5.2 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 5.31. *An algorithm is* recursive *if it calls itself.*

Examples of recursion that you may have already seen include *binary search*, *Quicksort*, and *merge sort*.

★**Question 5.32.** 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 5.33. The following is a simple iterative algorithm to compute $n!$.

```
int factorial(int n) {
    if(n<=0) {
        return 1;
    } else {
        int fact = 1;
        for(int i=2;i<=n;i++) {
            fact = fact*i;
        }
        return fact;
    }
}
```

We can create a slightly simpler algorithm by using recursion. Notice that

$$\begin{array}{rcll} 0! & = & 1 & \\ 1! & = & 1 & = \quad 1 \times 0! \\ 2! & = & 2 \times 1 & = \quad 2 \times 1! \\ 3! & = & 3 \times 2 \times 1 & = \quad 3 \times 2! \\ 4! & = & 4 \times 3 \times 2 \times 1 & = \quad 4 \times 3! \\ 5! & = & 5 \times 4 \times 3 \times 2 \times 1 & = \quad 5 \times 4! \\ & & \text{and in general, when } n > 1, & \\ n! & = & n \times (n-1) \times \cdots \times 2 \times 1 & = \quad n \times (n-1)! \end{array}$$

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 $n!$ when $n \geq 0$.

```
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 5.34. Let's take a closer look at the `factorial` algorithm from Example 5.33. If $n \leq 0$, `factorial` does not make a recursive call. Thus, it has a *base case*. When $n > 0$, it is clearly making a recursive call, so it has *inductive cases*. When a recursive call is made to `factorial`, the argument is smaller, so it is approaching a base case (i.e. making *progress*).

★**Question 5.35.** Consider the `ferzle` algorithm from Question 5.32 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 5.36. Prove that the recursive `factorial(n)` algorithm from Example 5.33 returns $n!$ for all $n \geq 0$.

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 5.37. 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)`
⋮	⋮	⋮

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 5.38.** Prove that the recursive `countdown(n)` algorithm from Example 5.37 works correctly. (Hint: Use induction.)

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 5.39. 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.

Here is an 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 5.40. Prove that the *recursive* `binarySearch` algorithm from Example 5.39 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. □

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 5.41. 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 5.42.** 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 5.37. 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 5.43. We want to compare the memory usage of the iterative and recursive algorithms to compute $n!$ that we gave in Example 5.33. The iterative one uses memory to store four numbers: n, $fact$, i, and return value. The recursive one 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.

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 5.44. 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 5.45.** 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 5.42. 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.