More Recursion

In this lecture, we will go over more cases for recursion

Your future in CS

I used to include this on my slides, but since these slides have changed - going to just leave it up here for every notebook. I get a lot of questions about more programming courses, the concentrations, and minors in computer science. Here is a brief reminder.

CS 165 – Next Course In Sequence, also consider CS 220 (math and stats especially)

- CO Jobs Report 2021 – 77% of *all* new jobs in Colorado require programming
- 60% of all STEM jobs requires *advanced* (200-300 level)
- 31% of all Bachelor of Arts degree titled jobs also required coding skills
- 2016 Report found on average jobs that require coding skills paid $22,000 more

- Concentrations in CS:
    - Computer science has a number of concentrations.
        - General concentration is the most flexible, and even allows students to double major or minor pretty easily.
        - Software Engineering
        - Computing Systems
        - Human Centered Computing
        - Networks and Security
        - Artificial Intelligence
        - Computer Science Education.
    - Minors:
        - Minor in Computer Science - choose your own adventure minor
        - Minor in Machine Learning - popular with stats/math, and engineering
        - Minor in Bioinformatics - Biology + Computer Science

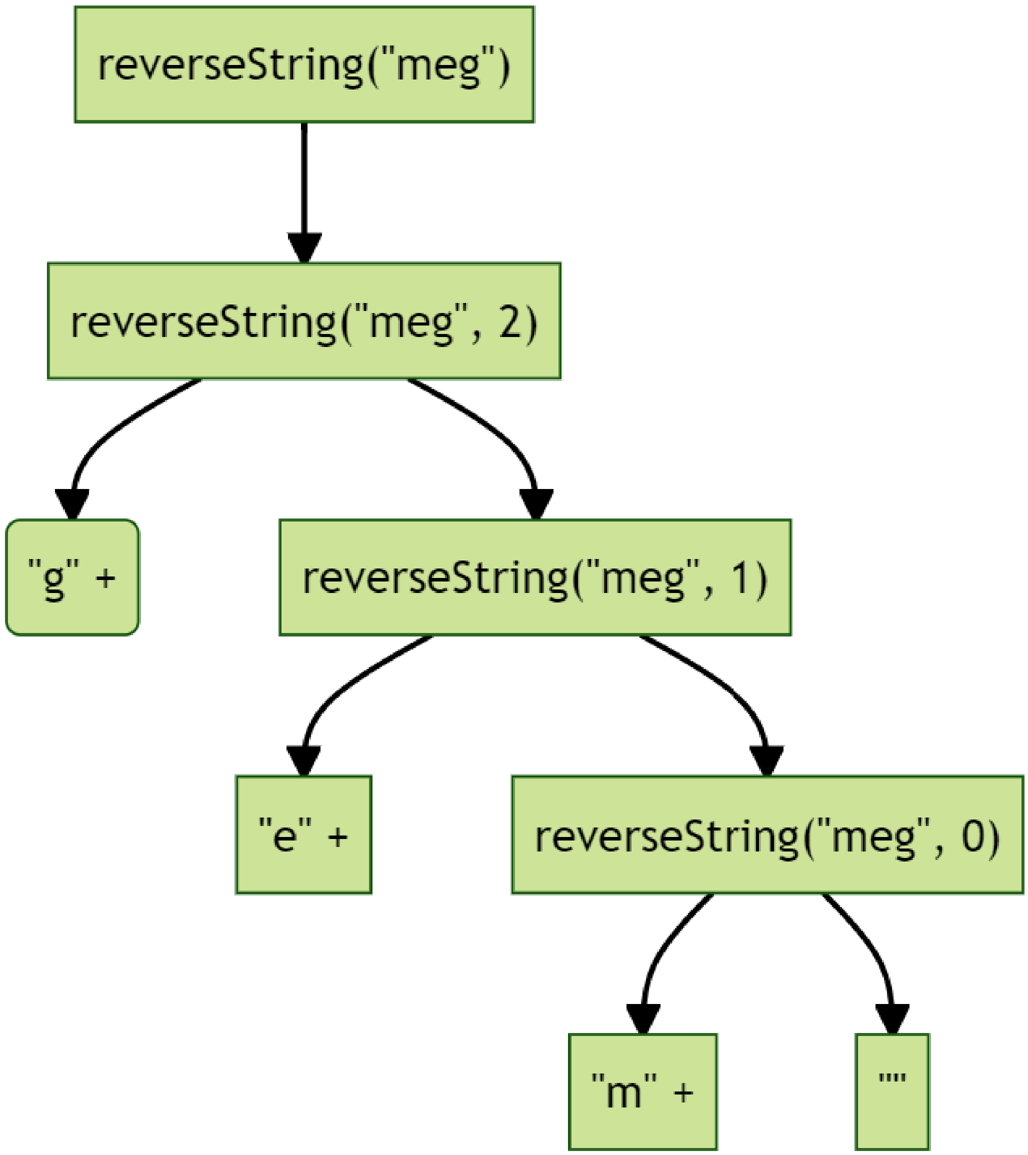
Quick Review

- Recursion is using method calls as a way to loop.
- Can help represent harder problems in a more elegant manner
- Can help handle problems that the loop depth is unknown
    - Array of Arrays
- It helps to draw out a tree to track method calls

```
public static String reverseString(String str) {
    return reverseString(str, str.length()-1);
```

```java
}

public static String reverseString(String str, int index) {
    if(index < 0) return "";
    return str.charAt(index) + reverseString(str, index-1);
}
System.out.println(reverseString("gem"));
```



Warmup Activity

- Write a recursive factorial method
  - $f(n) = f(n) * f(n-1)$
- Additionally, draw out the **tree** of the recursion

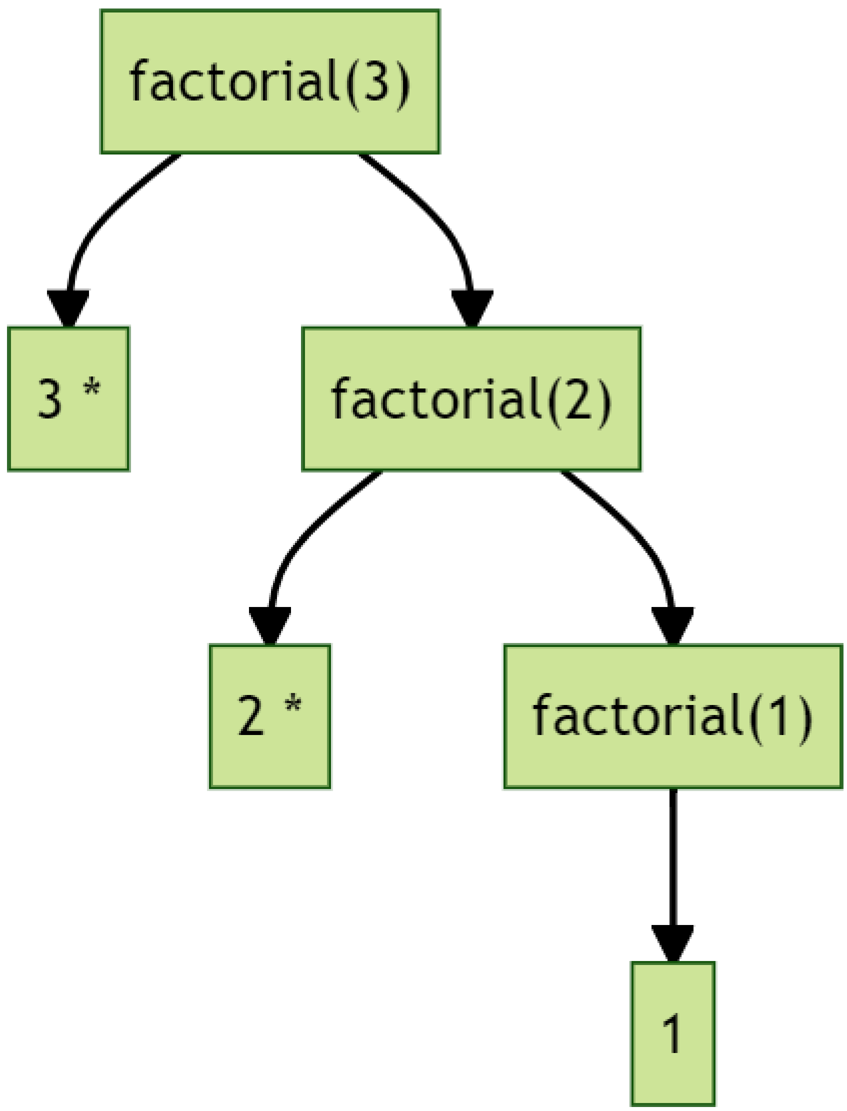
In [34]:
```java
public static int factorial(int n) {
    if(n < 2) return 1;
```

```
    return n * factorial(n-1);
}
```

