that these are not the same).

Evaluation of Solution 2: This is not correct. The second part of the expression seems to have disappeared. But how can we know it isn't equivalent? We just need to find a scenario where the two versions do different things. Notice that the 'simplified' expression is true when the list is not empty regardless of the value of element 0. But what if the list is not empty and element 0 is 50? The original expression is false and the 'simplified' expression is true. Clearly not the same.

Evaluation of Solution 3: This solution is not only correct, but it is very well argued.

4.65 Technically speaking, the final solution is not equivalent. However, it turns out that it is *better* than the original. This is because the original code would actually crash if the list is empty. Go back and look at the code and verify that this is the case. Then verify that the final simplified version will *not* crash.

4.66 (a) $p \oplus q$; (b) $(p \wedge \neg q) \vee (\neg p \wedge q)$ or $(p \vee q) \wedge \neg(p \wedge q)$. Other answers are possible, but most likely you came up with one of these. If not, construct a truth table to determine whether or not your answer is correct.

4.68

Evaluation of Proof 1: This is an incomplete proof. It only proves that in one case (p and q both being true) they are equivalent. It says nothing about, for instance, whether or not they have the same truth value when p is true and q is false.

Evaluation of Proof 2: This proof is also incomplete. It proves that in two cases they have the same truth value, but is silent about the other cases. Are we supposed to *assume* that in all other cases the expressions are both false?

Evaluation of Proof 3: This is either incomplete or incorrect, depending on how you read it. If by "precisely" the writer means "exactly when", then it is incorrect since the propositions are also true when both p and q are false. Otherwise the proof is incomplete because it does not deal with every case.

Evaluation of Proof 4: This is correct because it exhausts all of the cases. It is perhaps a bit brief, however. The only way I know the proof is actually correct is that I have to verify

[1]If you think about it, this is why the solution to this in the previous example failed.

what the writer said. By the definition of $p \leftrightarrow q$, what they said is clearly true. But to see that it is true of $(p \wedge q) \vee (\neg p \wedge \neg q)$ I have to actually plug in a few values and/or think about the meaning of the expression.

4.72 (a) is a predicate since it can be true or false depending on the value of x.; (b) is not a predicate since it is simply a false statement–it doesn't contain any variables.; (c) is a predicate since it can be true or false depending on the value of M.; (d) is not a predicate. This one is tricky. This is a definition. In this statement, x is not a variable but a label for a number so that it can be referred to later in the sentence.
4.76

(a) $\forall x(2x < 3x)$. In case it isn't obvious, there is nothing magical about x. You could also write your answer as $\forall a(2a < 3a)$, for instance.

(b) $\forall n(n! < n^n)$.

4.79 $\forall x \neq 0(x^2 \neq 0)$. Alternatively, $\forall x(x \neq 0 \rightarrow x^2 \neq 0)$.
4.82 $\exists x(x > 0)$.
4.84

Evaluation of Solution 1: While perhaps technically correct, this solution is not very good. It at least uses a quantifier. But the fact that it includes the phrase "is even" suggests that it could be phrased a bit more 'mathematically.'

Evaluation of Solution 2: This solution is pretty good. It is concise, but expresses the idea with mathematical precision. Although it doesn't directly appeal to the definition of even, it does use a fact that we all know to be true of even numbers.

Evaluation of Solution 3: This solution is also good. It clearly uses the definition of even. It is a bit more complicated since it uses two quantifiers, but I prefer this one slightly over the second solution. But that may be because I didn't come up with the second solution and I refuse to admit that someone had a better solution than what I thought of (which was this one).

4.86 $\forall x \exists y \exists z(x = y^2 + z^2)$.
4.87 y and z; non-negative integers; a perfect square; $z^2 = 2$; $z^2 \leq -1$; which is also impossible; exhausted/tried all possible values of y.
4.91 You may have a different answer, but here is one possibility based on the hint. If we let $P(x, y)$ be $x < y$ where the domain for both is the real numbers, then $\forall x \exists y(x < y)$ is true since for any given x, we can choose $y = x + 1$. However, $\exists y \forall x(x < y)$ is false since no matter what value we pick for y, $x < y$ is false for $x = y + 1$. In other words, it is not true for *all* values of x. As with the previous examples, the difference is that in this case we need to have a single value of y that works for *all* values of x.
4.95

(a) It is saying that every integer can be written as two times another integer. Simplified, it is saying that every integer is even.

(b) The most direct translation of the final line of the solution is "There is some integer that cannot be written as two times another integer for any integer." A smoothed-out translation would be "There is at least one odd integer."

(c) Since 3 is odd, the statement is clearly false.

4.98 Using De Morgan's Law, we get `!(P(i) || Q(i))`.

4.99 First notice that if $P(i)$ is true for every value of i, `result` will be true at the end of the first loop, so `isTrueForAll3` will return true without even considering Q. However, if $P(i)$ is false for any value of i, then it will go onto the second loop. The second loop will return false if $Q(i)$ is false for any value of i. But if $Q(i)$ is true for all values of i, the method returns true. So, how do we put this all together into a simple answer? Notice that the only time it returns true is if either $P(i)$ is always true or if $Q(i)$ is always true. In other words, `isTrueForAll3` is determining the truth value of $(\forall i P(i)) \vee (\forall i Q(i))$. By the way, notice that I used the variable `i` for both quantifiers. This is sort of like using the same variables for two separate loops.

4.108 The truth table for $p \leftrightarrow q$ is given to the right. The first row yields conjunctive clause $p \wedge q$, and the fourth row yields conjunctive clause $\neg p \wedge \neg q$. The disjunction of these is $(p \wedge q) \vee (\neg p \wedge \neg q)$.
Thus, $p \leftrightarrow q = (p \wedge q) \vee (\neg p \wedge \neg q)$.

p	q	$p \leftrightarrow q$
T	T	T
T	F	F
F	T	F
F	F	T

4.110 $Y = (\neg p \wedge q \wedge r) \vee (\neg p \wedge q \wedge \neg r) \vee (\neg p \wedge \neg q \wedge r) \vee (\neg p \wedge \neg q \wedge \neg r)$.

4.116 `11110000`; `11110000`; `00001111`; 15.

4.117 `00111001`.

4.120 `11000000`; `11111100`; `00111100`.

5.4 The prime numbers less than 10 are 2, 3, 5, and 7. But the problem asked for the *set of* prime numbers less than 10. Therefore, the answer is $\{2, 3, 5, 7\}$. If you were asked to *list* the prime numbers less than 10, an appropriate answer would have been $2, 3, 5, 7$ (but that is not what was asked).

5.7 6; 5; 6; A and C represent the same set.

5.10 ∞; ∞. You might think it is $\infty/2$, but you can't do arithmetic with ∞ since it isn't a number. Without getting too technical, although $\mathbb{Z}^+$ seems to have about half as many elements as $\mathbb{Z}$, it actually doesn't. It has the exact same number: ∞. ; 0.

5.13 $\{2a : a \in \mathbb{Z}\}$.

5.14 $\mathbb{Q} = \{a/b : a, b \in \mathbb{Z}, b \neq 0\}$.

5.17 (a) Yes. (b) Yes. A is a proper subset since 25, for instance, is in S but not in A. (c) Yes. Every set is a subset of itself. (d) No. No subset is a *proper subset* of itself. (e) No. $25 \in S$, but $25 \notin A$.

5.18 (a) yes. Any number that is divisible by 6 is divisible by 2.; (b) yes. Any number that is divisible by 6 is divisible by 3.; (c) no. $4 \in B$, but $4 \notin A$.; (d) no. $4 \in B$, but $4 \notin C$.; (e) no. $3 \in C$, but $3 \notin A$.; (f) no. $3 \in C$, but $3 \notin B$.

5.21 We will use the result of example 5.20. A subset of $\{a, b, c, d\}$ either contains d or it does not. Since the subsets of $\{a, b, c\}$ do not contain d, we simply list all the subsets of $\{a, b, c\}$ and then to each one of them we add d. This gives

$$\begin{array}{lllllll}
S_1 & = & \varnothing & \quad & S_9 & = & \{d\} \\
S_2 & = & \{a\} & & S_{10} & = & \{a, d\} \\
S_3 & = & \{b\} & & S_{11} & = & \{b, d\} \\
S_4 & = & \{c\} & & S_{12} & = & \{c, d\} \\
S_5 & = & \{a, b\} & & S_{13} & = & \{a, b, d\} \\
S_6 & = & \{b, c\} & & S_{14} & = & \{b, c, d\} \\
S_7 & = & \{a, c\} & & S_{15} & = & \{a, c, d\} \\
S_8 & = & \{a, b, c\} & & S_{16} & = & \{a, b, c, d\}
\end{array}$$

5.24 Based on the answer to Exercise 5.21, we have that $P(\{a,b,c,d\}) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a,b\}, \{b,c\}, \{a,c\}, \{a,b,c\}, \{d\}, \{a,d\}, \{b,d\}, \{c,d\}, \{a,b,d\}, \{b,c,d\}, \{a,c,d\}, \{a,b,c,d\}\}$. Notice that a list of these 16 sets not separated by commas and not enclosed in {} is not correct. It may have the correct *content*, but it is not in the proper *form*.

5.26 (a) By Theorem 5.25, $|P(A)| = 2^4 = 16$. (b) Similarly, $|P(P(A))| = 2^{16} = 65536$. (c) This is just getting a bit ridiculous, but the answer is $|P(P(P(A)))| = 2^{65536}$.

5.27 Applying Theorem 5.25, it is not too hard to see that the power set will be twice as big after a single element is added.

5.30 $\mathbb{Z}$, or the set of (all) integers.

5.33 $\emptyset$.

5.36 A; B.

5.40 B; A.

5.44 Since no integer is both even and odd, A and B are disjoint.

5.48

Evaluation of Proof 1: This solution has several problems.

1. $x \in \{A - B\}$ means 'x is an element of the set containing $A - B$,' not 'x is an element of $A - B$.' What they meant was '$x \in A - B$.'
2. At the end of the first sentence, 'x is not $\in B$' mixes mathematical notation and English in a strange way. This should be either '$x \notin B$' or 'x is not in B.'
3. In the second sentence, the phrase '$x \in A$ and $\overline{B}$' is a strange mixture of math and English that is potentially ambiguous. It should be rephrased as something like '$x \in A$ and $x \in \overline{B}$' or 'x is in both A and $\overline{B}$.'
4. Finally, what has been shown here is that $A - B \subseteq A \cap \overline{B}$. This is only half of the proof. They still need to prove that $A \cap \overline{B} \subseteq A - B$.

Evaluation of Proof 2: Overall, this proof is very confusing and unclear. More specifically,

1. This is an attempt at working through what each set is by using the definitions. That would be fine except for two things. First, they were asked to give a set containment proof. Second, the wording of the proof is confusing and hard to follow. I do not come away from this with a sense that anything has been proven.
2. They are not using the terminology properly. The terms 'universe' or 'universal set' would be appropriate, but not 'universal' on its own (used twice). Similarly, what does the phrase 'all intersection part' mean? Also, a set doesn't 'return' anything. A set is just a set. It contains elements, but it doesn't 'do' anything.

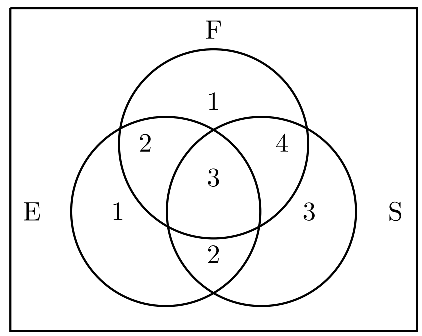
Evaluation of Proof 3: This proof contains a lot of correct elements. In fact, the first half is on the right track. However, they jumped from $x \in A$ and $x \notin B$ to $x \in A \cap \overline{B}$. Between these statements they should say something like '$x \notin B$ is equivalent to $x \in \overline{B}$' since the latter statement is really needed before they can conclude that $x \in A \cap \overline{B}$. Also, it would be better if they had 'by the definition of intersection' before or after the statement $x \in A \cap \overline{B}$. Finally, it would help clarify the proof if the end was something like 'We have shown that whenever $x \in A - B$, $x \in A \cap \overline{B}$. Thus, $A - B \subseteq A \cap \overline{B}$.'

The second half of the proof starts out well, but has serious flaws. The statement 'This means that $x \in A$ and $x \notin B$' should be justified by the definitions of complement and intersection, and might even involve two steps. This is the same problem they had in the first half of the proof. More serious is the statement 'which is what we just proved in the previous statement'. What exactly does that mean? It is unclear how 'what we just proved' immediately leads us to the conclusion that $A - B = A \cap \overline{B}$. First we need to establish that $x \in A - B$ based on the previous statements (easy). Then we can say that $A \cap \overline{B} \subseteq A - B$. Finally, we can combine this with the first part of the proof to say that $A - B = A \cap \overline{B}$.

In summary, the first half is pretty good. It should at least make the connection between $x \notin B$ and $x \in \overline{B}$. The other suggestions clarify the proof a little, but the proof would be O.K. if they were omitted. The second half is another story. It doesn't really prove anything, but instead makes a vague appeal to something that was proven before. Not only is what they are referring to unclear, but how the proof of one direction is related to the proof of the other direction is also unclear.

5.50 $x \in C$; $x \in B$; definition of union; $(x \in B \wedge x \in C)$; distributive law (the logical one); $(x \in A \cap C)$; definition of intersection; definition of union.

5.53 Let E be the set of all English speakers, S the set of Spanish speakers and F the set of French speakers in our group. We fill-up the Venn diagram (to the right) successively. In the intersection of all three we put 3. In the region common to E and S which is not filled up we put $5 - 3 = 2$. In the region common to E and F which is not already filled up we put $5 - 3 = 2$. In the region common to S and F which is not already filled up, we put $7 - 3 = 4$. In the remaining part of E we put $8 - 2 - 3 - 2 = 1$, in the remaining part of S we put $12 - 4 - 3 - 2 = 3$, and in the remaining part of F we put $10 - 2 - 3 - 4 = 1$. Therefore, $1 + 2 + 3 + 4 + 1 + 2 + 3 = 16$ people speak at least one of these languages.



5.56 $A \times B = \{(1,3), (2,3), (3,3), (4,3)\}$.

5.59 $A^2 = \{(0,0), (0,1), (1,0), (1,1)\}$.
$A^3 = \{(0,0,0), (0,1,0), (1,0,0), (1,1,0), (0,0,1), (0,1,1), (1,0,1), (1,1,1)\}$.

