An Introduction to Using and Abusing Recursion

Charles Cusack

Google results for “recursion”

Recursion in a Nutshell

- A function is called *recursive* if it calls itself.
- If a 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 function terminates without calling itself.
- **Example:** Compute $\sum_{i=1}^{n} i = 1 + 2 + \cdots + n$ using recursion.

```
int sum1ToN(int n) {
    if (n<=1)
        return 1;
    else
        return n + sum1ToN(n-1);
}
```

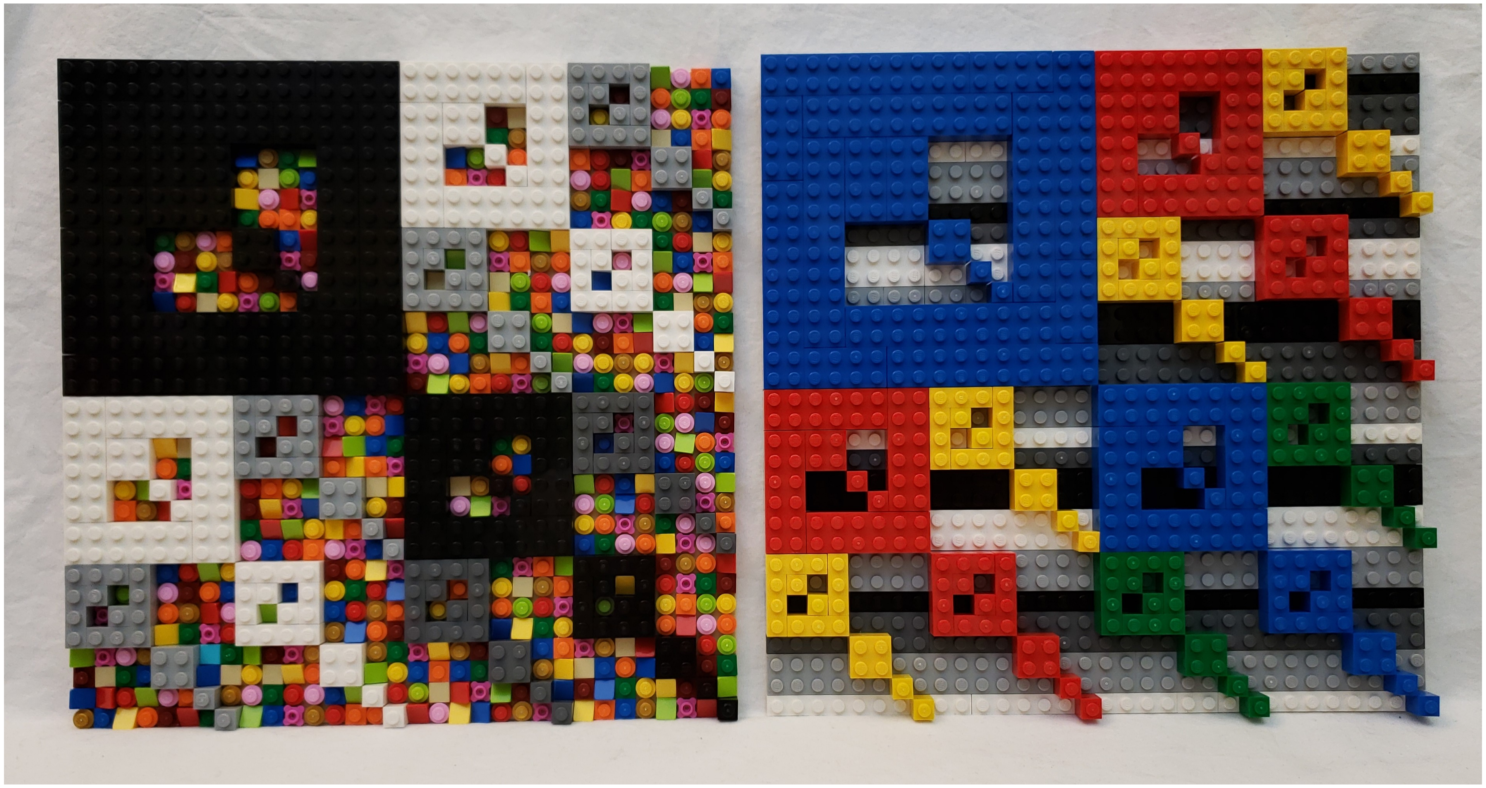
- What ensures that the function `sum1ToN` terminates?

Where is Recursion Seen/Used?

- We occasionally see recursion in the “real” world:
  - Russian Matryoshka (nested dolls)
  - Two almost parallel mirrors
  - A video camera pointed at the monitor
  - Fractals
- In computer science, it is useful when working with certain data structures and algorithms.
- Examples include
  - Computing $n!$
  - Binary Search
  - Merge Sort, Quick Sort and many other *divide-and-conquer* algorithms
  - Algorithms on binary trees.