5.62 (a) $10 * 50 = 500$ (b) $10 * 20 = 200$ (c) $10 * 10 = 100$ (d) $50 * 50 * 50 = 125,000$ (e) $10 * 50 * 20 = 10,000$

5.63

Evaluation of Solution 1: Although it is on the right track, this solution has several problems. First, it would be better to make it more clear that the assumption is that *both* A and B are not empty. But the bigger problem is the statement '(a, b) is in the cross product'. The problem is that a and b are not defined anywhere. Saying 'where $a \in A$ and $b \in B$' earlier does not guarantee that there is such an a or b. The proof needs to say something along the lines of 'Since A and B are not empty, then there exist some $a \in A$ and $b \in B$. Therefore $(a, b) \in A \times B \ldots$'

Evaluation of Solution 2: This one is way off. The proof is essentially saying 'Notice that $p \to q$. Therefore $q \to p$.' But these are not equivalent statements. Although it is true that if both A and B are the empty set, then $A \times B$ is also the empty set, this does not prove that both A and B must be empty in order for $A \times B$ to be empty. In fact, this isn't the correct conclusion.

Evaluation of Solution 3: The conclusion is incorrect, as is the proof. The problem is that the negation of 'both A and B are empty' is 'it is not the case that both A and B are empty' or 'at least one of A or B is not empty,' which is not the same thing as 'neither A nor B is empty.' So although the proof seems to be correct, it is not. The reason it seems almost correct is that except for this error, the rest of the proof follows proper proof techniques. Unfortunately, all it takes is one error to make a proof invalid.

Evaluation of Solution 4: This is a correct conclusion and proof.

5.70 $f(x) = x \bmod 2$ works. The domain is $\mathbb{Z}$, and the codomain can be a variety of things. $\mathbb{Z}$, $\mathbb{N}$, and $\{0,1\}$ are the most obvious choices. Note that we can pick any of these since the only requirement of the codomain is that the range is a subset of it. On the other hand, $\mathbb{R}$, $\mathbb{C}$ and $\mathbb{Q}$ could also all be given as the codomain, but they wouldn't make nearly as much sense.
5.74 We never said it was *always* wrong to work both sides of an equation. If you are working on an equation that you know to be true, there is absolutely nothing wrong with it. It is a problem only when you are starting with something you don't know to be true. In this case, we know that $2a-3 = 2b-3$ is true given the assumption made. Therefore, we are free to 'work both sides'.
5.75 Let $a, b \in \mathbb{R}$. If $f(a) = f(b)$, then $5a = 5b$. Dividing both sides by 5, we get $a = b$. Thus, f is one-to-one.
5.78 Notice that $f(4.5) = f(4) = 4$, so clearly f is not one-to-one. (Your proof may involve different numbers, but should be this simple.)
5.81 Notice that if $y = 2x+1$, then $y-1 = 2x$ and $x = (y-1)/2$. Let $b \in \mathbb{R}$. Then $f((b-1)/2) = 2((b-1)/2)+1 = b-1+1 = b$. Thus, every $b \in \mathbb{R}$ is mapped to by f, so f is onto.
5.84 Since the floor of any number is an integer, there is no a such that $f(a) = 4.5$ (for instance). Thus, f is not onto.
5.85 (a) f **is not one-to-one.** See Example 5.77 for a proof. (b) The same proof from Example 5.77 works over the reals. But I guess it doesn't hurt to repeat it: Since $f(-1) = f(1) = 1$, f **is not one-to-one.** (c) Let $a, b \in \mathbb{N}$. If $f(a) = f(b)$, that means $a^2 = b^2$. Taking the square root of both sides, we obtain $\sqrt{a^2} = \sqrt{b^2}$, or $|a| = |b|$ (if you didn't remember that $\sqrt{x^2} = |x|$, you do now). But since $a, b \in \mathbb{N}$, $|a| = a$ and $|b| = b$. Thus, $a = b$. Thus, f **is one-to-one.**
5.86 (a) Notice that if $f(a) = f(b)$, then $a+2 = b+2$ so $a = b$. Thus, f is one-to-one. Also notice that for any $b \in \mathbb{Z}$, $f(b-2) = b-2+2 = b$, so f is onto.
(b) Since $g(1) = g(-1) = 1$, g is not one-to-one. Also notice that there is no integer a such that $g(a) = a^2 = 5$, so g is not onto. (c) If $h(a) = h(b)$, then $2a = 2b$ so $a = b$. Thus, h is one-to-one. But there is no integer a such that $h(a) = 2a = 3$, so a is not onto. (d) Notice that $r(0) = \lfloor 0/2 \rfloor = \lfloor 0 \rfloor = 0$ and $r(1) = \lfloor 1/2 \rfloor = \lfloor 0 \rfloor = 0$, so r is not one-to-one. But for any integer b, $r(2b) = \lfloor 2b/2 \rfloor = \lfloor b \rfloor = b$, so r is onto.
5.88 (a) F. Consider $f(x) = \lfloor x \rfloor$ from $\mathbb{R}$ to $\mathbb{Z}$.
(b) F. Consider $f(x) = x^2$ from $\mathbb{R}$ to $\mathbb{R}$ which is not one-to-one.
(c) T. See Theorem 5.87.
(d) F. f maps 1 to two different values, so it isn't a function.
(e) T. We previously showed it was onto, and it isn't difficult to see that it is one-to-one.
(f) F. f is not onto, but it is one-to-one.
(g) T. By definition of range, it is a subset of the codomain.
(h) F. We have seen several counter examples to this.
(i) F. If $a = 2$ and $b = 0$, the odd numbers are not in the range.
(j) F. Same counterexample as the previous question.
(k) T. The proof is similar to several previous proofs.
5.94 If $f(a) = f(b)$, $3a-5 = 3b-5$. Subtracting 5 from both sides and then dividing both sides by 3, we get $a = b$. Thus, f is one-to-one. If $b \in \mathbb{R}$, notice that $f((b+5)/3) = 3((b+5)/3) - 5 =$

$b+5-5=b$, so there is some value that maps to b. Therefore, f is onto. Since f is one-to-one and onto, it has an inverse. To find the inverse, we let $y=3x-5$. Then $3x=y+5$, so $x=(y+5)/3$. Thus, $f^{-1}(x)=(y+5)/3$ (or $\frac{y}{3}+\frac{5}{3}$).

5.97 $(f\circ g)(x)=f(x/2)=\lfloor x/2\rfloor$, and $(g\circ f)(x)=g(\lfloor x\rfloor)=(\lfloor x\rfloor)/2$.

5.102 (a) F. f might not be onto–e.g. if $a=2$ and $b=0$.
(b) F. Same reason as the previous question.
(c) T. Since over the reals, f is one-to-one and onto.
(d) F. There are several problems. First, x^2 may not even have an inverse depending on the domain (which was not specified). Second, even if it had an inverse, it certainly wouldn't be $1/x^2$. That's its reciprocal, not its inverse. Its inverse would be $\sqrt{x}$ (again, assuming the domain was chosen so that it is invertible).
(e) F. This is only true if n is odd.
(f) F. $\sqrt{2}\notin\mathbb{N}$, so not only is it not invertible, it can't even be defined on $\mathbb{N}$.
(g) T. The nth root of a positive number is defined for all positive real numbers, so the function is well defined. It is not too difficult to convince yourself that the function is both one-to-one and onto when restricted to positive numbers, so it is invertible.
(h) T. In both cases you get $1/x^2$.
(i) F. $(f\circ g)(x)=f(x+1)=(x+1+1)^2=(x+2)^2=x^2+4x+4$, and $(g\circ f)(x)=g((x+1)^2)=(x+1)^2+1=x^2+2x+2$, which are clearly not the same.
(j) F. $(f\circ g)(x)=\lceil x\rceil$, and $(g\circ f)(x)=\lfloor x\rfloor$. (We'll leave it to you to see why this is the case.)
(k) F. Certainly not. $f(3.5)=3$, but $g(3)=3$, not 3.5.
(l) T. With the restricted domain, they are indeed inverses.

5.110 The following three cases probably make the most sense: When $a=b$, when $a<b$ and when $a>b$. These make sense because these are likely different cases in the code. Mathematically, we can think of it as follows. The possible inputs are from the set $\mathbb{Z}\times\mathbb{Z}$. The partition we have in mind is $A=\{(a,a):a\in\mathbb{Z}\}$, $B=\{(a,b):a,b\in\mathbb{Z},a<b\}$, and $C=\{(a,b):a,b\in\mathbb{Z},a>b\}$. Convince yourself that these sets form a partition of $\mathbb{Z}\times\mathbb{Z}$. That is, they are all disjoint from each other and $\mathbb{Z}\times\mathbb{Z}=A\cup B\cup C$.

Alternatively, you might have thought in terms of a and/or b being positive, negative, or 0. Although that may make some sense, given that we are comparing a and b with each other, it probably doesn't matter exactly what values a and b have (i.e. whether they are positive, negative, or 0), but what values they have *relative to each other*. That is why the first answer is much better. With that being said, it wouldn't hurt to include several tests for each of our three cases that involve various combinations of positive, negative, and zero values.

5.111 Did you define two or more subsets of $\mathbb{Z}$? Are they all non-empty? Do none of them intersect with each other? If you take the union of all of them, do you get $\mathbb{Z}$? If so, your answer is correct! If not, try again.

5.113 Since $\mathbb{R}=\mathbb{Q}\cup\mathbb{I}$ and $\mathbb{Q}\cap\mathbb{I}=\varnothing$, $\{\mathbb{Q},\mathbb{I}\}$ is a partition of $\mathbb{R}$. Hopefully this comes as no surprise.

5.118 R is a subset of $\mathbb{Z}\times\mathbb{Z}$, so it is a relation. By the way, this relation should look familiar. Did you read the solution to Exercise 5.110?

5.119 Is it a subset of $\mathbb{Z}^+\times\mathbb{Z}^+$? It is. So it is a relation on $\mathbb{Z}^+$.

5.121 (a) T is **not** reflexive since you cannot be taller than yourself. (b) N is reflexive because everybody's name starts with the same letter as their name does. (c) C is reflexive because everybody have been to the same city as they have been in. (d) K is **not** reflexive because you know who you are, so it is not the case that you don't know who you are. That is, $(a,a)\notin K$ for any a. (e) R is **not** reflexive because (Donald Knuth, Donald Knuth) (for instance) is not in the relation.

5.123 (a) T is **not** symmetric since if a is taller than b, b is clearly not taller than a. (b) N is symmetric since if a's name starts with the same letter as b's name, clearly b's name starts with the same letter as a's name. (c) C is symmetric since it is worded such that it doesn't distinguish between the first and second item in the pair. In other words, if a and b have been to the same city, then b and a have been to the same city. (d) K is **not** symmetric since $(\text{David Letterman}, \text{Chuck Cusack}) \in K$, but $(\text{Chuck Cusack}, \text{David Letterman}) \notin K$. (e) R is not symmetric since $(\text{Barack Obama}, \text{George W. Bush}) \in R$, but (George W. Bush, Barack Obama)$\notin R$.
5.125 (a) Just knowing that $(1,1) \in R$ is not enough to tell either way. (b) On the other hand, if $(1,2)$ and $(2,1)$ are both in R, it is clearly **not** anti-symmetric.
5.126 This is just the contrapositive of the original definition.
5.127 (a) T is anti-symmetric since whenever $a \neq b$, if a is taller than b, then b is not taller than a, so if $(a,b) \in T$, then $(b,a) \notin T$. (b) N is **not** anti-symmetric since $(\text{Bono}, \text{Boy George})$ and $(\text{Boy George}, \text{Bono})$ are both in N. (c) C is **not** anti-symmetric since $(\text{Bono}, \text{The Edge})$ and $(\text{The Edge}, \text{Bono})$ are both in C (since they have played many concerts together, they have certainly been in the same city at least once). (d) K is **not** anti-symmetric because both $(\text{Dirk Benedict}, \text{Jon Blake Cusack 2.0})$ and $(\text{Jon Blake Cusack 2.0}, \text{Dirk Benedict})$ are in K. (e) R is anti-symmetric since it only contains one element, $(\text{Barack Obama}, \text{George W. Bush})$, and (George W. Bush, Barack Obama)$\notin R$.
5.128 (a) No. The relation $R = \{(1,2),(2,1),(1,3)\}$ is neither symmetric ($(3,1) \notin R$) nor anti-symmetric ($(1,2)$ and $(2,1)$ are both in R).
(b) No. For example, R from answer (a) is not anti-symmetric, but isn't symmetric either.
(c) Yes. If you answered incorrectly, don't worry. You get to think about why the answer is 'yes' in the next exercise.
5.129 Many answers will work, but they all have the same thing in common: They only contain 'diagonal' elements (but not necessarily all of the diagonal elements). For instance, let $R = \{(a,a) : a \in \mathbb{Z}\}$. Go back to the definitions for symmetric and anti-symmetric and verify that this is indeed both. Another examples is $R = \{(\text{Ken}, \text{Ken})\}$ on the set of English words.
5.131 (a) T is transitive since if a is taller than b, and b is taller than c, clearly a is taller than c. In other words $(a,b) \in R$ and $(b,c) \in R$ implies that $(a,c) \in R$. (b) N is transitive because if a's name starts with the same letter as b's name, and b's name starts with the same letter as c's name, clearly it is the same letter in all of them, so a's name starts with the same letter as c's. (c) C is **not** transitive. You might think a similar argument as in (a) and (b) works here, but it doesn't. The proof from (b) works because names start with a single letter, so transitivity holds. But if $(a,b) \in C$ and $(b,c) \in C$, it might be because a and b have both been to Chicago, and b and c have both been to New York, but that a has never been to New York. In this case, $(a,c) \notin C$. So C is not transitive. (d) K is not transitive. For instance, $(\text{David Letterman}, \text{Chuck Cusack}) \in K$ and $(\text{Chuck Cusack}, \text{David Letterman's son}) \in K$, but $(\text{David Letterman}, \text{David Letterman's son}) \notin K$ since I sure hope he knows his own son. (e) R is transitive since there isn't even an $a,b,c \in R$ such that (a,b) and (b,c) are both in R, so it holds vacuously.
5.134 (a) T is **not** an equivalence relation since it is not symmetric. (b) N is an equivalence relation since it is reflexive, symmetric, and transitive. (c) C is **not** an equivalence relation since it is not transitive. (d) K is **not** an equivalence relation since it is not reflexive, symmetric, or transitive. This one isn't even close! (e) R is **not** an equivalence relation since it is not reflexive.
5.136 (a) T is a **not** partial order because it is not reflexive. (b) N is **not** a partial order since it is not anti-symmetric. (c) C is **not** a partial order since it is not anti-symmetric or transitive. (d) K is **not** a partial order since it is not reflexive, anti-symmetric, or transitive. (e) R is **not** a

partial order since it is not reflexive.
5.137 In the following, A, B, and C are elements of X. As such, they are sets.
(**Reflexive**) Since $A \subseteq A$, $(A, A) \in R$, so R is reflexive.
(**Anti-symmetric**) If $(A, B) \in R$ and $(B, A) \in R$, then we know that $A \subseteq B$ and $B \subseteq A$. By Theorem 5.45, this implies that $A = B$. Therefore R is anti-symmetric.
(**Transitive**) If $(A, B) \in R$ and $(B, C) \in R$, then $A \subseteq B$ and $B \subseteq C$. But the definition of $\subseteq$ implies that $A \subseteq C$, so $(A, C) \in R$, and R is transitive.
Since R is reflexive, anti-symmetric, and transitive, it is a partial order.
5.138 (a) Since $(1, 1) \notin R$, R is not reflexive. (b) Since $(1, 2) \in R$, but $(2, 1) \notin R$, R is not symmetric. (c) A careful examination of the elements reveals that it *is* anti-symmetric. (d) A careful examination of the elements reveals that it *is* transitive. (e) Since it is not reflexive or symmetric, it is not an equivalence relation. (f) Since it is not reflexive, it is not a partial order.
5.140 $((a, b), (a, b))$; bc; da; $((c, d), (a, b))$; symmetric; $ad = bc$; $cf = de$; de/f; $b(de/f)$; $af = be$; $((a, b), (e, f))$
6.3 (a)$x_0 = 1+(-2)^0 = 1+1 = 2$ (b)$x_1 = 1+(-2)^1 = 1-2 = -1$ (c)$x_2 = 1+(-2)^2 = 1+4 = 5$ (d)$x_3 = 1 + (-2)^3 = 1 - 8 = -7$ (e)$x_4 = 1 + (-2)^4 = 1 + 16 = 17$
6.4 We will just provide the final answer for these. If you can't get these answers, you may need to brush up on your algebra skills. (a) 2, 1/2,5/4, 7/8, 17/16; (b) 2, 2, 3, 7, 25; (c) 1/3, 1/5, 1/25, 1/119, 1/721; (d) 2, 9/4, 64/27, 625/256, 7776/3125
6.7 Notice that $x_0 = 1$, $x_1 = 5 \cdot 1 = 5$, $x_2 = 5 \cdot 5 = 5^2$, $x_3 = 5 \cdot 5^2 = 5^3$, etc. Looking back, we can see that $1 = 5^0$, so $x_0 = 5^0$. Also, $x_1 = 5 = 5^1$. So it seems likely that the solution is $x_n = 5^n$. This is *not* a proof, though!
6.8 Notice that $x_0 = 1$, $x_1 = 1 \cdot 1 = 1$, $x_2 = 2 \cdot 1 = 2$, $x_3 = 3 \cdot 2 = 6$, $x_4 = 4 \cdot 6 = 24$, $x_3 = 5 \cdot 24 = 120$, etc. Written this way, no obvious pattern is emerging. Sometimes *how* you write the numbers matters. Let's try this again: $x_1 = 1 \cdot 1 = 1!$, $x_2 = 2 \cdot 1 = 2!$, $x_3 = 3 \cdot 2 \cdot 1 = 3!$, $x_4 = 4 \cdot 3 \cdot 2 \cdot 1 = 4!$, $x_3 = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 5!$, etc. Now we can see that $x_n = n!$ is a likely solution. Again, this isn't a proof.
6.9 Their calculations are correct (Did you check them with a calculator? You *should* have! How else can you tell whether or not their solution is correct?). So it does seem like $a_n = 2^n$ is the correct solution. However,

$$a_5 = \left\lfloor \tfrac{1+\sqrt{5}}{2} \times a_4 \right\rfloor + a_3 = \left\lfloor \tfrac{1+\sqrt{5}}{2} \times 16 \right\rfloor + 8 = 33 \neq 2^5$$

so the solution that seems 'obvious' turns out to incorrect. We won't give the actual solution since the point of this example is to demonstrate that just because a pattern holds for the first several terms of a sequence, it does not guarantee that it holds for the whole sequence.
6.11 Hopefully you came up with the solution $x_n = 5^n$. Since $x_0 = 1 = 5^0$, it works for the initial condition. If we plug this back into the right hand side of $x_n = 5 \cdot x_{n-1}$, we get

$$\begin{aligned} 5 \cdot x_{n-1} &= 5 \cdot 5^{n-1} \\ &= 5^n \\ &= x_n, \end{aligned}$$

which verifies the formula. Therefore $x_n = 5^n$ is the solution.
6.12 Hopefully you came up with the solution $x_n = n!$. Since $x_0 = 1 = 0!$, it works for the initial condition. If we plug this back into the right hand side of $x_n = n \cdot x_{n-1}$, we get

$$\begin{aligned} n \cdot x_{n-1} &= n \cdot (n-1)! \\ &= n! \\ &= x_n, \end{aligned}$$

which verifies the formula. Therefore $x_n = n!$ is the solution.

6.13 The computations are correct, the conclusion is correct, but unfortunately, the final code has a serious problem. It works *most* of the time, but it does not deal with negative values correctly. It should return 3 for all negative values, but it continues to use the formula. The problem is they forgot to even consider what the function does for negative values of n. They probably could have formatted their answer better, too. It's difficult to follow in paragraph form. They could have put the various values of `ferzle(n)` each on their own line and presented it mathematically instead of in sentence form. For instance, instead of 'ferzle(1) returns ferzle(0)+2, which is $3 + 2 = 5$,' they should have '$ferzle(1) = ferzle(0) + 2 = 3 + 2 = 5$.' It would have made it much easier to see the pattern.

6.14

```
int ferzle(int n) {
        if(n<=0) {
            return 3;
    } else {
            return 2*n+3;
        }
}
```

6.18 We didn't do anything wrong. We wrote the inequality in the other order, and the indexes are one lower than those given in the definition. But that's O.K. The definition is simply trying to convey the idea that every term is strictly greater than the previous. That is what we showed. We can show that $x_n < x_{n+1}$, $x_{n+1} > x_n$, $x_{n-1} < x_n$, or $x_n > x_{n-1}$. They all mean essentially the same thing. The only difference is the order in which the inequalities are written (the first two and last two are saying exactly the same thing–we just flipped the inequality) and what values of n are valid. For instance, if the sequence starts at 0, then we need to assume $n \geq 0$ for the first pair of inequalities and $n \geq 1$ for the second pair.

6.20 If you got stuck on this one, first realize that $x_n = \dfrac{n^2+1}{n} = n + \dfrac{1}{n}$. This form might make the algebra a little easier. Then, follow the technique of the previous example–show that $x_{n+1} - x_n > 0$. So, if necessary, go back and try again. If you already attempted a proof, you may proceed to read the solution.

Notice that,

$$\begin{aligned} x_{n+1} - x_n &= \left(n+1+\frac{1}{n+1}\right) - \left(n + \frac{1}{n}\right) \\ &= 1 + \frac{1}{n+1} - \frac{1}{n} \\ &= 1 - \frac{1}{n(n+1)} \\ &> 0, \end{aligned}$$

the last step since $1/n(n + 1) < 1$ when $n \geq 1$. Therefore, $x_{n+1} - x_n > 0$, so $x_{n+1} > x_n$, i.e., the sequence is strictly increasing. If your solution is significantly different than this, make sure you determine one way or another if it is correct.

6.21 We could go into much more detail than we do here, and hopefully you did when you wrote down your solutions. But we'll settle for short, informal arguments this time. (a) This is just a linear function. It is **strictly increasing**. (b) Since this keeps going from positive to negative to positive, etc. it is **non-monotonic**. (c) We know that $n!$ is strictly increasing. Since this is the reciprocal of that function, it is almost *strictly decreasing* (since we are dividing by a number that is getting larger). However, since $1/0! = 1/1! = 1$, it is just **decreasing**. (d) This is getting closer to 1 as n increases. It is **strictly increasing** (e) This is $n(n - 1)$. $x_1 = 0$, $x_2 = 2$,

$x_3 = 6$, etc. Each term is multiplying two numbers that are both getting larger, so it is **strictly increasing**. (f) This is similar to the previous one, but $x_0 = x_1 = 0$, so it is just **increasing**. (g) This alternates between -1 and 1, so it is **non-monotonic**. (h) Each term subtracts from 1 a smaller numbers than the last term, so it is **strictly increasing**. (i) Each term adds to 1 a smaller number than the last term, so it is **strictly decreasing**.

6.24 You should have concluded that $a = -\frac{2}{3^{17}}$ and that $r = \frac{2}{3^{16}} / \left(-\frac{2}{3^{17}}\right) = -3^{17}/3^{16} = -3$ (or you could have divided the second and third terms). Then the n-th term is $-\frac{2}{3^{17}}(-3)^{n-1} = \frac{2(-1)^n}{3^{18-n}}$ (Make sure you can do the algebra to get to this simplified form). Finally, the 17th term is $\frac{2(-1)^{17}}{3^{18-17}} = -\dfrac{2}{3}$

6.26 We are given that $ar^5 = 20$ and $ar^9 = 320$. Dividing, we can see that $r^4 = 16$. Thus $r = \pm 2$. (We don't have enough information to know which it is). Since $ar^5 = 20$, we know that $a = 20/r^5 = \pm 20/32$. So the third term is $ar^2 = (\pm 20/32)(\pm 2)^2 = \pm 80/32 = \pm 5/2$. Thus $|ar^2| = 5/2$.

6.30

(a) The difference between the each of the first 4 terms of the sequence is 7, so it appears to be an arithmetic sequence. Doing a little math, the correct answer appears to be (d) 51.

(b) Although the sequence *appears to be* arithmetic, we cannot be certain that it is. If you are told it is arithmetic, then 51 is absolutely the correct answer. Notice that the previous example specifically stated that you should assume that the pattern continues. This one did not. Without being told this, the rest of the sequence could be anything. The 8th term could be 0 or 8, 675, 309 for all we know. Of the choices given, 51 is the most *obvious* choice, but any of the answers could be correct. This is one reason I hate these sorts of questions on tests.

Although I think it is important to point out the flaw in these sorts of questions, it is also important to conform to the expectations when answering such questions on standardized tests. In other words, instead of disputing the question (as some students might be inclined to do), just go with the obvious interpretation.

6.31 (a) The closed form was $x_n = 5^n$, which is clearly geometric (with $a = 1$ and $r = 5$) and not arithmetic. (b) Since the solution for this one is $x_n = n!$, this is neither arithmetic or geometric. (c) Since the sequence is essentially $f_n = 2n + 3$, with initial condition $f_0 = 3$, it is an arithmetic sequence. It is clearly not geometric.

6.34 $\displaystyle\sum_{i=0}^{100} y^i$

6.36 $\displaystyle\sum_{i=0}^{50} (y^2)^i$ or $\displaystyle\sum_{i=0}^{50} y^{2i}$

6.38 (a) 2 (b)11 (c) 100 (d) 101

6.43 (a) $\displaystyle\sum_{k=5}^{6} 5 = 5\sum_{k=5}^{6} 1 = 5 \cdot 2 = 10.$

(b) $\displaystyle\sum_{k=20}^{30} 200 = 200\sum_{k=20}^{30} 1 = 200(30 - 20 + 1) = 200 * 11 = 2200.$

6.46 Using Theorem 6.44, we get the following answers (a) $2 \cdot 5 = 10$. (b) $(30 - 20 + 1)200 = 11 * 200 = 2200$. (c) 900 (d) 909. Notice that this one has one more term than the previous one. The fact that the additional index is 0 doesn't matter since it is adding 9 for that term.

6.47 This solution contains an 'off by one' error. The correct answer is $10(75-25+1) = 10 * 51 = 510$.

6.50 (a) $20 \cdot 21/2 = 210$ (b) $100 \cdot 101/2 = 5050$ (c) $1000 \cdot 1001/2 = 500500$

6.51

Evaluation of Solution 1: Another example of the 'off by one error'. They are using the formula $n(n-1)/2$ instead of $n(n+1)/2$.

Evaluation of Solution 2: This answer doesn't even make sense. What is k in the answer? k is just an index of the summation. The index should *never* appear in the answer. The problem is that you can't pull the k out of the sum since each term in the sum depends on it.

6.52 It is true. The additional term that the sum adds is 0, so the sum is the same whether or not it starts at 0 or 1.

6.55 $\displaystyle\sum_{i=1}^{100} 2-i = \sum_{i=1}^{100} 2 - \sum_{i=1}^{100} i = 200 - \frac{100 \cdot 101}{2} = 200 - 5050 = -4850.$

6.57 The sum of the first n odd integers is

$$\sum_{k=1}^{n}(2k-1) = \sum_{k=1}^{n} 2k - \sum_{k=1}^{n} 1 = 2\sum_{k=1}^{n} k - \sum_{k=1}^{n} 1 = 2\frac{n(n+1)}{2} - n = n^2 + n - n = n^2.$$

6.59 (a) $\displaystyle\sum_{k=10}^{20} k = \sum_{k=1}^{20} k - \sum_{k=1}^{9} k = 20 \cdot 21/2 - 9 \cdot 10/2 = 210 - 45 = 165.$

(b) $\displaystyle\sum_{k=21}^{40} k = \sum_{k=1}^{40} k - \sum_{k=1}^{20} k = 40 \cdot 41/2 - 20 \cdot 21/2 = 820 - 210 = 610.$

6.60

Evaluation of Solution 1: Another example of the off-by-one error. The second sum should end at 29, not 30.

Evaluation of Solution 2: This one has two errors, one of which is repeated twice. It has the same error as the previous solution, but it also uses the incorrect formula for each of the sums (the off-by-one error).

Evaluation of Solution 3: This one is correct.

6.61 Two errors are made that cancel each other out. The first error is that the second sum in the second step should go to 29, not 30. But in the computation of that sum in the next step, the formula $n(n-1)/2$ is used instead of $n(n+1)/2$ (The correct formula was used for the first sum). This is a rare case where an off-by-one error is followed by the opposite off-by-one error that results in the correct answer.

It should be emphasized that even though the correct answer is obtained, *this is an incorrect solution*. They obtained the correct answer by sheer luck.

6.63 There are two ways to answer this. The smart aleck answer is 'because it is correct.' But *why* is it correct with 2, and couldn't it be slightly modified to work with 1 or 0? The answer is *no* because if you plug 1 or 0 into $\frac{1}{(k-1)k}$, you get a division by 0. Hopefully I don't need to tell you that this is a bad thing.

6.64

$$\begin{aligned}
\sum_{k=1}^{n} k^3+k &= \sum_{k=1}^{n} k^3+\sum_{k=1}^{n} k \\
&= \frac{n^2(n+1)^2}{4}+\frac{n(n+1)}{2} \\
&= \frac{n(n+1)}{2}\left(\frac{n(n+1)}{2}+1\right) \\
&= \frac{n(n+1)}{2}\left(\frac{n^2+n+2}{2}\right) \\
&= \frac{n(n+1)(n^2+n+2)}{4}
\end{aligned}$$

6.66 (a) $\sum_{i=1}^{n}\sum_{j=1}^{i} 1=\sum_{i=1}^{n} i=\frac{n(n+1)}{2}$.

(b) $\sum_{i=1}^{n}\sum_{j=1}^{i} j=\sum_{i=1}^{n}\frac{i(i+1)}{2}=\cdots=\frac{n(n+1)(n+2)}{6}$. (This one involves doing a little algebra, applying two formulas, and then doing a little more algebra. Make sure you work it out until you get this answer.)