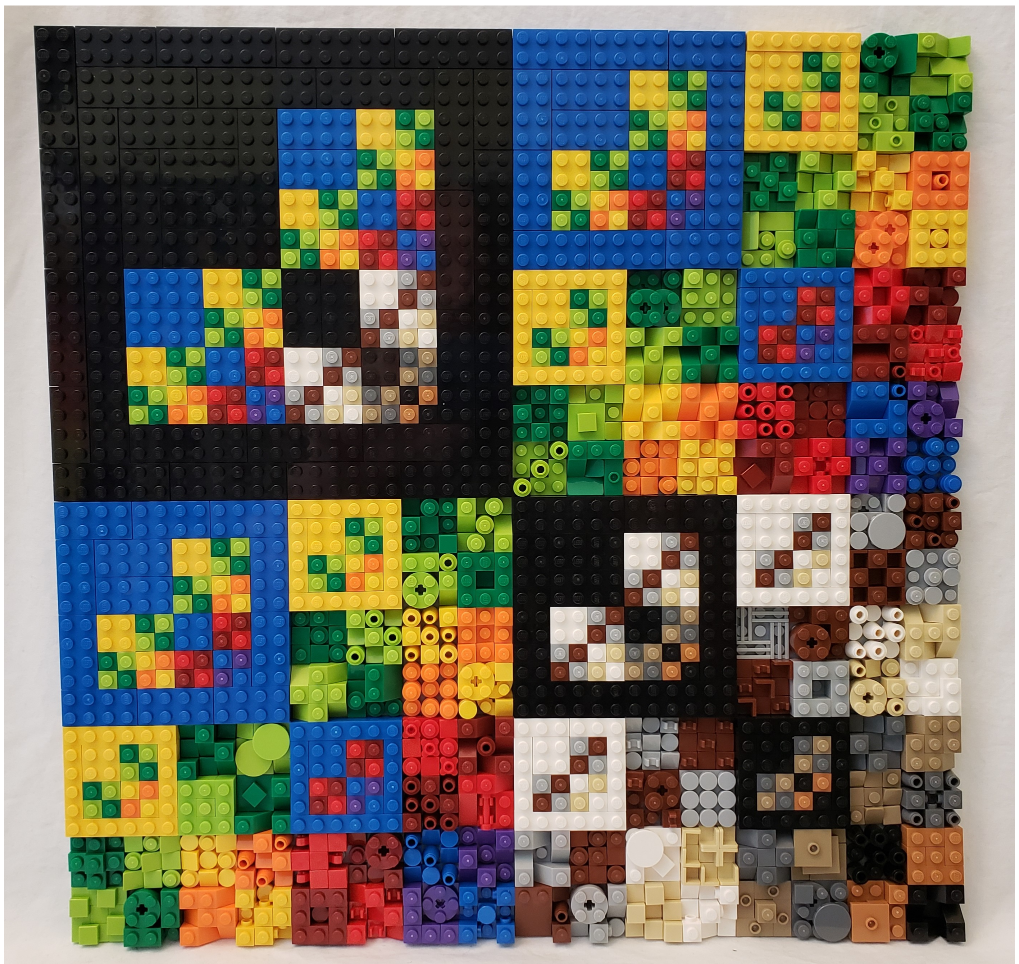
Recursion in Art

- Two *partial fractal Latin squares*.
- *Grayscale Fractal on Colored Background* and *GRBY Fractal over GRAY Stripes*, 2019, by Charles Cusack.



Recursion in Art

- *Textured Fractal Latin Square*, 2018, by Charles Cusack.



Recursion Example

- Suppose we want to implement a function **CountDown(n)** which outputs the integers from n down to 1, where $n > 0$.
- **Example:** The call **CountDown(5)** results in output:

  5 4 3 2 1

- The easiest way to implement this is probably a loop:

```
void CountDown(int n) {
    for (i=n ; i>0 ; i--)
        print(i);
}
```

- This is (hopefully) nothing new or earth shattering.
- Let's consider how to do it with recursion.

Recursive CountDown(n)

- How can we think of this function recursively?
- **CountDown(n)** outputs n followed by the numbers from $n - 1$ down to 1.
- But **CountDown($n - 1$)** outputs numbers $n - 1$ down to 1.
- Thus, the output from **CountDown(n)** is n followed by the output from **CountDown($n - 1$)**.
- Based on this, we can write the function recursively as follows:

```
void CountDown(int n) {
        print(n);
        CountDown(n-1);
}
```

- We are close, but something is wrong here. What is it?

Recursive CountDown(n) Error

- The problem is, CountDown never stops:

Execute	Output	Then Execute
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)**
$\vdots$	$\vdots$	$\vdots$

- The problem is not with the recursion, but with our logic. We are supposed to stop printing when $n = 1$, but we didn't take that into account.
- To fix this, we modify it so that a call to **CountDown(0)** produces no output and does not call **CountDown** again.
- Calls to **CountDown(**n**)** when $n < 0$ should also produce no output.
- We can take care of both of these problems at once.

Recursive CountDown(n) Fixed

- The following version of **CountDown(n)** is correct:

```
void CountDown(int n) {
        if(n>0) {
              print(n);
              CountDown(n-1);
        }
}
```

- Now **CountDown(n)** does exactly what we want when $n > 0$.
- When $n \leq 0$, **CountDown(n)** does nothing (as it should).

Making Recursion Work

- In order for a recursive function to work properly, it must be defined so that it will eventually terminate.
- A proper recursive definition has both of the following:
  - *Base case(s)*: One or more cases which are solved non-recursively.
    - In other words, when a function gets to the base case, it does not call itself again.
    - This is also called a *stopping case* or *terminating condition*.
  - *Inductive case(s)*: A recursive rule for all cases except the base case(s).
    - Inductive cases should always progress toward the base case.

CountDown Example Continued

- Recall the function **CountDown(n)**

```
void CountDown(int n) {
    if(n>0) {
        print(n);
        CountDown(n-1);
    }
}
```

- *Base case*:
  When $n \leq 0$, **CountDown(n)** does nothing.

- *Inductive case*:
  When $n > 0$, **CountDown(n)** outputs n and calls **CountDown($n - 1$)**.
  - Notice, the recursive call is closer to the base case.

Example: Factorial

- Recall that $n! = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n$.
- **Example:** $7! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 7 = 5040$.
- We can also define $n!$ recursively:

$$n! = \begin{cases} 1 & \text{when } n = 0, 1 \\ n \cdot (n-1)! & \text{when } n > 1 \end{cases}$$

- **Example:**

$$\begin{aligned} 1! &= 1 \\ 2! &= 2 \cdot 1! = 2 \cdot 1 = 2 \\ 3! &= 3 \cdot 2! = 3 \cdot 2 = 6 \\ 4! &= 4 \cdot 3! = 4 \cdot 6 = 24 \\ 5! &= 5 \cdot 4! = 5 \cdot 24 = 120 \end{aligned}$$

Computing Factorial

- We can compute $n!$ iteratively:

```
int factorial (int n) {
    int result=1;
    while (n>1) {
        result = result * n;
        n--;
    }
    return result;
}
```

- Or recursively:

```
int factorial(int n) {
    if(n<=1)
        return 1;
    else
        return n * factorial(n-1);
}
```

Factorial: Comparing Implementations

Iterative

```
int factorial (int n) {
    int result=1;
    while (n>1) {
        result = result*n;
        n--;
    }
    return result;
}
```

Recursive

```
int factorial(int n) {
  if(n<=1)
     return 1;
  else
     return n*factorial(n-1);
}
```

- Notice that algorithms are about equally complicated.
- Both take on the order of n operations to compute $n!$.
- Is one clearly better?
- What about memory usage?

Recursion and Memory

- Without digressing too much, it is important to understand a little bit about memory use of recursive functions.
- When a function is called, a chunk of memory called an *activation record* or *stack frame* is allocated for use by the function.
- For simplicity, we will assume each activation record contains
  - memory for each parameter
  - memory for each local variable
  - memory for the return value
- In order to support function calls, the run-time system treats memory as a stack of activation records
- Thus, a recursive function that calls itself n times must allocate n activation records.
- You can learn more about this in a programming languages course.