(c) $\sum_{i=1}^{n}\sum_{j=1}^{n} ij=\sum_{i=1}^{n}\left(i\sum_{j=1}^{n} j\right)=\sum_{i=1}^{n}\left(i\frac{n(n+1)}{2}\right)=\frac{n(n+1)}{2}\sum_{i=1}^{n} i=\frac{n(n+1)}{2}\frac{n(n+1)}{2}=\frac{n^2(n+1)^2}{4}$.

6.69 (a) $\frac{3^{50}-1}{2}$ (b) $\frac{1-y^{101}}{1-y}$ or $\frac{y^{101}-1}{y-1}$ (We won't give the alternatives for the rest. If your answer differs, do some algebra to make sure it is equivalent.) (c) $\frac{1-(-y)^{101}}{1-(-y)}=\frac{1+y^{101}}{1+y}$ (d) $\frac{1-y^{102}}{1-y^2}$.

6.72 $x^5-1=(x-1)(x^4+x^3+x^2+x+1)$.

6.73 $2^1+2^2+2^3+\cdots+2^{n+1}$; 2^0; 2^{n+1}; $2^{n+1}-2^0$

6.75 $a\sum_{k=0}^{n} r^k$; $a\frac{1-r^{k+1}}{1-r}$.

6.76 Let $S=a+ar+ar^2+\cdots+ar^n$. Then $rS=ar+ar^2+\cdots+ar^{n+1}Z$, so

$$\begin{aligned}
S-rS &= a+ar+ar^2+\cdots+ar^n-ar-ar^2-\cdots-ar^{n+1} \\
&= a-ar^{n+1}.
\end{aligned}$$

From this we deduce that

$$S=\frac{a-ar^{n+1}}{1-r},$$

that is,

$$\sum_{k=0}^{n} ar^k=\frac{a-ar^{n+1}}{1-r}.$$

7.6 (a) Since we assumed that $n\geq 1$, $-3n$ is certainly negative. In other words, $-3n\leq 0$. That's why in the first step we could say that $5n^2-3n+20\leq 5n^2+20$. (b) We used the fact that $20\leq 20n^2$ whenever $n\geq 1$.

If either of these solutions is not clear to you, you need to brush up on your algebra.

7.7 This is incorrect. It is *not true* that $-12n \leq -12n^2$ when $n \geq 1$. (If this isn't clear to you after thinking about it for a few minutes, you may need to do some algebra review.) In fact, that error led to the statement $4n^2 - 12n + 10 \leq 2n^2$ which cannot possibly be true as n gets larger since it would require that $2n^2 - 12n + 10 \leq 0$. This is not true as n gets larger. In fact, when $n = 10$, for instance, it is clearly not true. But it *is true* that $-12n < 0$ when $n \geq 0$, so instead of replacing it with $-12n^2$, it should be replaced with 0 as in previous examples.

7.8 (a) Sure. Add the final step of $25n^2 \leq 50n^2$ to the algebra in the proof. In fact, any number above 25 can easily be used. Some values under 25 can also be used, but they would require a modification of the algebra used in the proof. The bottom line is that there is generally no 'right' value to use for c. If you find a value that works, then it's fine. (b) Clearly not. For this to work, we would need $5n^2 - 3n + 20 < 2n^2$ to hold as n increases towards ∞. But this would imply that $3n^2 - 3n + 20 < 0$. But when $n \geq 1$, $3n^2$ is positive and larger than $3n$, so $3n^2 - 3n + 20 > 0$. (c) Sure. The proof used the fact that the inequality is true when $n \geq 1$, so it is clearly also true if $n \geq 100$. And the definition of Big-O does not require that we use the smallest possible value for n_0. (d) No. We would need a constant c such that $5 \cdot 0^2 - 3 \cdot 0 + 20 = 20 \leq 0 = c \cdot 0^2$, which is clearly impossible.

7.9 If $n \geq 1$,

$$\begin{aligned} 5n^5 - 4n^4 + 3n^3 - 2n^2 + n &\leq 5n^5 + 3n^3 + n \\ &\leq 5n^5 + 3n^5 + n^5 \\ &= 9n^5. \end{aligned}$$

Therefore, $5n^5 - 4n^4 + 3n^3 - 2n^2 + n = O(n^5)$.

7.10 We used $n_0 = 1$ and $c = 9$. Your values for n_0 and c may differ. This is O.K. if you have the correct algebra to back it up.

7.13 Since $4n^2 \leq 4n^2 + n + 1$ for $n \geq 0$, $4n^2 = \Omega(n^2)$.

7.14 We used $c = 4$ and $n_0 = 0$. You might have used $n_0 = 1$ or some other positive value. As long as you chose a positive value for n_0, it works just fine. You could have also used any value for c larger than 0 and at most 4.

7.17 It is O.K. Since the second inequality holds when $n \geq 0$, it also holds when $n \geq 1$.

In general, when you want to combine inequalities that contain two different assumptions, you simply make the more restrictive assumption. In this case, $n \geq 1$ is more restrictive than $n \geq 0$. In general, if you have assumptions $n \geq a$ and $n \geq b$, then to combine the results with these assumptions, you assume $n \geq max(a, b)$.

7.22 $g(n)$ appears in the denominator of a fraction. If at some point it does not become (and remain) non-zero, the limit in the definition will be undefined. If you never took a calculus course and are not that familiar with limits, do not worry a whole lot about this subtle point.

7.23 o is like $<$ and ω is like $>$.

7.24 (a) No. If $f(n) = \Theta(g(n))$, f and g grow at the same rate. But $f(n) = o(g(n))$ expresses the idea that f grows slower than g. It is impossible for f to grow at the same rate as g and slower than g. (b) Yes! If f grows no faster than g, then it is possible that it grows slower. For instance, $n = O(n^2)$ and $n = o(n^2)$ are both true. (c) No. If f and g grow at the same rate, then $f(n) = O(g(n))$, but $f(n) \neq o(g(n))$. For instance, $3n = O(n)$, but $3n \neq o(n)$. (d) Yes. In fact, it is guaranteed! If f grows slower than g, then f grows no faster than g.

7.25

Evaluation of Solution 1: Although this proof sounds somewhat reasonable, it is way too informal and convoluted. Here are some of the problems.

1. This student misunderstands the concept behind 'ignoring the constants.' We can ignore the constants *after we know that* $f(n) = O(g(n))$. We can't ignore them in order to prove it.
2. The phrase 'become irrelevant' (used twice) is not precise. We have developed mathematical notation for a reason—it allows us to make statements like these precise. It's kind of like saying that a certain car costs 'a lot'. What is 'a lot'? Although \$30,000 might be a lot for most of us, people with a lot more money than I have might not think that \$500,000 is a lot.
3. The phrase 'This leaves us with $n^k + n^{k-1} + \cdots + n = O(n^k)$' is odd. What precisely do they mean? That this is true or that this is what we need to prove now? In either case, it is incorrect. Similarly for the second time they use the phrase 'This leaves us with'.
4. The second half of the proof is unnecessarily convoluted. They essentially are claiming that their proof has boiled down to showing that $n^k = O(n^k)$. To prove this, they use an incredibly drawn out, yet vague, explanation that is in a single unnecessarily long sentence. Why are they even bringing Θ and Ω into this proof? Why don't they just say something like 'since $n^k \leq 1n^k$ for all $n \geq 1$, $n^k = O(n^k)$'? I believe the answer is obvious: they don't really understand what they are doing here. They clearly have a vague understanding of the notation, but they don't understand the formal definition.

The bottom line is that this student understands that the statement they needed to prove is correct, and they have a vague sense of *why* it is true, but they did not have a clear understanding of how to use the definition of Big-O to prove it. The most important thing to take away from this example is this: *Be precise, use the notation and definitions you have learned, and if your proofs look a lot different than those in the book, you might question whether or not you are on the right track.*

Evaluation of Solution 2: This proof is correct.

7.26 We cannot say anything about the relative growth rates of $f(n)$ and $g(n)$ because we are only given upper bounds for each. It is possible that $f(n) = n^2$ and $g(n) = n$, so that $f(n)$ grows faster, or vice-versa. They could also both be n.

7.27 (a) F. This is saying that $f(n)$ grows *no* faster than $g(n)$.
(b) F. They grow at the same rate.
(c) F. $f(n)$ might grow slower than $g(n)$. For instance, $f(n) = n$ and $g(n) = n^2$.
(d) F. They might grow at the same rate. For instance, $f(n) = g(n) = n$.
(e) F. If $f(n) = n$ and $g(n) = n^2$, $f(n) = O(g(n))$, but $f(n) \neq \Omega(g(n))$.
(f) T. By Theorem 7.18.
(g) F. If $f(n) = n$ and $g(n) = n^2$, $f(n) = O(g(n))$, but $f(n) \neq \Theta(g(n))$.
(h) F. If $f(n) = n$ and $g(n) = n^2$, $f(n) = O(g(n))$, but $g(n) \neq O(f(n))$.

7.37 $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$; $\frac{1}{c_2} f(n)$; $\frac{1}{c_1} f(n)$; $c_3 h(n) \leq g(n) \leq c_4 h(n)$ for all $n \geq n_1$.; c_2; $c_2 c_4$; $\max\{n_0, n_1\}$; $c_1 c_3 h(n)$; $c_2 c_4 h(n)$; $\Theta(h(n))$; Θ; transitive;

7.39 (a) T. By Theorem 7.36.
(b) T. By Theorem 7.18.
(c) T. By Theorem 7.32.
(d) F. The backwards implication is true, but the forward one is not. For instance, if $f(n) = n$ and $g(n) = n^2$, clearly $f(n) = O(g(n))$, but $f(n) \neq \Theta(g(n))$.

(e) F. Neither direction is true. For instance, if $f(n) = n$ and $g(n) = n^2$, $f(n) = O(g(n))$, but $g(n) \neq O(f(n))$.
(f) T. By Theorem 7.36.
(g) T. By Theorem 7.18.
(h) T. By Theorem 7.28.
7.42 $c_1 n^2$; $c_2 n^2$; $\frac{1}{2} - \frac{3}{n}$; $\frac{10-6}{20} = \frac{1}{5}$; $\frac{1}{5}n^2$; $\frac{1}{2}n^2$; 10.
7.43 There are a few ways to think about this. First, the larger n is, the smaller $\frac{3}{n}$ is, so a smaller amount is being subtracted. But that's perhaps too fuzzy. Let's look at it this way:

$$n \geq 10 \Rightarrow \frac{10}{3} \leq \frac{n}{3} \Rightarrow \frac{3}{n} \leq \frac{3}{10} \Rightarrow -\frac{3}{n} \geq -\frac{3}{10} \Rightarrow \frac{1}{2} - \frac{3}{n} \geq \frac{1}{2} - \frac{3}{10}.$$

7.45 (a) Theorem 7.18. (b) Absolutely not! Theorem 7.18 requires that we also prove $f(n) = \Omega(g(n))$. Here is a counterexample: $n = O(n^2)$, but $n \neq \Theta(n^2)$. So $f(n) = O(g(n))$ does not imply that $f(n) = \Theta(g(n))$.
7.46 Notice that when $n \geq 1$, $n! = 1 \cdot 2 \cdot 3 \cdots n \leq n \cdot n \cdots n = n^n$. Therefore $n! = O(n^n)$ (We used $n_0 = 1$, and $c = 1$.)
7.49 If $f(x) = O(g(x))$, then there are positive constants c_1 and n_0' such that

$$0 \leq f(n) \leq c_1\, g(n) \text{ for all } n \geq n_0',$$

and if $g(x) = O(h(x))$, then there are positive constants c_2 and n_0'' such that

$$0 \leq g(n) \leq c_2\, h(n) \text{ for all } n \geq n_0''.$$

Set $n_0 = \max(n_0', n_0'')$ and $c_3 = c_1\, c_2$. Then

$$0 \leq f(n) \leq c_1\, g(n) \leq c_1\, c_2\, h(n) = c_3\, h(n) \text{ for all } n \geq n_0.$$

Thus $f(x) = O(h(x))$.
7.53 (a) ∞ (b) ∞ (c) ∞ (d) ∞ (e) 0 (f) 0 (g) 8675309
7.57 Theorem 7.51 part (b) implies that $\lim\limits_{n\to\infty} n^2 = \infty$. Since the limit being computed was actually $\lim\limits_{n\to\infty} \dfrac{1}{n^2}$, Theorem 7.55 was used to obtain the final answer of 0 for the limit.
7.58 Notice that $\lim_{x\to\infty} \frac{3x^3}{x^2} = \lim_{x\to\infty} 3x = \infty$, so $3x^3 = \omega(x^2)$ by the second case of the Theorem 7.50, which also implies that $3x^3 = \Omega(x^2)$.
7.63 Notice that $\lim\limits_{n\to\infty} \dfrac{n(n+1)/2}{n^2} = \lim\limits_{n\to\infty} \dfrac{n^2+n}{2n^2} = \lim\limits_{n\to\infty} \dfrac{1}{2} + \dfrac{1}{2n} = \dfrac{1}{2} + 0 = \dfrac{1}{2}$, so $n(n+1)/2 = \Theta(n^2)$.
7.64 (a) Since $\lim\limits_{x\to\infty} \dfrac{2^x}{3^x} = \lim\limits_{x\to\infty} \left(\dfrac{2}{3}\right)^x = 0$, the result follows.
(b) If $x \geq 1$, then clearly $(3/2)^x \geq 1$, so $2^x \leq 2^x \left(\frac{3}{2}\right)^x = \left(\frac{2\times 3}{2}\right)^x = 3^x$. Therefore, $2^x = O(3^x)$.
7.70

Evaluation of Proof 1: 7^x grows faster than 5^x does not mean $7^x - 5^x > 0$ for all $x \neq 0$. For one thing, we are really only concerned about positive values of x. Further, we are specifically concerned about very large values of x. In other words, we want something to be true for all x that are 'large enough'. Also, this statement does not take into account constant factors. Similarly, a tight bound does *not* imply that $7^x - 5^x = 0$. The bottom line: This one is way off. They are not conveying an understanding of what 'upper bound' really means, and they certainly haven't proven anything. Frankly, I don't think they have a clue what they are trying to say in this proof.

Evaluation of Proof 2: This one has several problems. First, the application of l'Hopital's rule is incorrect. The result should be $\lim_{x\to\infty} \frac{5^x \log 5}{7^x \log 7}$, which should make it obvious that l'Hopital's rule doesn't actually help in this case. (The key to this one is to do a little algebra.) The next problem is the statement 'but $x \log 7$ gets there faster'. What exactly does that mean? Asymptotically faster, or just faster? If the former, it needs to be proven. If the latter, that isn't enough to prove relative growth rates. Finally, even if this showed that $5^x = O(7^x)$, that only shows that 7^x is an upper bound on 5^x. It does not show that the bound is not tight. The bottom line is that bad algebra combined with vague statements falls way short of a correct proof.

Evaluation of Proof 3: This proof is *very* close to being correct. The main problem is that they only stated that $5^x = O(7^x)$, but they also needed to show that $5^x \neq \Theta(7^x)$. It turns out that the theorem they mention also gives them that. So all they needed to add is 'and $5^x \neq \Theta(7^x)$' at the end. Technically, there is another problem—they should have taken the limit of $5^x/7^x$. What they really showed using the limit theorem is that $7^x = \omega(5^x)$, which is equivalent to $5^x = o(7^x)$. It isn't a major problem, but technically the limit theorem does not directly give them the result they say it does. If you are trying to prove that $f(x)$ is bounded by $g(x)$, put $f(x)$ on the top and $g(x)$ on the bottom.

7.72 You should have come up with $n^2 \log n$ for the upper bound. If you didn't, now that you know the answer, go back and try to write the proofs before reading them here. (a) If $n > 1$,

$$\ln(n^2+1) \leq \ln(n^2+n^2) = \ln(2n^2) = (\ln 2 + \ln n^2) \leq (\ln n + 2\ln n) = 3\ln n$$

Thus when $n > 1$,

$$n\ln(n^2+1) + n^2\ln n \leq n3\ln n + n^2\ln n \leq 3n^2\ln n + n^2\ln n \leq 4n^2\ln n.$$

Thus, $n\ln(n^2+1) + n^2\ln n = O(n^2\ln n)$. (You may have different algebra in your proof. Just make certain that however you did it that it is correct.)

(b)
$$\begin{aligned}
\lim_{x\to\infty} \frac{n\ln(n^2+1) + n^2\ln n}{n^2\ln n} &= \lim_{x\to\infty} \frac{n\ln(n^2+1)}{n^2\ln n} + 1 \\
&= 1 + \lim_{x\to\infty} \frac{\ln(n^2+1)}{n\ln n} \\
&= 1 + \lim_{x\to\infty} \frac{\frac{2n}{n^2+1}}{1\cdot\ln n + n\cdot\frac{1}{n}} && \text{(l'Hopital)} \\
&= 1 + \lim_{x\to\infty} \frac{2n}{(n^2+1)(\ln n+1)} \\
&= 1 + \lim_{x\to\infty} \frac{2}{2n(\ln n+1) + (n^2+1)\cdot\frac{1}{n}} && \text{(l'Hopital)} \\
&= 1 + \lim_{x\to\infty} \frac{2}{2n(\ln n+1) + n + \frac{1}{n}} \\
&= 1 + 0 = 1.
\end{aligned}$$

Therefore, $n\ln(n^2+1) + n^2\ln n = \Theta(n^2\log n)$.

7.74 We can see that $(n^2-1)^5 = \Theta(n^{10})$ since

$$\lim_{n\to\infty}\frac{(n^2-1)^5}{n^{10}} = \lim_{n\to\infty}\left(\frac{n^2-1}{n^2}\right)^5 = \lim_{n\to\infty}\left(1-\frac{1}{n^2}\right)^5 = 1.$$

7.75 The following limit shows that $2^{n+1}+5^{n-1} = \Theta(5^n)$.

$$\lim_{n\to\infty}\frac{2^{n+1}+5^{n-1}}{5^n} = \lim_{n\to\infty}\frac{2^{n+1}}{5^n}+\frac{5^{n-1}}{5^n} = \lim_{n\to\infty}2\left(\frac{2}{5}\right)^n+\frac{1}{5} = 0+\frac{1}{5}.$$

Note that we could also have shown that $2^{n+1}+5^{n-1} = \Theta(5^{n-1})$, but that is not as simple of a function.

7.78 Since $a < b$, $b-a > 0$. Therefore, $\lim_{n\to\infty}\frac{n^a}{n^b} = \lim_{n\to\infty} n^{a-b} = \lim_{n\to\infty}\frac{1}{n^{b-a}} = 0$. By Theorem 7.50, $n^a = o(n^b)$.

7.81 Since $a < b$, $a/b < 1$. Therefore, $\lim_{n\to\infty}\frac{a^n}{b^n} = \lim_{n\to\infty}\left(\frac{a}{b}\right)^n = 0$. By Theorem 7.50, $a^n = o(b^n)$.

7.86 (a) False since 3^n grows faster than 2^n. (b) True since 2^n grows slower than 3^n. (c) False since 3^n grows faster than 2^n, which means it does not grow slower or at the same rate. (d) True since they both have the same growth rate. Remember, exponentials with different bases have different growth rates, but logarithms with different bases have the same growth rate. (e) True since they have the same growth rate. Remember that if $f(n) = \Theta(g(n))$, then $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. (f) False since they have the same growth rate, so $\log_{10} n$ does not grow slower than $\log_3 n$.

7.89 Using l'Hopital's rule, we have $\lim_{n\to\infty}\frac{\log_c(n)}{n^b} = \lim_{n\to\infty}\frac{\frac{1}{n\ln(c)}}{b\,n^{b-1}} = \lim_{n\to\infty}\frac{1}{\ln(c)b\,n^b} = 0$ since $b > 0$. Thus, Theorem 7.50 tells us that $\log_c n = o(n^b)$.

7.94 (a) Θ; (b) o (O is correct, but not precise enough.); (c) Θ; (d) o (O is correct, but not precise enough.); (e) Θ since $2^n = 2\,2^{n-1}$; (f) Ω (Technically it is ω, but I'll let it slide if you put Ω since we haven't used ω much.); (g) through (j) are all o (O is correct, but not precise enough.)

7.96 If your answers do not all start with Θ, go back and redo them before reading the answers. Your answers should match the following *exactly*. (a) $\Theta(n^7)$. (b) $\Theta(n^8)$. (c) $\Theta(n^2)$. (d) $\Theta(3^n)$. (e) $\Theta(2^n)$. (f) $\Theta(n^2)$. (g) $\Theta(n^{.000001})$. (h) $\Theta(n^n)$.

7.97 Here is the correct ranking:
10000
$\log x$, $\log(x^{300})$
$\log^{300} x$
$x^{.000001}$
x, $\log(2^x)$
$x\log(x)$
$x^{log_2 3}$
x^2
x^5
2^x
3^x

7.98 Modern computers use multitasking to perform several tasks (seemingly) at the same time. Therefore, if an algorithm takes 1 minute of real time (wall-clock time), it might be that 58 seconds of that time was spent running the algorithm, but it could also be the case that only 30 seconds of that time were spent on that algorithm, and the other 30 seconds spent on other processes. In this case, the CPU time would be 30 seconds, but the wall-clock time 60 seconds.

Further complicating matters is increasing availability of machines with multiple processors. If an algorithm runs on 4 processors rather than one, it might take 1/4th the time in terms of wall-clock time, but it will probably take the same amount of CPU time (or close to it).

7.99 We cannot be certain whose algorithm is better with the given information. Maybe Sue used a *TRS-80 Model IV* from the 1980s to run her program and Stu used *Tianhe-2* (The fastest computer in the world from about 2013-2014). In this case, it is possible that if Sue ran her program on *Tianhe-2* it would have only taken 2 minutes, making her the real winner.

7.100 As has already been mentioned, other processes on the machine can have a significant influence on the wall-clock time. For instance, if I run two CPU-intensive programs at once, the wall-clock time of each might be about twice what it would be if I ran them one at a time. If they are run on a machine with multiple cores the wall-clock time might be closer to the CPU-time. But other processes that are running can still throw off the numbers.

7.101 For the most part, yes. This is especially true if the running times of the algorithms are not too close to each other (in other words, if one of the algorithms is significantly faster than the other). However, the number of other processes running on the machine can have an influence on CPU-time. For instance, if there are more processes running, there are more context switches, and depending on how the CPU-time is counted, these context switches can influence the runtime. So although comparing the CPU-time of two algorithms that are run on the same computer gives a pretty good indication of which is better, it is still not perfect.

7.103 This one is a little more tricky. The answer is $n \cdot m$ since this is how many entries are in the matrix. Sometimes we need to use two numbers to specify the input size. As suggested previously, we will ignore the size of the two other pieces of data.

7.109 We focus on the assignment (=) inside the loop and ignore the other instructions. This should be fine since assignment occurs at least as often as any other instruction. In addition, it is important to note that `max` takes constant time (did you remember to explicitly say this?), as do all of the other operations, so we aren't under-counting. It isn't too difficult to see that the assignment will occur n times for an array of size n since the code goes through a loop with $i = 0, \ldots, n-1$. Thus, the complexity of `maximum` is always $\Theta(n)$. That is, $\Theta(n)$ is the best, average, and worst-case complexity of `maximum`.

7.112 The line in the inner for loop takes constant time (let's call it c). The inner loop executes $k = 50$ times, each time doing c operations. Thus the inner loop does $50 \cdot c$ operations, which is still just a constant. The outer loop executes n times, each time executing the inner loop, which takes $50 \cdot c$ operations. Thus, the whole algorithm takes $50 \cdot c \cdot n = \Theta(n)$ time.

7.113 The line in the inner for loop takes constant time (let's call it c). The inner loop executes n^2 times since j is going from 0 to $n^2 - 1$, so each time the inner loop executes, it does cn^2 operations. The outer loop executes n times, each time executing the inner loop. Thus, the total time is $n \times cn^2 = \Theta(n^3)$.

This is an example of an algorithm with a double-nested loop that is *worse than* $\Theta(n^2)$. The point of this exercise is to make it clear that you should never jump to conclusions too quickly when analyzing algorithms. Read the limits on loops very carefully!

7.116 (a) `AreaTrapezoid` is constant. (b) `factorial`is not constant. It should be easy to see that it has a complexity of $\Theta(n)$. (c) `absoluteValue` is constant if we assume `sqrt` takes constant time.

7.121 Although it has a nested loop, the inside loop always executes 6 times, which is a constant. So the algorithm takes about $6 \cdot c \cdot n = \Theta(n)$ operations, not $\Theta(n^2)$.

7.126 (a) Since `factorial` has a complexity of $\Theta(n)$, it is not quadratic. (b) Since there are

n^2 entries it to consider, the algorithm takes $\Theta(n^2)$ time, so it would be quadratic.[2]

7.127 Bubble sort, selection sort, and insertion sort are three of them that you may have seen before.

7.133 As we mentioned in our analysis, executing the conditional statement takes about 3 operations, and if it is true, about 3 additional operations are performed. So the worst case is no more than about twice as many operations as the best case. In other words, we are comparing $c \cdot n^2$ to $2c \cdot n^2$, both of which are $\Theta(n^2)$.

7.136 Since both of these methods require accessing the ith element of the list for some integer i, and since we must traverse the list from the head, clearly the complexity of both methods is $\Theta(i)$.

We could be less specific and say that the complexity is $\Theta(n)$ since $0 \leq i < n$. However, when analyzing algorithms that make repeated calls to these methods, using $\Theta(i)$ might give a more accurate answer overall. It makes the analysis more difficult, but sometimes it is worth it.

Note: For doubly-linked lists, some implementations traverse starting at the tail if the index is closer to the end of the list. However, that just means the complexity is no worse than $\Theta(n/2) = \Theta(n)$*. In other words, it only changes the complexity by a constant factor.*

7.138 All of them should be $\Theta(1)$, assuming we keep track of how many elements are currently in the stack (which is a reasonable thing to do).

7.139 For an array, either enqueue or dequeue (but not both) will be $\Theta(n)$. All of the others will be $\Theta(1)$, assuming we keep track of how many elements are currently in the queue. Note that the advantage of the circular array implementation is that both enqueue and dequeue are $\Theta(1)$.

7.140 For the array implementation, addToFront, removeFirst and contains will all be $\Theta(n)$ and the rest will be $\Theta(1)$. For the linked list implementation, contains will be $\Theta(n)$ and the rest will be constant if we assume there is both a head and tail pointer. If there is no tail pointer, addToEnd will be $\Theta(n)$.

7.141 For unbalanced, all operations can be done in $\Theta(h)$ time, where h is the height of the tree. This is the best answer you can give. You could also say $O(n)$ time since the height is no more then n, but this answer is not precise enough to be of much use. You *cannot* say $\Theta(\log n)$ since this is not necessarily true for an unbalanced tree.

For balanced (red-black, AVL, etc.), all operations can be implemented with complexity $\Theta(\log n)$.

7.142 The average-case complexity for all of these operations is $\Theta(1)$, and the worst-case complexity is $\Theta(n)$.

7.143

Evaluation of Solution 1: I have no idea what logic they are trying to use here. Sure, a^n is an exponential function, but what does that have to do with how long this algorithm takes? This solution is way off.

Evaluation of Solution 2: Having the i in the answer is nonsense since it doesn't mean anything in the context of a complexity—it is just a variable that happens to be used to index a loop. Further, the answer should be given using Θ-notation. So this solution is just plain wrong. Since having an i in the complexity does not make any sense, this person either has a fundamental misunderstanding of how to analyze algorithms or they didn't think about their final answer. Don't be like this person!

Evaluation of Solution 3: This solution is O.K., but it has a slight problem. Although the analysis given estimates the worst-case behavior, it over-estimates it. By replacing i with $n-1$,

[2] Technically this is linear with respect to the *size* of the input since the size of the input is n^2. But it is quadratic in n. In either case, it is $\Theta(n^2)$.

they are over-estimating how long the algorithm takes. The call to `pow` only takes $n-1$ time once. This solution can really only tell us that the complexity is $O(n^2)$. Is it possible the over-estimation of time resulted in a bound that isn't tight? Even if it turns out that $\Theta(n^2)$ is the correct bound, this solution does not prove it. Although they are on the right track, this person needed to be a little more careful in their analysis.

7.144 Before you read too far: if you did not use a summation in your solution, go back and try again! This is very similar to the analysis of `bubblesort`. The for loop takes i from 0 to $n-1$, and each time the code in the loop takes i time (since that is how long `power(a,i)` takes). Thus, the complexity is

$$\sum_{i=0}^{n-1} i = \frac{(n-1)n}{2} = \Theta(n^2).$$

Notice that just because the answer is $\Theta(n^2)$, that does not mean that the third solution to Evaluate 7.143 was correct. As we stated in the solution to that problem, because they overestimated the number of operations, they only proved that the algorithm has complexity $O(n^2)$.

7.145 Here is one solution.

```
double addPowers(double a, int n) {
    if(a==1) {
        return n+1;
    } else {
        double sum = 1; // for the $a^0$ term.
        double pow = 1;
        for(int i=1;i<n;i++) {
            pow = pow*a;
            sum += pow;
        }
        return sum;
    }
}
```

If $a=1$, the algorithm takes constant time. Otherwise, it executes a constant number of operations and a single for loop n times. The code in the loop takes constant time. Thus the algorithm takes $\Theta(n)$ time.