Example:

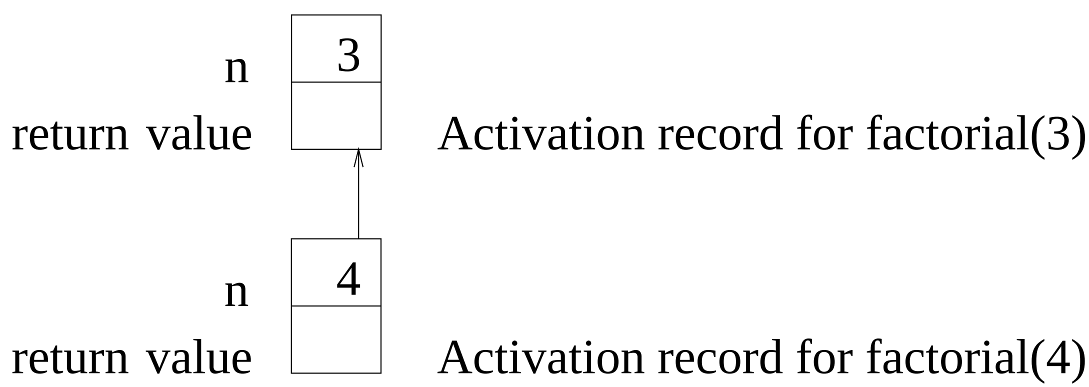
- **factorial(4)**

n	4
return value	

Activation record for factorial(4)

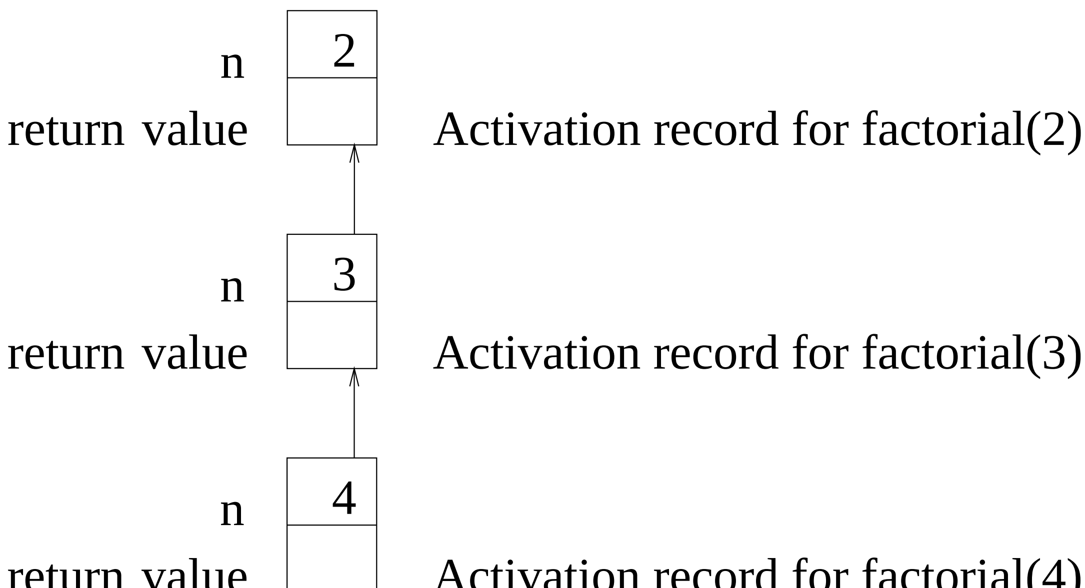
Example:

- **factorial(4):** call to factorial(3)



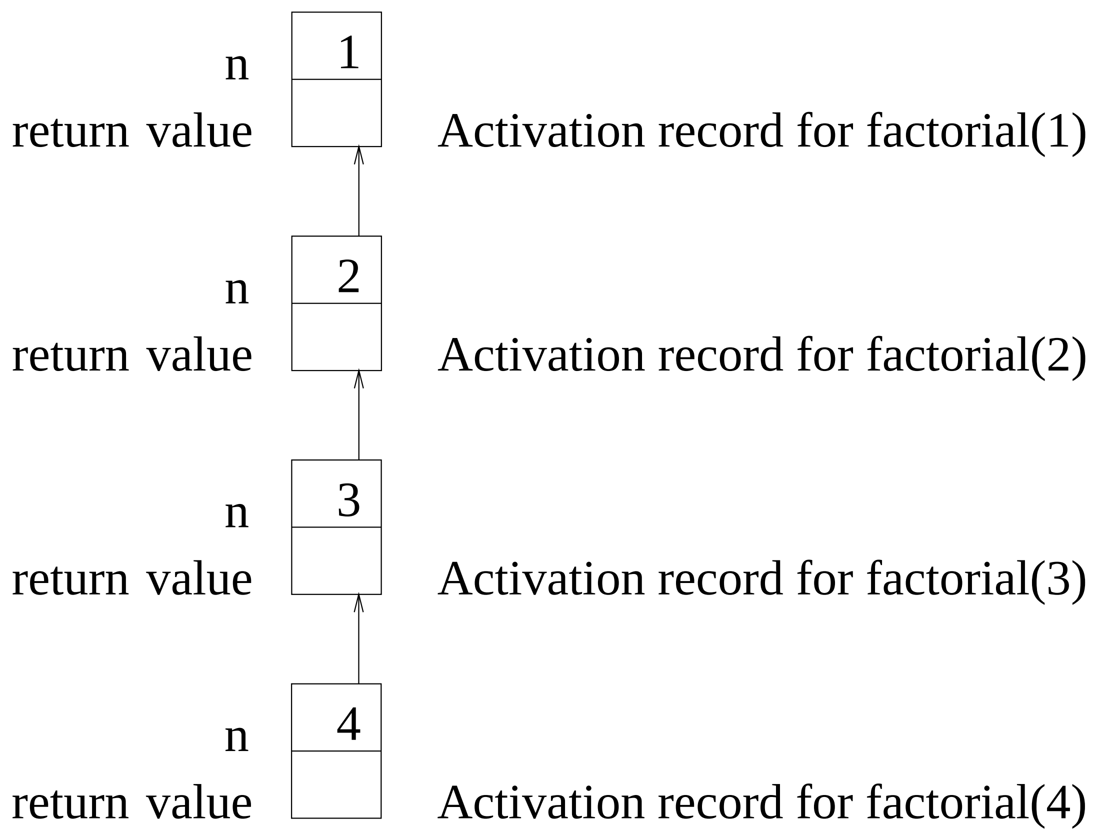
Example:

- **factorial(4):** call to factorial(2)