7.146 If you used recursion instead of a loop, cool idea. However, go back and do it again. There is an even *simpler* way to do it. Need a hint? Apply some of that discrete mathematics material you have been learning! When you have a solution that does not use a loop or recursion (or you get stuck), keep reading.

The trick is to use the formula for a geometric series (did you recognize that this is what `addPowers` is really computing?). We need a special case for $a=1$ because the formula requires that $a \neq 1$.

```
double addPowers(double a, int n) {
    if(a==1) {
        return n+1;
    } else {
        return (1-power(a,n+1))/(1-a);
    }
}
```

If $a=1$, the algorithm takes constant time. Otherwise, it executes a constant number of operations and a single call to `power(a,n+1)` which takes $n+1$ time. Thus the algorithm takes $\Theta(n+1) = \Theta(n)$ time.

It is worth noting that $a = 0$ is a tricky case. `addPowers` can't really be computed for $a = 0$ since 0^0 is undefined. It is for this reason that the first term of a geometric sequence is technically 1, not a^0. Since $a^0 = 1$ for all other values of a, the case of $a = 0$ is usually glossed over. If you don't understand what the fuss is about, don't worry too much about it.

7.148

Evaluation of Solution 1: This takes about $2+4n+4(m-1) = 4n+4m-2$ operations, which is essentially the same as the 'C' version. Unfortunately, it is slightly worse than the original solution since it is now incorrect. All they did is omit adding the final n in the first sum. This went from a 'C' to a 'D' (at best).

Evaluation of Solution 2: This one takes about $2+4(n-1-m+1) = 2+4(n-m)$ operations. This is a lot better than the previous solutions. Unfortunately, it misses adding the final n, so it is incorrect. It also is not as efficient as possible. I'd say this is still a 'C'.

Evaluation of Solution 3: This student figured out the trick—they know a formula to compute the sum, so they tried to use it. Unfortunately, they used the formula incorrectly and/or they made a mistake when manipulating the sum (it is impossible to tell exactly what they did wrong—they made either one or two errors), so the algorithm is not correct. In terms of efficiency, their solution is great because it takes a constant number of operations no matter what n and m are. Because their answer is efficient and *very close* to being correct, I'd probably give them a 'B'.

7.149 We use the fact that $\displaystyle\sum_{k=m}^{n} k = \sum_{k=1}^{n} - \sum_{k=1}^{m-1} = \frac{n(n+1)}{2} - \frac{(m-1)m}{2}$ to give us the following solution:

```
int sumFromMToN(int m,int n) {
      return n*(n+1)/2 - (m-1)*m/2;
   }
```

Since this is just doing a fixed number of arithmetic operations no matter what the values of m and n are, it takes constant time.

7.152 The analysis of these is very similar to the analysis of Examples 7.151, so the details are omitted. (a) For a LinkedList the `contains` method takes $\Theta(m)$ time, so the overall complexity is $\Theta(nm)$. (b)For a HashSet it takes $\Theta(1)$ time to call `contains` (on average), so the overall complexity is $\Theta(n+n) = \Theta(n)$.

7.154 (a) contains takes $\Theta(m)$ time so the complexity is $\Theta(n(\log n + m))$.
(b) Here the contains method takes $\Theta(\log m)$ time, so the overall complexity is $\Theta(n(\log n + \log m))$ (or $\Theta(n \log n + n \log m)$ if you prefer to write it that way).
Note that we don't know the relationship between n and m so we can't simplify either answer.

7.156

n		$\lfloor n/2 \rfloor$	
decimal	binary	decimal	binary
12	1100	6	110
13	1101	6	110
32	100000	16	10000
33	100001	16	10000
118	1110110	59	111011
119	1110111	59	111011

7.157 The next theorem answers the question about the pattern.

8.2 (a) No. The domain is $\mathbb{Z}$, which does not have a 'starting point'. (b) Yes. The domain is $\mathbb{Z}^+$. (c) Yes. The domain is $\{2, 3, 4, \ldots\}$. (d) Yes. The domain is $\mathbb{Z}^+$. (e) No. The domain is $\mathbb{R}$ which is not a subset of $\mathbb{Z}$. Thus, not only is there no 'starting point,' there is no clear ordering of the real numbers from one to the next.
8.4 Modus ponens.
8.5 You can immediately conclude that $P(6)$ is true using modus ponens. If that was your answer, good. But you can keep going. Since $P(6)$ is true, you can conclude that $P(7)$ is true (also by modus ponens). But then you can conclude that $P(8)$ is true. And so on. The most complete answer you can give is that $P(n)$ is true for all $n \geq 5$. You *cannot* conclude that $P(n)$ is true for all $n \geq 1$ because we don't know anything about the truth values of $P(1)$, $P(2)$, $P(3)$, and $P(4)$.
8.6 There are various ways to say this, including what was said in the paragraph above. Here is another way to say it:

> If $P(a)$ is true, and for any value of $k \geq a$, $P(k)$ true implies that $P(k+1)$ is true, then $P(n)$ is true for all $n \geq a$.

8.7 If you answered **yes** and you aren't lying, great! If you answered *no* or you answered **yes** but you lied, it is important that you think about it some more and/or get some help. If you want to succeed at writing induction proofs, understanding this is an important step!
8.10 $\dfrac{1(1+1)}{2}$; $P(1)$ is true; $P(k)$ is true; $\displaystyle\sum_{i=1}^{k} i = \frac{k(k+1)}{2}$; $P(k+1)$; $\displaystyle\sum_{i=1}^{k+1} i = \frac{(k+1)(k+2)}{2}$; $\displaystyle\sum_{i=1}^{k} i$; $\dfrac{k(k+1)}{2}$; $\dfrac{k}{2}+1$; $\dfrac{(k+1)(k+2)}{2}$; $P(k+1)$ is true; $P(1)$ is true; $k \geq 1$; all $n \geq 1$; induction *or* the principle of mathematical induction *or* PMI.
8.11

(a) $P(k)$ is the statement "$\displaystyle\sum_{i=1}^{k} i \cdot i! = (k+1)! - 1$"

(b) $P(k+1)$ is the statement "$\displaystyle\sum_{i=1}^{k+1} i \cdot i! = (k+2)! - 1$"

(c) $LHS(k) = \displaystyle\sum_{i=1}^{k} i \cdot i!$

(d) $RHS(k) = (k+1)! - 1$

(e) $LHS(k+1) = \displaystyle\sum_{i=1}^{k+1} i \cdot i!$

(f) $RHS(k+1) = (k+2)! - 1$

8.14 Define $P(n)$ to be the statement "$\displaystyle\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$". We need to show that $P(n)$ is true for all $n \geq 1$.
Base Case: Since $\displaystyle\sum_{i=1}^{1} i^2 = 1^2 = 1 = \frac{1(2)(3)}{6}$, $P(1)$ is true. (If your algebra is in a different order,

like $\sum_{i=1}^{1} i^2 = \frac{1(2)(3)}{6} = 1$, it is incorrect. We only know that $\sum_{i=1}^{1} i^2 = \frac{1(2)(3)}{6}$ because we first saw that $\sum_{i=1}^{1} i^2 = 1$, and then were able to see that $1 = \frac{1(2)(3)}{6}$.)

Inductive Hypothesis: Let $k \geq 1$ and assume $P(k)$ is true. That is, $\sum_{i=1}^{k} i^2 = \frac{k(k+1)(2k+1)}{6}$.

As a side note, I know that what I need to prove next is

$$\sum_{i=1}^{k+1} i^2 = \frac{(k+1)(k+2)(2(k+1)+1)}{6} = \frac{(k+1)(k+2)(2k+3)}{6}.$$

I am only writing this down now so that I know what my goal is. I am not going to start working both sides of this or otherwise manipulate it. I can't because I don't know whether or not it is true yet.

Inductive Step: Notice that

$$\begin{aligned}
\sum_{i=1}^{k+1} i^2 &= \sum_{i=1}^{k} i^2 + (k+1)^2 \\
&= \frac{k(k+1)(2k+1)}{6} + (k+1)^2 \\
&= (k+1)\left(\frac{k(2k+1)}{6} + (k+1)\right) \\
&= (k+1)\left(\frac{k(2k+1)+6(k+1)}{6}\right) \\
&= (k+1)\left(\frac{2k^2+k+6k+6}{6}\right) \\
&= (k+1)\left(\frac{2k^2+7k+6}{6}\right) \\
&= (k+1)\left(\frac{(2k+3)(k+2)}{6}\right) \\
&= \frac{(k+1)(k+2)(2k+3)}{6}.
\end{aligned}$$

Therefore $P(k+1)$ is true.
Summary: Since $P(1)$ is true and $P(k) \to P(k+1)$ is true when $k \geq 1$, $P(n)$ is true for all $n \geq 1$ by induction.
8.16 For $k = 1$ we have $1 \cdot 2 = 2 + (1-1)2^2$, and so the statement is true for $n = 1$. Let $k \geq 1$ and assume the statement is true for k. That is, assume

$$1 \cdot 2 + 2 \cdot 2^2 + 3 \cdot 2^3 + \cdots + k \cdot 2^k = 2 + (k-1)2^{k+1}.$$

We need to show that

$$1 \cdot 2 + 2 \cdot 2^2 + 3 \cdot 2^3 + \cdots + (k+1) \cdot 2^{k+1} = 2 + k2^{k+2}.$$

Using some algebra and the inductive hypothesis, we can see that

$$\begin{aligned} 1\cdot 2+2\cdot 2^2+3\cdot 2^3+\cdots+k\cdot 2^k+(k+1)2^{k+1} &= 2+(k-1)2^{k+1}+(k+1)2^{k+1} \\ &= 2+(k-1+k+1)2^{k+1} \\ &= 2+2k2^{k+1} \\ &= 2+k2^{k+2}. \end{aligned}$$

Thus, the result is true for $k+1$. The result follows by induction.

8.18 This proof is very close to being correct, but is suffers from a few small but important errors:

- For the sake of clarity, it might have been better to use k throughout most of the proof instead of n. The exception is in the final sentence where n is correct.
- The base case is just some algebra without context. A few words are needed. For instance, 'notice that when $n=1$,'.
- The base case is presented incorrectly. Notice that the writer starts by writing down what she wants to be true and then deduces that it is indeed correct by doing algebra on both sides of the equation. As we have already mentioned, *you should* **never** *start with what you want to prove and work both sides!* It is not only sloppy, but it can lead to incorrect proofs. Whenever I see students do this, I always tell them to use what I call the U method. What I mean is rewrite your work by starting at the upper left, going down the left side, then doing up the right side. So the above should be rewritten as:

  $$1\cdot 1! = 1 = 2!-1 = (1+1)!-1.$$

  Notice that if the U method does not work (because one or more steps isn't correct), it is probably an indication of an incorrect proof. Consider what happens if you try it on the proof in Exercise 2.91. You would write $-1=(-1)^2=1=1^2=1$. Notice that the first equality is incorrect.

  The U method can sometimes apply to inequalities as well.
- When the writer makes her assumption, she says 'for $n\geq 1$'. This is O.K., but there is some ambiguity here. Does she mean for *all* n, or for a particular value of n? She must mean the latter since the former is what she is trying to prove. It would have been better for her to say 'for some $n\geq 1$.'
- The algebra in the inductive step is perfect. However, what does it mean? She should include something like 'Notice that' before her algebra just to give it a little context. It often doesn't take a lot of words, but adding a few phrases here and there goes a long way to help a proof flow more clearly.
- She says 'Therefore it is true for n'. She must have meant $n+1$ since that is what she just proved.
- As with her assumption, her final statement could be clarified by saying 'for all $n\geq 1$.'

Overall, the proof has almost all of the correct content. Most of the problems have to do with presentation. But as we have seen with other types of proofs, the details are really important to get right!

8.19 Given this proof, we know that $P(1)$ is true. We also know that $P(2) \to P(3)$, $P(3) \to P(4)$, etc, are all true. Unfortunately, the proof omits showing that $P(2)$ is true, so modus ponens never applies. In other words, knowing that $P(2) \to P(3)$ is true does us no good unless we know $P(2)$ is true, which we don't. Because of this, we don't know anything about the truth values of $P(3)$, $P(4)$, etc. The proof either needs to show that $P(2)$ is true as part of the base case, or the inductive step needs to start at 1 instead of 2.

8.21 Because our inductive hypothesis was that $P(k-1)$ is true instead of $P(k)$. If we assumed that $k \geq 0$, then when $k = 0$ it would mean we are assuming $P(-1)$ is true, and we don't know whether or not it is since we never discussed $P(-1)$.

8.25 This contains a very subtle error. Did you find it? If not, go back and carefully re-read the proof and think carefully–at least one thing said in the proof *must* be incorrect. What is it?

O.K., here it is: The statement 'goat 2 is in both collections' is not always true. If $n = 1$, then the first collection contains goats 1 through 1, and the second collection contains goats 2 through 2. In this case, there is no overlap of goat 2, so the proof falls apart.

8.26

Evaluation of Proof 1: This solution is on the right track, but it has several technical problems.

- The base case should be $k = 0$, not $k = 1$.
- The way the base case is worded could be improved. For instance, what purpose does saying '$2 = 2$' serve? Also, the separate sentence that just says 'it is true' is a little vague and awkward. I would reword this as:

  > The total number of palindromes of length $2 \cdot 1$ is $2 = 2^1$, so the statement is true for $k = 1$.

  Of course, the base case should *really* be $k = 0$, but if it were $k = 1$, that is how I would word it.
- The connection between palindromes of length $2k$ and $2(k+1)$ is not entirely clear and is incorrect as stated. A palindrome of length $2(k+1)$ can be formed from a palindrome of length $2k$ by adding a 0 to both the beginning and end or adding a 1 to both the beginning and the end. This what was probably meant, but it is not what the proof actually says.

  But we need to say a little more about this. Every palindrome of length $2(k+1)$ can be formed from exactly one palindrome of length $2k$ with this method. But is this enough? Not quite. We also need to know that every palindrome of length $2k$ can be extended to a palindrome of length $2(k+1)$, and it should be clear that this is the case. In summary, the inductive step needs to establish that there are twice as many binary palindromes of length $2(k+1)$ as there are of length $2k$. The argument has to convince the reader that there is a 2-to-1 correspondence between these sets of palindromes. In other words, we did not omit or double-count any.

Evaluation of Proof 2: The base case correct. Unfortunately, that is about the only thing that is correct.

- The second sentence is wrong. We cannot say that 'it is true for all n'–that is precisely what we are trying to prove. We need to assume it is true for a *particular* n and then prove it is true for $n+1$.
- The rest of the proof is one really long sentence that is difficult to follow. It should be split into much shorter sentences, each of which provides one step of the proof.