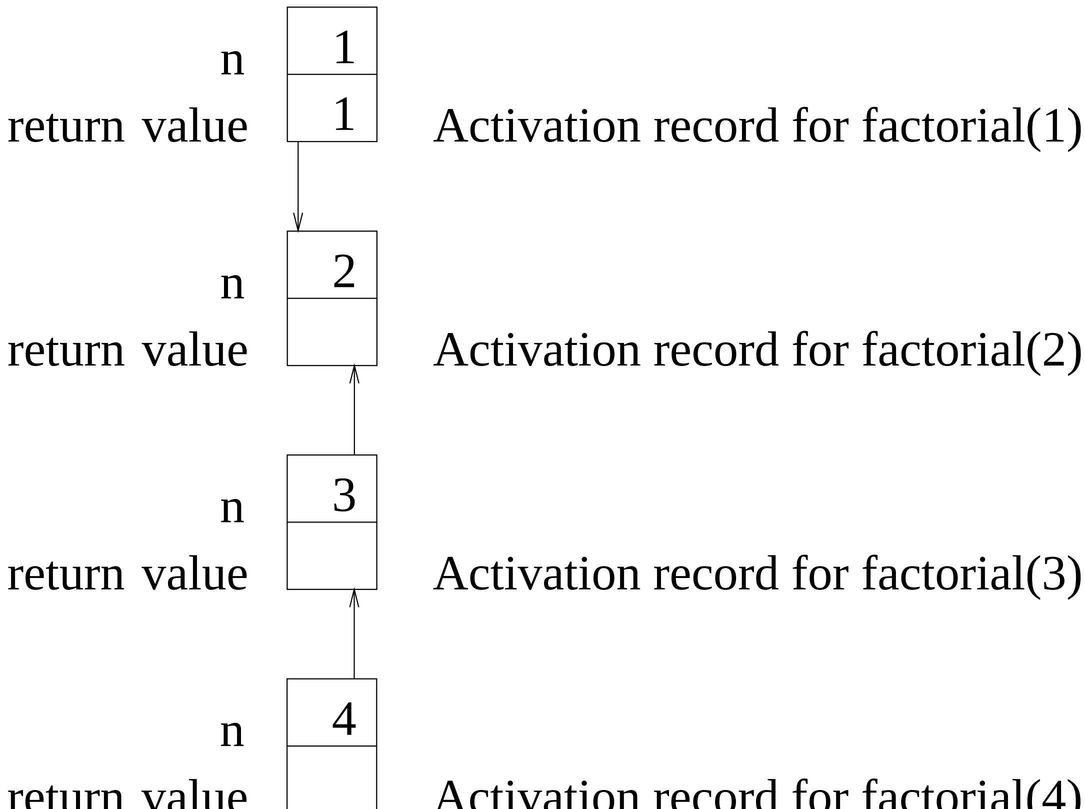
Example:

- **factorial(4):** call to factorial(1)



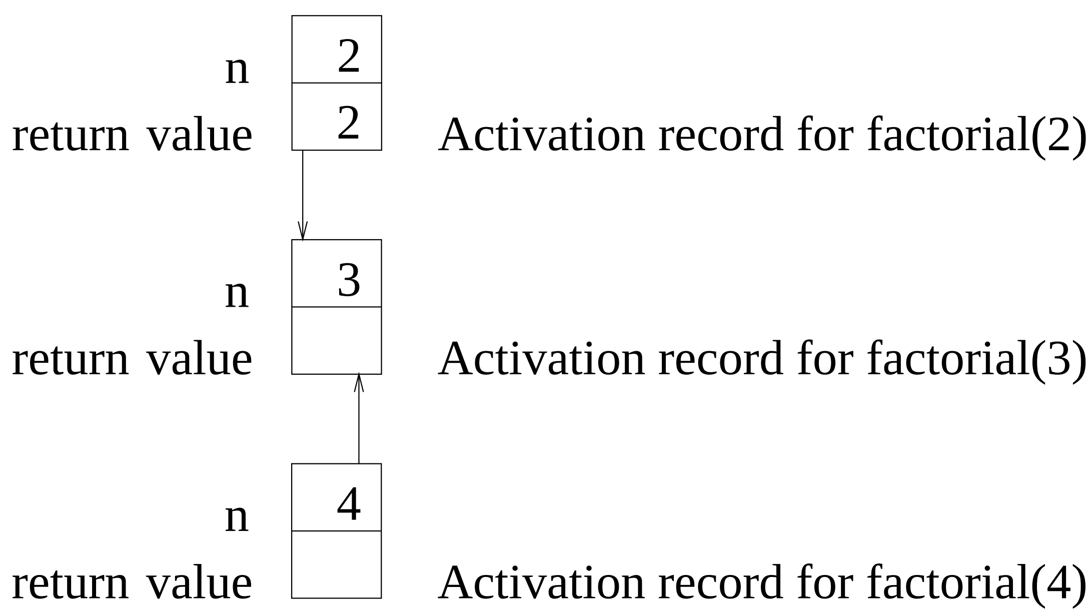
Example:

- **factorial(4):** factorial(1) is the base case, so it returns 1.



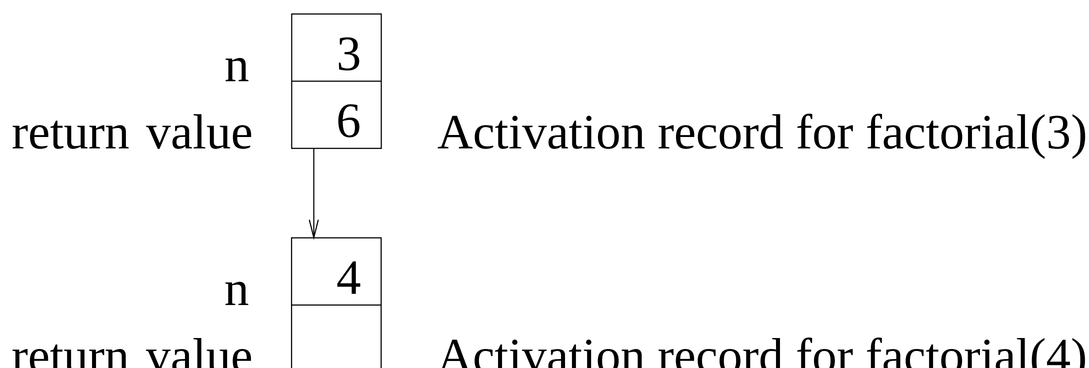
Example:

- **factorial(4):** factorial(2) returns 2.



Example:

- **factorial(4):** factorial(3) returns 6.