- The term 'binary number' should be replaced with 'binary palindrome' throughout. It causes confusion, especially when the words 'add' and 'consecutive' are used. These mean something very different if we have numbers in mind instead of strings.
- I don't think the phrase 'each consecutive binary number' means what the writer thinks it means. The binary numbers 1001 and 1010 are consecutive (representing 9 and 10), but that is probably *not* what the writer has in mind.
- The term 'permutations' shows up for some reason. I think they might have mean 'strings' or something else.
- Why bring up the 4 possible ways to extend a binary string by adding to the beginning and end if only two of them are relevant? Why not just consider the ones of interest in the first place?
- In the context of a proof, the phrase 'you are adding' doesn't make sense. Why am I adding something and what am I adding it to? And do they mean addition (of the binary numbers) or appending (of strings)?
- They switch from n to k in the middle of the proof to provide further confusion.

Evaluation of Proof 3: This proof has most of the right ideas, but it does not put them together well. The base case is correct. It sounds like the writer understands what is going on with the inductive step, but needs to communicate it more clearly. More specifically, what does 'assume $2k \to 2^k$ palindromes' mean? I think I am supposed to read this as 'assume that there are 2^k palindromes of length $2k$.' [3]

The final sentence is also problematic. The first phrase tries to connect to the previous sentence, but the connection needs to be a little more clear. The final phrase is not a complete thought. In the first place, I know that $2^k + 2^k = 2^{k+1}$ and this has nothing to do with the previous phrases. In other words, the 'so' connecting the phrases doesn't make sense. But more seriously, why do I care that $2^k + 2^k = 2^{k+1}$? What he meant was something like 'so there are $2^k + 2^k = 2^{k+1}$ palindromes of length $2k + 2$'.

8.27 The empty string is the only string of length 0, and it is a palindrome. Thus there is $1 = 2^0$ palindromes of length 0.

Now assume there are 2^n binary palindromes of length $2n$. For every palindrome of length $2n$, exactly two palindromes of length $2(n+1)$ can be constructed by appending either a 0 or a 1 to both the beginning and the end. Further, every palindrome of length $2(n+1)$ can be constructed this way. Thus, there are twice as many palindromes of length $2(n+1)$ as there are of length $2n$. By the inductive hypothesis, there are $2 \cdot 2^n = 2^{n+1}$ binary palindromes of length $2(n+1)$.

The result follows by PMI.

8.30 Yes. It clearly calls itself in the else clause.

8.33 (a) The base cases are $n <\leq 0$. (b) The inductive cases are $n > 0$. (c) Yes. For any value $n > 0$, the recursive call uses the value $n - 1$, which is getting closer to the base case of 0.

8.36 Notice that if $n \leq 0$, `countdown(0)` prints nothing, so it works in that case. For $k \geq 0$, assume `countdown(k)` works correctly.[4] Then `countdown(k+1)` will print 'k' and call `countdown(k)`. By the inductive hypothesis, `countdown(k)` will print 'k k-1 ... 2 1', so `countdown(k+1)` will print 'k+1 k k-1 ... 2 1', so it works properly. By PMI, `countdown(n)` works for all $n \geq 0$.

[3] In general, avoid the use of mathematical symbols in constructing the grammar of an English sentence. One of the most common abuses I see is the use of $\to$ in the middle of a sentence.

[4] We are letting $n = 0$ be the base case. You could also let $n = 1$ be the base case, but then you would need to prove that `countdown(1)` works.

8.40 It is pretty clear that the recursive algorithm is much shorter and was a lot easier to write. It is also a lot easier to make a mistake implementing the iterative algorithm. So far, it looks like the recursive algorithm is the clear winner. However, in the next section we will show you why the recursive algorithm we gave should never be implemented. It turns out that is is very inefficient.

The bottom line is that the iterative algorithm is better in this case. Don't feel bad if you thought the recursive algorithm was better. After the next section, you will be better prepared to compare recursive and iterative algorithms in terms of efficiency.

8.43 `PrintN` will print from 1 to n, and `NPrint` will print from n to 1. If you go the answer wrong, go back and convince yourself that this is correct.

8.46 (a) $r_{n/2}$. (b) 1. (c) $a_{n-1} + 2 \cdot a_{n-2} + 3 \cdot a_{n-3} + 4 \cdot a_{n-4}$. (d) There are none.

8.50 It means to find a closed-form expression for it. In other words, one that does not define the sequence recursively.

8.52 When $n = 1$, $T(1) = 1 = 0 + 1 = \log_2 1 + 1$. Assume that $T(k) = \log_2 k + 1$ for all $1 \leq k < n$ (we are using strong induction). Then

$$\begin{aligned} T(n) &= T(n/2) + 1 \\ &= (\log_2(n/2) + 1) + 1 \\ &= \log_2 n - \log_2 2 + 2 \\ &= \log_2 n - 1 + 2 \\ &= \log_2 n + 1. \end{aligned}$$

So by PMI, $T(n) = \log_2 n + 1$ for all $n \geq 1$.

8.54 We begin by computing a few values to see if we can find a pattern. $A(2) = A(1) + 2 = 2 + 2 = 4$, $A(3) = A(2) + 2 = 4 + 2 = 6$, $A(4) = 8$, $A(5) = 10$, etc. It seems pretty obvious that $A(n) = 2n$. It holds for $n = 1$, so we have our base case. Assume $A(n) = 2n$. Then $A(n+1) = A(n) + 2 = 2n + 2 = 2(n+1)$, so it holds for $n+1$. By PMI, $A(n) = 2n$ for all $n \geq 1$.

8.57 It contains 3 very different looking recursive terms so it is very unlikely we will be able to find any sort of meaningful pattern by iteration.

8.59

$$\begin{aligned} H(n) &= 2H(n-1) + 1 \\ &= 2(2H(n-2) + 1) + 1 \\ & \quad = 2^2 H(n-2) + 2 + 1 \\ &= 2^2(2H(n-3) + 1) + 2 + 1 \\ & \quad = 2^3 H(n-3) + 2^2 + 2 + 1 \\ & \vdots \\ &= 2^{n-1}H(1) + 2^{n-2} + 2^{n-3} + \cdots + 2 + 1 \\ &= 2^{n-1} + 2^{n-2} + 2^{n-3} + \cdots + 2 + 1 \\ &= 2^n - 1 \end{aligned}$$

Thus, $H(n) = 2^n - 1$. Luckily, this matches our answer from Example 8.53.

8.61 Iterating a few steps, we discover:

$$\begin{aligned} T(n) &= T(n/2)+1 \\ &= T(n/4)+1+1 \\ &= T(n/2^2)+2 \quad \text{(I think I see a pattern!)} \\ &= T(n/2^3)+1+2 \\ &= T(n/2^3)+3 \quad \text{(I \textit{do} see a pattern!)} \\ &\ \vdots \\ &= T(n/2^k)+k \end{aligned}$$

We need to find k such that $n/2^k = 1$. We already saw in Example 8.58 that $k = \log_2 n$ is the solution. Therefore, we have

$$\begin{aligned} T(n) &= T(n/2^k)+k \\ &= T(n/2^{\log_2 n})+\log_2 n \\ &= T(1)+\log_2 n \\ &= 1+\log_2 n \end{aligned}$$

Therefore, $T(n) = 1 + \log_2 n$.

8.62 The final answer is $T(n) = 2^{n+1} - n - 2$ or $T(n) = 4 \cdot 2^{n-1} - n - 2$. It is important that you can work this out yourself, so try your best to get this answer without looking further. But if you get stuck or have a different answer, you can refer to the following skeleton of steps–we have omitted many of the steps because we want you to work them out. It does provide a few reference points along the way, however.

$$\begin{aligned} T(n) &= 2T(n-1)+n \\ &= 2(2T(n-2)+(n-1))+n \quad \text{(having n instead of $(n-1)$ is a common error)} \\ &= 2^2T(n-2)+3n-2 \quad \text{(it is unclear yet if I should have $3n-2$ or some other form)} \\ &\ \vdots \quad \text{(many skipped steps)} \\ &= 2^kT(n-k)+(2^k-1)n-\sum_{i=1}^{k-1} i2^i \quad \text{(the all-important pattern revealed)} \\ &\ \vdots \quad \text{(plug in appropriate value of k and simplify)} \\ &= 2^{n+1}-n-2. \end{aligned}$$

8.67 Here, $a = 2, b = 2$, and $d = 0$. ($d = 0$ since $1 = 1 \cdot n^0$. In general, $c = c \cdot n^0$, so when $f(n)$ is a constant, $d = 0$.) Since $a > 2^0$, we have $T(n) = \Theta(n^{\log_a a}) = \Theta(n^1) = \Theta(n)$ by the third case of the Master Theorem.

8.69 We have $a = 7$, $b = 2$, and $d = 2$. Since $7 > 2^2$, the third case of the Master Theorem applies so $T(n) = \Theta(n^{\log_2 7})$, which is about $\Theta(n^{2.8})$.

8.70 Because it isn't true. Although the growth rate of $n^{\log_2 7}$ and $n^{2.8}$ are close, they are not exactly the same, so $\Theta(n^{\log_2 7}) \neq \Theta(n^{2.8})$. We *could* say that $T(n) = \Theta(n^{\log_2 7}) = O(n^{2.81})$, but then we have lost the 'tightness' of the bound. And I want to be able to say "Yo dawg, that bound is really *tight*!"

8.71 Here we have $a = 1$, $b = 2$, and $d = 0$. Since $1 = 2^0$, the second case of the Master Theorem tells is that $T(n) = \Theta(n^0 \log n) = \Theta(\log n)$. Since we have already seen several times that $T(n) = \log_2 n + 1$, we can notice that this answer is consistent with those. It's a good thing.
8.77 By raising the subscripts in the homogeneous equation we obtain the characteristic equation $x^n = 9x^{n-1}$ or $x = 9$. A solution to the homogeneous equation will be of the form $x_n = A(9)^n$. Now $f(n) = -56n + 63$ is a polynomial of degree 1 and so we assume that the solution will have the form $x_n = A9^n + Bn + C$. Now $x_0 = 2, x_1 = 9(2) - 56 + 63 = 25, x_2 = 9(25) - 56(2) + 63 = 176$. We thus solve the system

$$2 = A + C,$$

$$25 = 9A + B + C,$$

$$176 = 81A + 2B + C.$$

We find $A = 2, B = 7, C = 0$, so the solution is $x_n = 2(9^n) + 7n$.
8.81 The characteristic equation is $x^2 - 4x + 4 = (x - 2)^2 = 0$. There is a multiple root and so we must test a solution of the form $x_n = A2^n + Bn2^n$. The initial conditions give

$$1 = A,$$

$$4 = 2A + 2B.$$

This solves to $A = 1, B = 1$. The solution is thus $x_n = 2^n + n2^n$.
8.83 We have $a = 2$, $b = 2$, and $d = 1$. Since $2 = 2^1$, we have that $T(n) = \Theta(n \log n)$ by the second case of the Master Theorem.
8.84 (a) C_1 and C_2 are of lower order than n. Thus, we can do this according to part (c) of Theorem 7.28. (b) We know that the $\Theta(n)$ represents some function $f(n)$. By definition of Θ, there are constants, c_1 and c_2 such that $c_1 n \leq f(n) \leq c_2 n$ for all $n \geq n_0$ for some constant n_0. So we essentially replaced $f(n)$ with $c_2 n$ (since we are looking for a worst-case). Note that it doesn't matter what c_2 is. We know there is some constant that works, so we just call it c.
8.86 $T(n) = 2T(n-1) + T(n-5) + T(\sqrt{n}) + 1$ if $n > 5$, $T(1) = T(2) = T(3) = T(4) = T(5) = 1$. You can also have $+c$ instead of $+1$ in the recursive definition.
8.87 Beyond the recursive calls, `StoogeSort` does only a constant amount of work–we'll call it 1. Then it makes three calls with sub-arrays of size $(2/3)n$. Therefore, $T(n) = 3T((2/3)n) + 1$ or $T(n) = 3T(2n/3) + 1$, with base case $T(1) = 1$.
8.88 We can use the Master Theorem for this one. $a = 3$, $b = 3/2$ and $d = 0$. (Notice that $b \neq 2/3$! If you made this mistake, make sure you understand why it is incorrect.) Since $3 > 1 = (3/2)^0$, the third case of the Master Theorem tells us that $T(n) = \Theta\left(n^{\log_{3/2}(3)}\right)$. Although this looks weird, we can have a rational number as the base of a logarithm (in fact, the base of $ln(n)$ is e, an irrational number). It might be helpful to compute the log since it isn't clear how good or bad this complexity is. Notice that $\log_{3/2}(3) \approx 2.71$, so $T(n)$ is approximately $\Theta(n^{2.71})$, but $T(n) \neq \Theta(n^{2.71})$, so resist the urge to place an equals sign between these.
8.89 The complexity of `Mergesort` is $\Theta(n \log n)$ and the complexity of `StoogeSort` is $\Theta\left(n^{\log_{3/2}(3)}\right)$ which is $\Omega(n^{2.7})$. Clearly $\Theta\left(n^{\log_{3/2}(3)}\right)$ grows faster than $\Theta(n \log n)$, so `Mergesort` is faster. *Remember: Faster growth rate means slower algorithm!*
8.90 Let $T(n)$ be the complexity of this algorithm. From the description, it seems pretty clear that $T(n) = 5T(n/3) + cn$. Using the Master Theorem with $a = 5$, $b = 3$, and $d = 1$, we see that $5 > 3^1$, so the third case applies and $T(n) = \Theta\left(n^{\log_3(5)}\right)$, which is approximately $\Theta(n^{1.46})$.
9.4 Nobody in their right mind will choose fruit if cake and ice cream are available, so there are $3 + 8 = 11$ choices. Just kidding. There are really $3 + 8 + 5 = 16$ different choices.