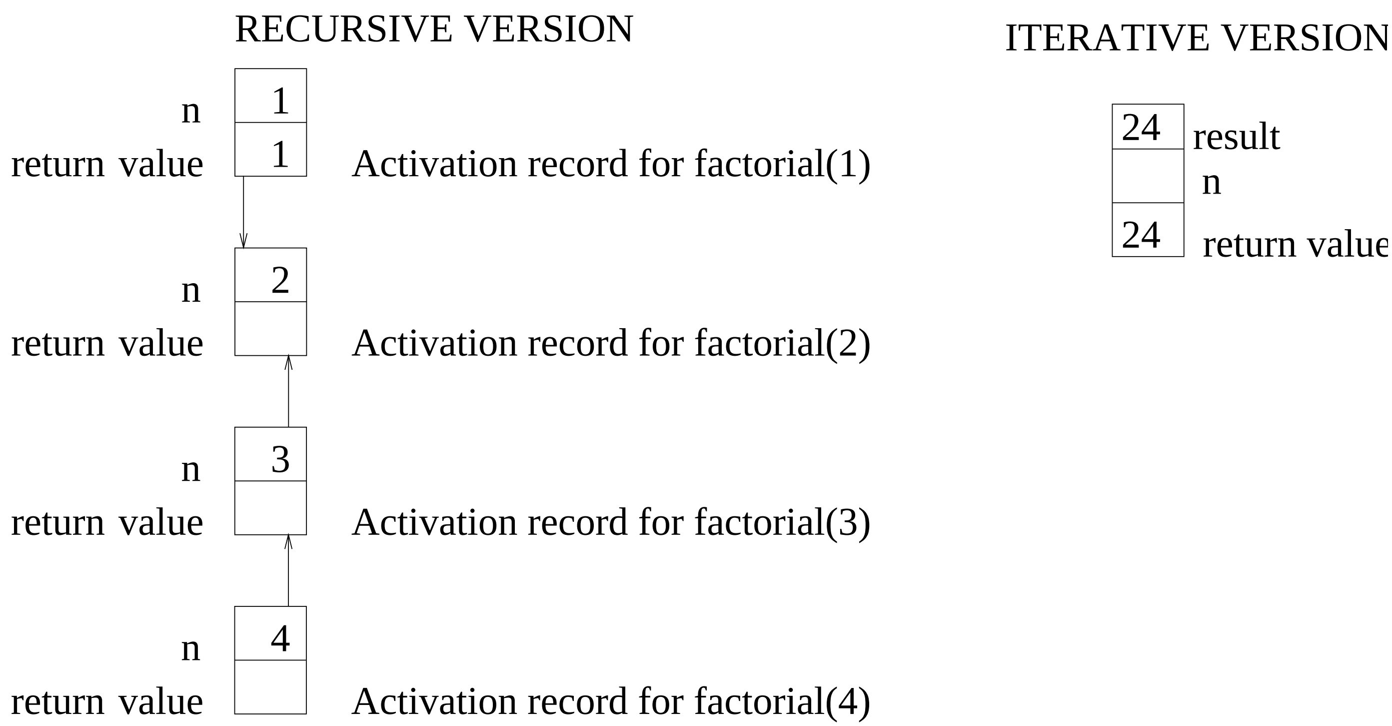


Example:

- **factorial(4):** returns 24.
- This was the original function call, so the execution is finished.

n	4	
return value	24	Activation record for factorial(4)

Memory Usage



- For input n, the recursive implementation needs $2n$ integers
- The iterative implementation needs only 3 integers.
- In general, recursive algorithms use more memory than equivalent iterative algorithms.

Limits of Recursion

- Clearly infinite recursion is bad:

```
int infiniteRecursion(int n) {
    if (n==0)   return 1;
    else        return infiniteRecursion(n);
}
```

  Since n never reaches zero, the function is called and records are pushed onto the stack until the system runs out of memory.

- Even if recursion is not infinite it can still run too deep since computers only have a finite amount of memory.
- Languages/platforms often have some defined limit on how deep recursive calls can go.
- On the other hand, it is rare that you run into this limit with properly implemented recursive functions.
- One notable exception: some small embedded systems with very limited memory can barely handle multiple function calls and do not even support recursion.

Recursive Problem Solving

- In general, we can solve a problem with recursion if we can:
  - Find one or more simple cases of the problem that can be solved directly.
  - Find a way to break up the problem into smaller instances of the *same* problem.
  - Find a way to combine the smaller solutions.
- Classic examples of this include
  - *Binary Search* (We'll look at this next)
  - Divide-and-conquer algorithms like *Quick Sort* and *Merge Sort*
  - Solving the *Towers of Hanoi*
  - Working with *binary trees*
- Take a data structures and/or algorithms course to learn more about these and other applications of recursion.

Searching with Recursion: Binary Search

- If you need to search a list for some value, typically you need to perform a **linear search** through the list.
- If the list is sorted, there is a *much* quicker algorithm.
- The **binary search** algorithm finds an item v on a sorted list by repeatedly discarding half of the list.
- 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.
- Although this can be implemented with iteration, it is more natural to think recursively.

Recursive Binary Search

```
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;
    }
}
```

- Requires on the order of $\log_2 n$ operations as opposed to n for linear search. (Take a discrete math and/or algorithms course to see why.)

Fibonacci Numbers

- The Fibonacci numbers are a sequence of integers that are of interest to mathematicians, computer scientists, artists, etc.
- They are given by the following recursive definition:

$$\mathrm{F}(n) = \begin{cases} 0 & \text{if } n{=}0 \\ 1 & \text{if } n{=}1 \\ \mathrm{F}(n-1) + \mathrm{F}(n-2) & \text{if } n > 1 \end{cases}$$

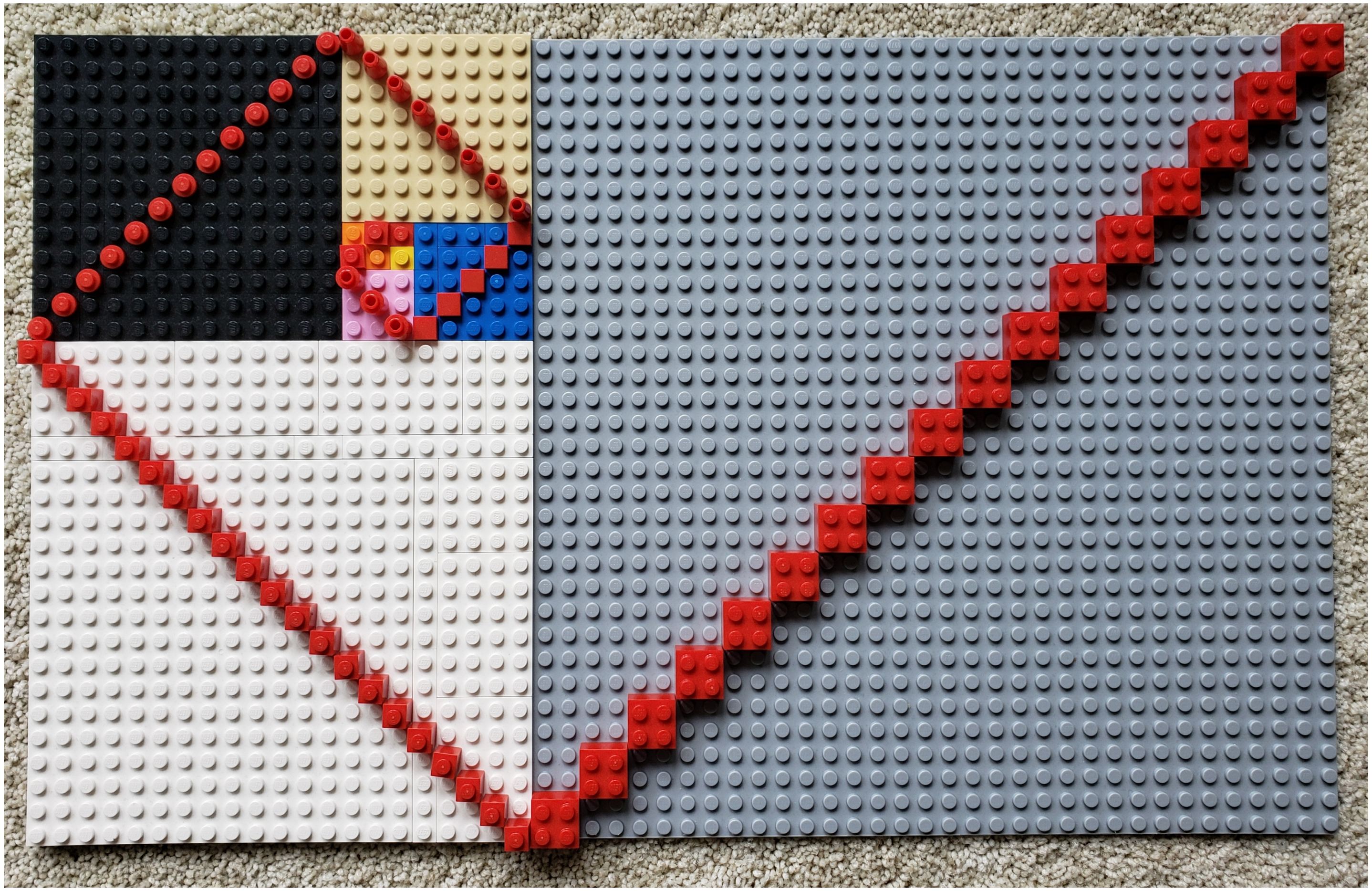
- The first few are:

```
F(0) = 0
F(1) = 1
F(2) = F(0) + F(1) = 0 + 1 =  1
F(3) = F(1) + F(2) = 1 + 1 =  2
F(4) = F(2) + F(3) = 1 + 2 =  3
F(5) = F(3) + F(4) = 2 + 3 =  5
F(6) = F(4) + F(5) = 3 + 5 =  8
```

- We will consider both an iterative and a recursive algorithm to calculate **`F(n)`**

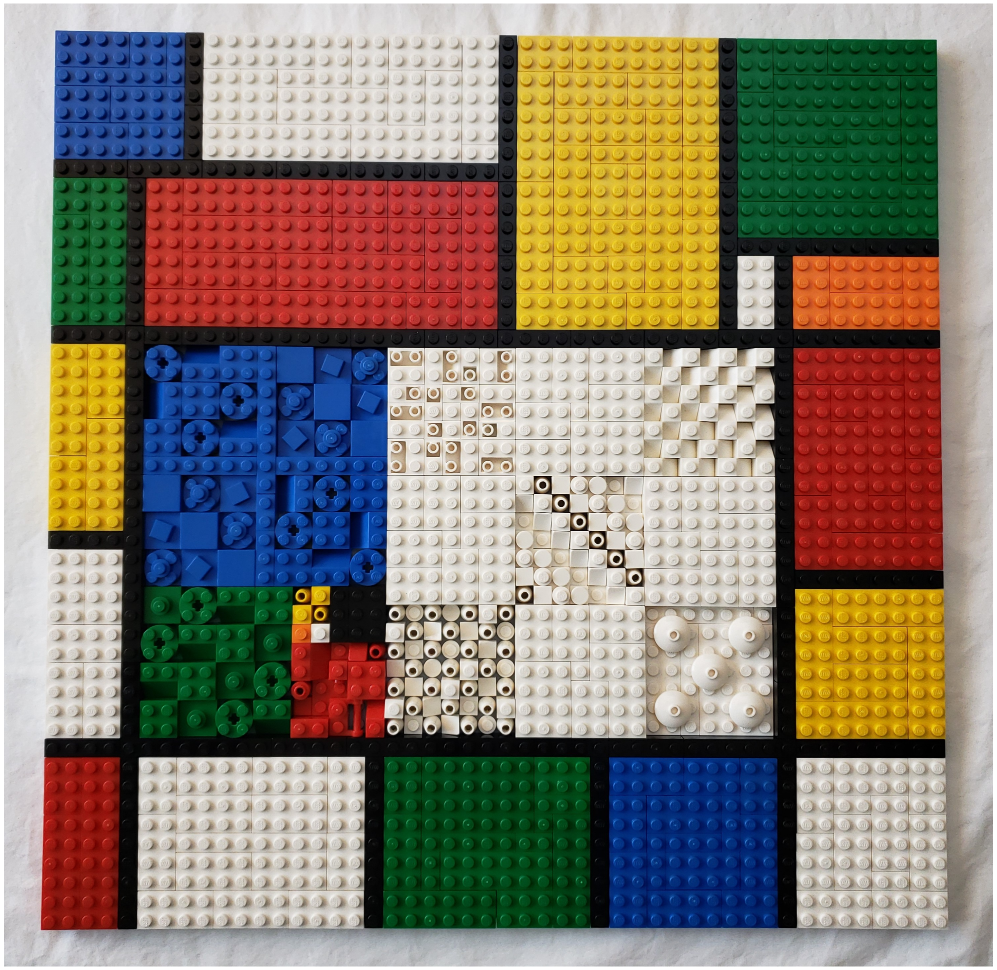
Fibonacci Numbers in Art

- The *golden rectangle/spiral*, based on Fibonacci numbers, is often used in art.