9.7 There are 26 choices for each of the first three characters, and 10 choices for each of the final three characters. Therefore, there are $26^3 \cdot 10^3$ possible license plates.
9.10 Every divisor of n is of the form $p_1^{b_1}p_2^{b_2}\cdots p_k^{b_k}$, where $0 \leq b_1 \leq a_1$, $0 \leq b_2 \leq a_2$, …, $0 \leq b_k \leq a_k$. (We could also write this as $0 \leq b_i \leq a_i$ for $0 \leq i \leq k$.) Therefore there are a_1+1 choices for b_1, a_2+1 choices for b_2, all the way through a_k+1 choices for b_k. Since each of the b_is are independent of each other, the product rule tells us that the number of divisors of n is $(a_1+1)(a_2+1)\cdots(a_k+1)$.
9.11 Unless the p_i are distinct, the b_is are not independent of each other. In other words, if the p_is are distinct, then each different choice of the b_is will produce a different number. But this is not the case if the p_is are not distinct. For instance, if we write $32 = 2^3 2^2$, we can get the factor 4 as $2^2 2^0$, $2^1 2^1$, or $2^0 2^2$. Clearly we would count 4 three times and would obtain the incorrect number of divisors.
9.13 Write $n = \underbrace{1+1+\cdots+1}_{n-1\ +'\text{s}}$. There are two choices for each plus sign–leave it or perform the addition. Each of the 2^{n-1} ways of making choices leads to a different expression, and every expression can be constructed this way. Therefore, there are 2^{n-1} such ways of expressing n.
9.15 This combines the product and sum rules. We now have $10 + 26 = 36$ choices for each character, and there are 5 characters, so the answer is 36^5.
9.16 Each bit can be either 0 or 1, so there are 2^n bit strings of length n.
9.18 $53 \cdot 63^2$; $53 \cdot 63^3$; $53 \cdot 63^{k-1}$.
9.21 It contains at least one repeated digit. The wording of your answer is very important. Your answer should not be "it has some digit twice" since this is vague–do you mean 'exactly twice'? If so, that is incorrect. If you mean 'at least twice', then it is better to be explicit and say it that way or just say 'repeated'. To be clear, we don't know that it contains any digit exactly twice, and we also don't know how many unique digits the number has–it might be 22222222222, but it also might be 98765432101.
9.24 If all the magenta, all the yellow, all the white, 14 of the red and 14 of the blue marbles are drawn, then in among these $8 + 10 + 12 + 14 + 14 = 58$ there are no 15 marbles of the same color. Thus we need 59 marbles in order to insure that there will be 15 marbles of the same color.
9.25 She knows that you are the 25th person in line. If everyone gets 4 tickets, she will get none, but you will get the 4 you want. She can get one or more tickets if one or more people in front of her, including you, get less than 4.
9.28 There are seven possible sums, each one a number in $\{-3, -2, -1, 0, 1, 2, 3\}$. By the Pigeonhole Principle, two of the eight sums must add up to the same number.
9.32 We have $\lceil \frac{16}{5} \rceil = 4$, so some cat has at least four kittens.
9.33

Evaluation of Proof 1: This proof is incomplete. It kind of argues it for 5, not n in general. Even then, the proof is neither clear not complete. For instance, what are the 4 'slots'?

Evaluation of Proof 2: They only prove it for $n = 2$. It needs to be proven for *any* n.

Evaluation of Proof 3: You can't assume somebody had shaken hands with everyone else without some justification. You certainly can't assume it was any particular person (i.e. person n). Similarly, you can't assume the next person has shaken $n-2$ hands without justifying it. The final statement is weird (what does 'fulfills the contradiction' mean?) and needs justification (why is it a problem that the last person shakes no hands?).

9.34 Notice that if someone shakes $n-1$ hands, then nobody shakes 0 hands and vice-verse. Thus, we have two cases. If someone shakes $n-1$ hands, then the n people can shake hands with between 1 and $n-1$ other people. If nobody shakes hands with $n-1$ people, then the n people can shake hands with between 0 and $n-2$ other people. In either case, there are $n-1$ possibilities for the number of hands that the n people can shake. The pigeonhole principle implies that two people shake hands with the same number of people.

Note: You cannot say that the two cases are that someone shakes hands with $n-1$ or someone shakes hands with 0. It may be that *neither* of these is true. The two cases are someone shakes hands with $n-1$ others or nobody does. Alternatively, you could say someone shakes hands with 0 others or nobody does.

9.35 Choose a particular person of the group, say Charlie. He corresponds with sixteen others. By the pigeonhole principle, Charlie must write to at least six of the people about one topic, say topic I. If any pair of these six people corresponds about topic I, then Charlie and this pair do the trick, and we are done. Otherwise, these six correspond amongst themselves only on topics II or III. Choose a particular person from this group of six, say Eric. By the Pigeonhole Principle, there must be three of the five remaining that correspond with Eric about one of the topics, say topic II. If amongst these three there is a pair that corresponds with each other on topic II, then Eric and this pair correspond on topic II, and we are done. Otherwise, these three people only correspond with one another on topic III, and we are done again.

9.39 EAT, ETA, ATE, AET, TAE, and TEA.

9.42 Since there are 15 letters and none of them repeat, there are 15! permutations of the letters in the word UNCOPYRIGHTABLE.

9.44 (a) $5 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 = 25,200$. (b) We condition on the last digit. If the last digit were 1 or 5 then we would have 5 choices for the first digit and 2 for the last digit. Then there are 6 left to choose from for the second, 5 for the third, etc. So this leads to

$$5 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 2 = 7,200$$

possible phone numbers. If the last digit were either 3 or 7, then we would have 4 choices for the first digit and 2 for the last. The rest of the digits have the same number of possibilities as above, so we would have

$$4 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 2 = 5,760$$

possible phone numbers. Thus the total number of phone numbers is

$$7200 + 5760 = 12,960.$$

9.46 Label the letters T_1, A_1, L_1, and L_2. There are 4! permutations of these letters. However, every permutation that has L_1 before L_2 is actually identical to one having L_1 before L_2, so we have double-counted. Therefore, there are $4!/2 = 12$ permutations of the letters in $TALL$.

9.47 $TALL$, $TLAL$, $TLLA$, $ATLL$, $ALTL$, $ALLT$, $LLAT$, $LALT$, $LATL$, $LLTA$, $LTLA$, and $LTAL$. That makes 12 permutations, which is exactly what we said it should be in Exercise 9.46.

9.48 Following similar logic to the previous few examples, since we have one letter that is repeated three times, and a total of 5 letters, the answer is $5!/3! = 20$.

9.49 Ten of them are $AIEEE$, $AEIEE$, $AEEIE$, $AEEEI$, $EAIEE$, $EAEIE$, $EAEEI$, $EEAIE$, $EEAEI$, $EEEAI$. The other ten are identical to these, but with the A and I swapped.

9.52 We can consider $SMITH$ as one block along with the remaining 5 letters A, L, G, O, and R. Thus, we are permuting 6 'letters', all of which are unique. So there are $6! = 720$ possible permutations.

9.55 (a) $5 \cdot 8^6 = 1310720$. (b) $5 \cdot 8^5 \cdot 4 = 655360$. (c) $5 \cdot 8^5 \cdot 4 = 655360$.

9.59 (a) $\dfrac{7\cdot 6\cdot 5\cdot 4\cdot 3}{1\cdot 2\cdot 3\cdot 4\cdot 5}=21$. (b) $\dfrac{12\cdot 11}{1\cdot 2}=66$. (c) $\dfrac{10\cdot 9\cdot 8\cdot 7\cdot 6}{1\cdot 2\cdot 3\cdot 4\cdot 5}=252$.
(d) $\dfrac{200\cdot 199\cdot 198\cdot 197}{1\cdot 2\cdot 3\cdot 4}=64,684,950$. (e) 1.

9.62 (a) $\binom{17}{15}=\binom{17}{2}=\dfrac{17\cdot 16}{1\cdot 2}=136$. (b) $\binom{12}{10}=\binom{12}{2}=\dfrac{12\cdot 11}{1\cdot 2}=66$.
(c) $\binom{10}{6}=\binom{10}{4}=\dfrac{10\cdot 9\cdot 8\cdot 7}{1\cdot 2\cdot 3\cdot 4}=210$. (d) $\binom{200}{196}=\binom{200}{4}=\dfrac{200\cdot 199\cdot 198\cdot 197}{1\cdot 2\cdot 3\cdot 4}=64,684,950$.
(e) $\binom{67}{66}=\binom{67}{1}=67/1=67$.

9.66 $12, 13, 14, 15, 23, 24, 25, 34, 35, 45$.

9.69

Evaluation of Solution 1: This solution does not take into account which woman was selected and which 15 of the original 16 are left, so this is not correct.

Evaluation of Solution 2: This solution has two problems. First, it counts things multiple times. For instance, any selection that contains both Sally and Kim will be counted twice–once when Sally is the first woman selected and again when Kim is selected first. Second, the product rule should have been used instead of the sum rule. Of course, that hardly matters since it would have been wrong anyway.

Evaluation of Solution 3: This solution is correct.

9.71 To count the number of shortest routes from A to B that pass through point O, we count the number of paths from A to O (of which there are $\binom{5}{3}=10$) and the number of paths from O to B (of which there are $\binom{4}{3}=4$). Using the product rule, the desired number of paths is $\binom{5}{3}\binom{4}{3}=10\cdot 4=40$.

9.72

Evaluation of Solution 1: This answer is incorrect since it will count some of the committees multiple times. If you did not come up with an example of something that gets counted multiple times, you should do so to convince yourself that this answer is incorrect.

Evaluation of Solution 2: This solution is incorrect since it does not take into account which man and woman were selected and which 14 of the original 16 are left.

9.73 There are $\binom{16}{5}$ possible committees. Of these, $\binom{9}{5}$ contain only men and $\binom{7}{5}$ contain only women. Clearly these two sets of committees do not overlap. Therefore, the number of committees that contain at least one man and at least one woman is $\binom{16}{5}-\binom{9}{5}-\binom{7}{5}$.

9.74 Because we subtracted the size of both of these from the total number of possible committees. If the sets intersected, we would have subtracted some possibilities twice and the answer would have been incorrect.

9.76

Evaluation of Solution 1: This solution is incorrect since it double counts some of the possibilities.

Evaluation of Solution 2: This solution is incorrect because it does not take into account the requirement that one course from each group must be taken.

9.77

Evaluation of Solution 1: This solution is incorrect since it counts some of the possibilities multiple times.

Evaluation of Solution 2: This solution is incorrect because it does not take into account the requirement that one course from each group must be taken.

9.80 Using 10 bars to separate the meat and 3 stars to represent the slices, we can see that this is exactly the same as the previous two examples. Thus, the solution is $\binom{13}{10} = \binom{13}{3} = 286$.

9.84 We want the number of positive integer solutions to

$$a + b + c + d = 100,$$

which by Theorem 9.82 is

$$\binom{100-1}{4-1} = \binom{99}{3} = 156849.$$

9.87 Observe that $1024 = 2^{10}$. We need a decomposition of the form $2^{10} = 2^a 2^b 2^c$, that is, we need integers solutions to

$$a + b + c = 10, \quad a \geq 0, b \geq 0, c \geq 0.$$

By Corollary 9.85 there are $\binom{10+3-1}{3-1} = \binom{12}{2} = 66$ such solutions.

9.91

$$\begin{aligned}
(2x - y^2)^4 &= \binom{4}{0}(2x)^4 + \binom{4}{1}(2x)^3(-y^2) + \binom{4}{2}(2x)^2(-y^2)^2 + \binom{4}{3}(2x)(-y^2)^3 + \binom{4}{4}(-y^2)^4 \\
&= (2x)^4 + 4(2x)^3(-y^2) + 6(2x)^2(-y^2)^2 + 4(2x)(-y^2)^3 + (-y^2)^4 \\
&= 16x^4 - 32x^3y^2 + 24x^2y^4 - 8xy^6 + y^8
\end{aligned}$$

9.92

$$\begin{aligned}
(\sqrt{3} + \sqrt{5})^4 &= (\sqrt{3})^4 + 4(\sqrt{3})^3(\sqrt{5}) + 6(\sqrt{3})^2(\sqrt{5})^2 + 4(\sqrt{3})(\sqrt{5})^3 + (\sqrt{5})^4 \\
&= 9 + 12\sqrt{15} + 90 + 20\sqrt{15} + 25 \\
&= 124 + 32\sqrt{15}
\end{aligned}$$

9.94 Using a little algebra and the binomial theorem, we can see that

$$\sum_{k=1}^{n} \binom{n}{k} 3^k = \sum_{k=0}^{n} \binom{n}{k} 3^k - 1 = \sum_{k=0}^{n} \binom{n}{k} 1^{n-k} 3^k - 1 = (1+3)^n - 1 = 4^n - 1.$$

9.98 Let A be the set of camels eating wheat and B be the set of camels eating barley. We know that $|A| = 46$, $|B| = 57$, and $|A \cup B| = 100 - 10 = 90$. We want $|A \cap B|$. By Theorem 9.96 (solving it for $|A \cap B|$),

$$|A \cap B| = |A| + |B| - |A \cup B| = 46 + 57 - 90 = 13.$$

9.101 Observe that $1000 = 2^3 5^3$, and thus from the 1000 integers we must weed out those that have a factor of 2 or of 5 in their prime factorization. If A_2 denotes the set of those integers divisible by 2 in the interval $[1, 1000]$ then clearly $|A_2| = \left\lfloor \frac{1000}{2} \right\rfloor = 500$. Similarly, if A_5 denotes the set of those integers divisible by 5 then $|A_5| = \left\lfloor \frac{1000}{5} \right\rfloor = 200$. Also $|A_2 \cap A_5| = \left\lfloor \frac{1000}{10} \right\rfloor = 100$. This means that there are $|A_2 \cup A_5| = 500 + 200 - 100 = 600$ integers in the interval $[1, 1000]$ sharing at least a factor with 1000, thus there are $1000 - 600 = 400$ integers in $[1, 1000]$ that do not share a factor prime factor with 1000.

9.104 Using Theorem 9.102, we know that $28 + 29 + 19 - 14 - 10 - 12 + 8 = 48\%$ watch at least one of these sports. That leaves 52% that don't watch any of them.

9.106 Let C denote the set of people who like candy, I the set of people who like ice cream, and K denote the set of people who like cake. We are given that $|C| = 816$, $|I| = 723$, $|K| = 645$, $|C \cap I| = 562$, $|C \cap K| = 463$, $|I \cap K| = 470$, and $\mathrm{card}\,(C \cap I \cap K) = 310$. By Inclusion-Exclusion we have

$$\begin{aligned}
|C \cup I \cup K| &= |C| + |I| + |K| \\
&\quad -|C \cap I| - |C \cap K| - |I \cap C| \\
&\quad +|C \cap I \cap K| \\
&= 816 + 723 + 645 - 562 - 463 - 470 + 310 \\
&= 999.
\end{aligned}$$

The investigator miscounted, or probably did not report one person who may not have liked any of the three things.

9.108 We can either use inclusion-exclusion for four sets or use a few applications of inclusion-exclusion for two sets. Let's try the latter.

Let A denote the set of those who lost an eye, B denote those who lost an ear, C denote those who lost an arm and D denote those losing a leg. Suppose there are n combatants. Then

$$\begin{aligned}
n &\geq |A \cup B| \\
&= |A| + |B| - |A \cap B| \\
&= .7n + .75n - |A \cap B|,
\end{aligned}$$

$$\begin{aligned}
n &\geq |C \cup D| \\
&= |C| + |D| - |C \cap D| \\
&= .8n + .85n - |C \cap D|.
\end{aligned}$$

This gives

$$|A \cap B| \geq .45n,$$

$$|C \cap D| \geq .65n.$$

This means that

$$\begin{aligned}
n &\geq |(A \cap B) \cup (C \cap D)| \\
&= |A \cap B| + |C \cap D| - |A \cap B \cap C \cap D| \\
&\geq .45n + .65n - |A \cap B \cap C \cap D|,
\end{aligned}$$

whence

$$|A \cap B \cap C \cap D| \geq .45 + .65n - n = .1n.$$

This means that at least 10% of the combatants lost all four members.

GNU Free Documentation License

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index