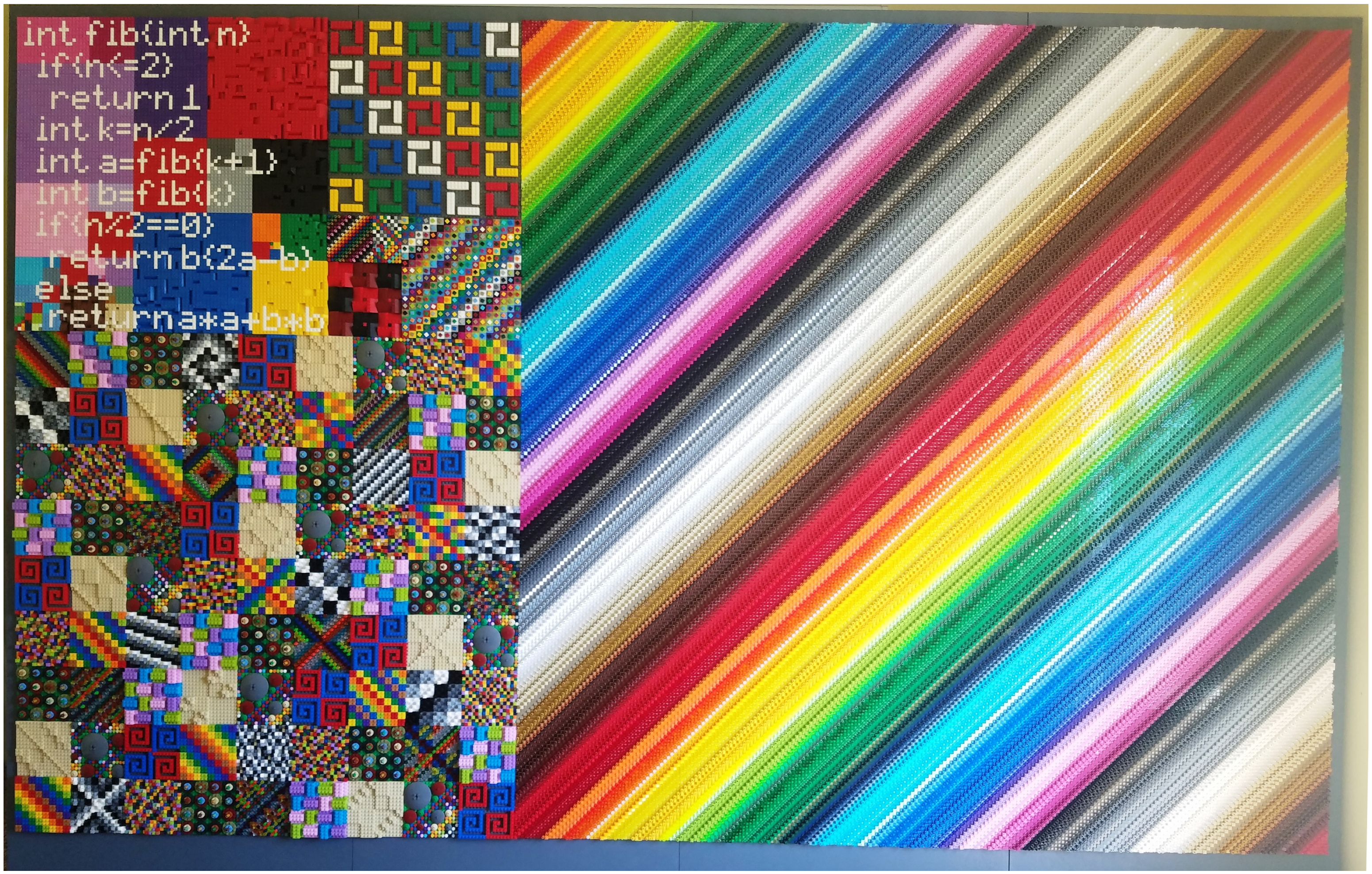
Fibonacci Numbers in Art

- *Mondriacci*, 2018, by Charles Cusack.



Fibonacci Numbers in Art

- *Increasing Asquareness*, ArtPrize 2016, by Charles Cusack.



Iterative Fibonacci Function

- Start with $F(0) = 0$ and $F(1) = 1$, then compute the next number based on the previous two until we reach $F(n)$.
- Since $F(n) = F(n-1) + F(n-2)$, we must keep track of the previous two numbers as we go.

```
int FibI(int n) {
    if (n <= 1) return(n);
    else {
        int fib=0, fibm1=1, fibm2=0, index=1;
        while (index < n) {
            fib = fibm1 + fibm2;
            fibm2 = fibm1;
            fibm1 = fib;
            index = index + 1;
        }
        return(fib);
}}
```

Recursive Fibonacci Function

- While the iterative solution started from $F(0)$ and $F(1)$ and worked forward, the recursive solution starts from n and works backwards.
- We use $F(n) = F(n-1) + F(n-2)$ as before.

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

- As you can see, the recursive function is much simpler to program.
- **True or False:** The recursive algorithm to compute the nth Fibonacci number is an example of an algorithm that is both elegant and efficient.

Iterative versus Recursive Fibonacci

n	FibI	FibR
1	1.4	0.8
2	1.1	0.4
3	1.1	0.5
4	0.9	0.7
5	1.4	0.9
6	1.0	1.2
7	1.0	1.6
8	0.8	3.5
9	1.0	10.3
10	1.0	7.5
11	1.2	9.1
12	1.1	2.9
13	1.1	5.6
14	1.3	6.1
15	0.9	7.6
16	0.9	23.5
17	1.0	15.6
18	0.9	25.1

- Let's compare the running times (in microseconds).
- So far they are both pretty fast, although the recursive one seems to be taking a little bit longer as n increases.
- But how bad can it get? It's probably just a little overhead due to recursion.

Iterative versus Recursive Fibonacci

n	FibI	FibR	n	FibI	FibR
1	1.4	0.8	19	2.0	55.7
2	1.1	0.4	20	1.0	63.8
3	1.1	0.5	21	2.4	173.0
4	0.9	0.7	22	2.3	255.4
5	1.4	0.9	23	1.0	119.1
6	1.0	1.2	24	0.9	192.4
7	1.0	1.6	25	6.6	311.5
8	0.8	3.5	26	1.4	529.0
9	1.0	10.3	27	1.5	880.0
10	1.0	7.5	28	1.4	1417.0
11	1.2	9.1	29	1.5	2432.8
12	1.1	2.9	30	1.7	3626.4
13	1.1	5.6	31	1.7	7230.0
14	1.3	6.1	32	2.2	9696.3
15	0.9	7.6	33	4.0	17492.1
16	0.9	23.5	34	2.4	26544.4
17	1.0	15.6	35	1.6	40849.3
18	0.9	25.1	36	1.7	85165.4

- It is looking like this might be more than just a little recursive overhead.
- But let's see some more data before we jump to a conclusion.

Iterative versus Recursive Fibonacci

- O.K., maybe `FibR` is a really bad algorithm. But why?

n	FibI	FibR	n	FibI	FibR	n	FibI	FibR
1	1.4	0.8	19	2.0	55.7	37	4.5	106568.8
2	1.1	0.4	20	1.0	63.8	38	1.5	172959.0
3	1.1	0.5	21	2.4	173.0	39	1.5	282319.0
4	0.9	0.7	22	2.3	255.4	40	1.7	447233.6
5	1.4	0.9	23	1.0	119.1	41	1.5	725519.9
6	1.0	1.2	24	0.9	192.4	42	1.8	1182831.5
7	1.0	1.6	25	6.6	311.5	43	1.6	1885098.2
8	0.8	3.5	26	1.4	529.0	44	1.9	3076539.2
9	1.0	10.3	27	1.5	880.0	45	1.4	4922548.0
10	1.0	7.5	28	1.4	1417.0	46	7.3	8006058.3
11	1.2	9.1	29	1.5	2432.8	47	1.5	12873996.6
12	1.1	2.9	30	1.7	3626.4	48	1.6	20800990.7
13	1.1	5.6	31	1.7	7230.0	49	1.9	33745830.4
14	1.3	6.1	32	2.2	9696.3	50	1.9	55031994.2
15	0.9	7.6	33	4.0	17492.1	51	1.9	84096872.3
16	0.9	23.5	34	2.4	26544.4	52	5.2	135878807.6
17	1.0	15.6	35	1.6	40849.3	53	1.7	218251652.5
18	0.9	25.1	36	1.7	85165.4	54	1.7	355260801.8

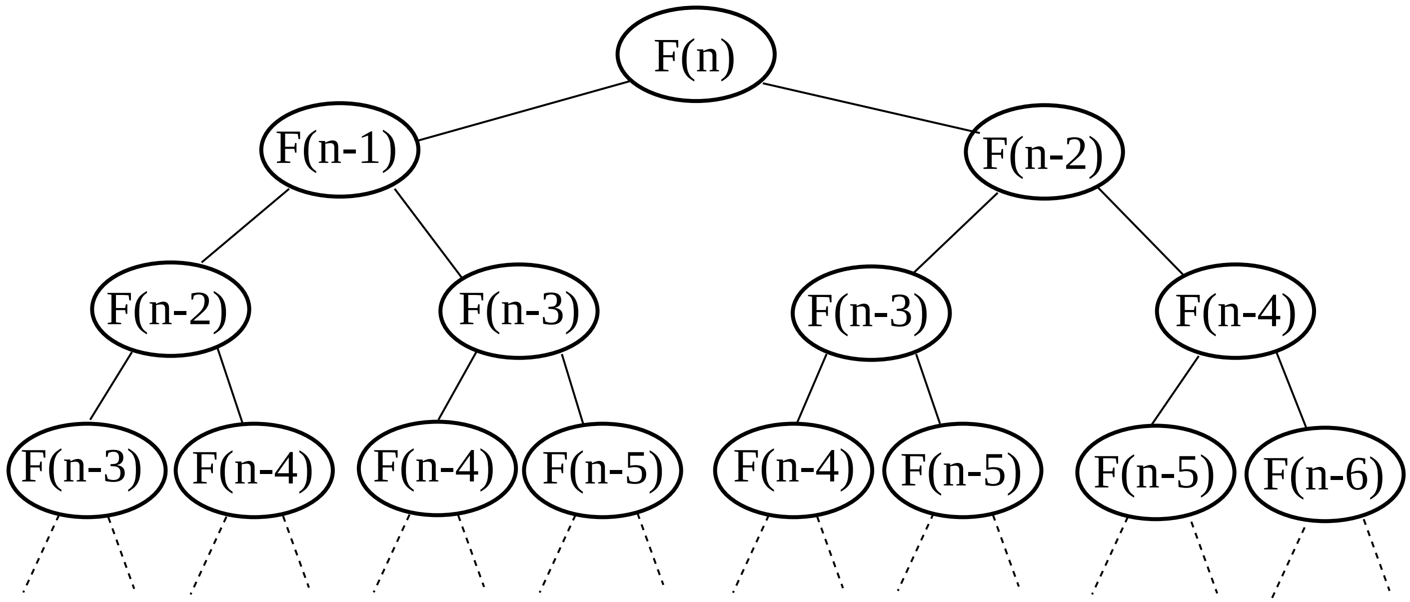
Recursive Fibonacci Problem

n	F(n)	FibR Time	Ratio
37	24157817	106568.8	226.69
38	39088169	172959.0	226.00
39	63245986	282319.0	224.02
40	102334155	447233.6	228.82
41	165580141	725519.9	228.22
42	267914296	1182831.5	226.50
43	433494437	1885098.2	229.96
44	701408733	3076539.2	227.99
45	1134903170	4922548.0	230.55
46	1836311903	8006058.3	229.37
47	2971215073	12873996.6	230.79
48	4807526976	20800990.7	231.12
49	7778742049	33745830.4	230.51
50	12586269025	55031994.2	228.71
51	20365011074	84096872.3	242.16
52	32951280099	135878807.6	242.50
53	53316291173	218251652.5	244.29
54	86267571272	355260801.8	242.83

- Notice that the ratio of $F(n)$ and the time it takes to compute $F(n)$ is approximately constant (or increases very slowly).
- In other words, the time it takes to compute $F(n)$ is proportional to $F(n)$.
- Since $F(n)$ grows exponentially (take a discrete math course for more details), the running time of `FibR(n)` is exponential.

Recursive Fibonacci Problem

- The recursive calls in `FibR(n)` look something like the following



  where the leaves are all $F(0)$ and $F(1)$.

- Recall that $F(0) = 0$ and $F(1) = 1$.
- Thus, `FibR(n)` is computing $F(n)$ by adding $F(n)$ 1s and about that many 0s. That does not seem like an efficient way to compute $F(n)$!

Recursive Fibonacci Solution

- We could certainly make `FibR` faster if we added $F(2) = 1$ as a base case.
  - That way we don't add a bunch of 0s which is really stupid.
  - But that would only give us a small amount of speedup.
- If you want to see how to fix this algorithm, take an algorithms course and learn about *dynamic programming*, *memoization*, and *space-time tradeoffs*.
- Briefly, it can be fixed by using an array to store computed values of $F(n)$ and looking up values that have already been computed rather than computing them again.
- The fixed algorithm performs about as well as the iterative one.

Common Recursion Errors

Forgetting or having incomplete base cases

- **Example:** This function goes into infinite recursion if given a negative number for N.

```
void Sum1toN(int N)
{
    if (N == 0)
        return(0);
    else
        return(N + Sum1toN(N-1));
}
```

Common Recursion Errors

Getting things backwards

- **Example:** One of these functions prints from 1 up to n, the other from n down to 1. Which is which?

```
void Count1(int n) {
    if (n > 0) {
        Count1(n-1);
        print(n);
    }
}
```

```
void Count2(int n) {
    if (n > 0) {
        print(n);
        Count2(n-1);
    }
}
```

Recursion: Advantages

- Recursion mimics the way we think about some problems.
  - For example, binary search is very similar to the way we search through the index of a book (sort of).
- Recursive solutions can be very intuitive to program.
- Often recursive functions to solve problems can be much shorter than iterative (non-recursive) functions. This can make the code easier to understand, modify, and/or debug.
- Many of 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 thought of in terms of recursive functions. (e.g. Quick Sort and Merge Sort)

Recursion: Disadvantages

- Each time one function calls another, the computer's operating system must take care of a number of things:
  - Recording how to re-start the calling function later on.
  - Passing the parameters from the calling function to the called function (often by pushing the parameters onto a stack).
  - Setting up space for the called function's local variables.
  - Recording where the calling function's local variables are stored.
- Doing all this requires time and memory.
- Thus a function which makes many recursive calls can require a lot of extra time and memory—more than a non-recursive solution might.
- Sometimes an obvious recursive solution to a problem can be inherently inefficient (e.g. `FibR(n)`).

Recursion: Conclusion

- A recursive function is one that invokes another instance of itself.
- Recursive functions have *base cases* and *recursive cases.*
- Base cases solve the problem directly.
- Recursive cases need to get closer to base cases.
- Recursion can often provide a more elegant solution than iteration.
- Each instance of a function has its own set of local variables and parameters.
- Recursive solutions are often less efficient, in terms of time and space, than an equivalent iterative solution.
- Some problems are difficult to solve without recursion (particularly when the data structure is defined recursively).