In [36]:
```
System.out.println("Testing 5: " + factorial(5));
System.out.println("Testing 5: " + factorial(6));
System.out.println("Testing 5: " + factorial(7));
```

```
Testing 5: 120
Testing 5: 720
Testing 5: 5040
```

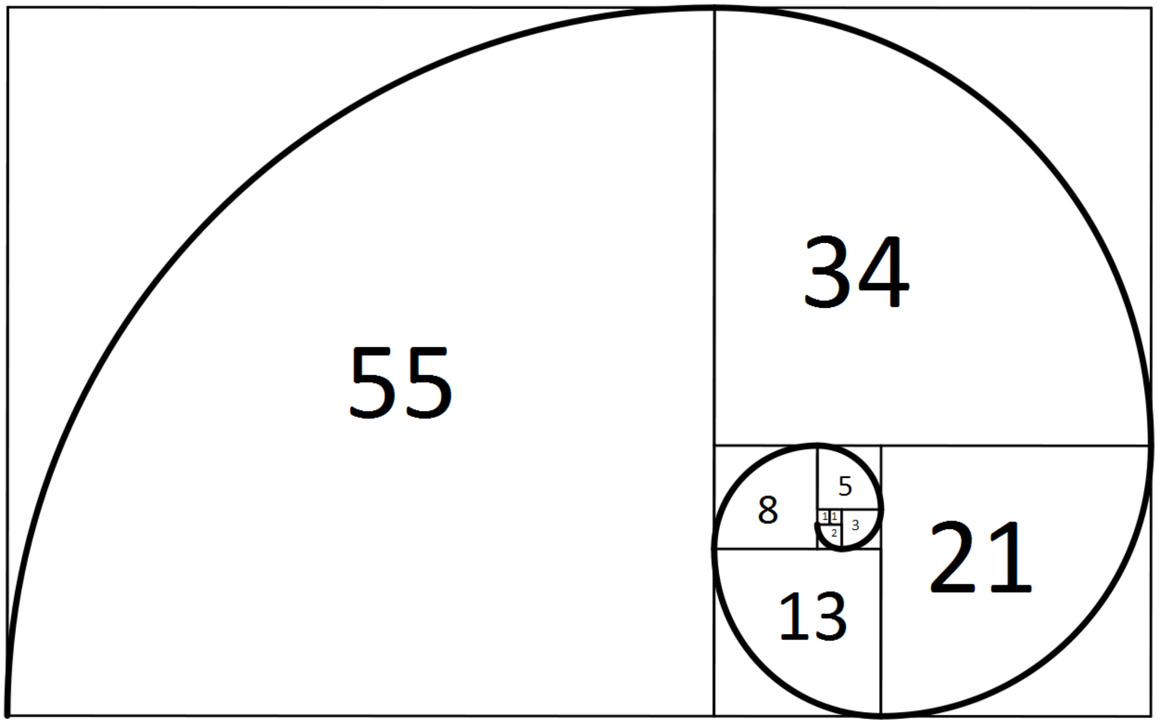
Tree for factorial



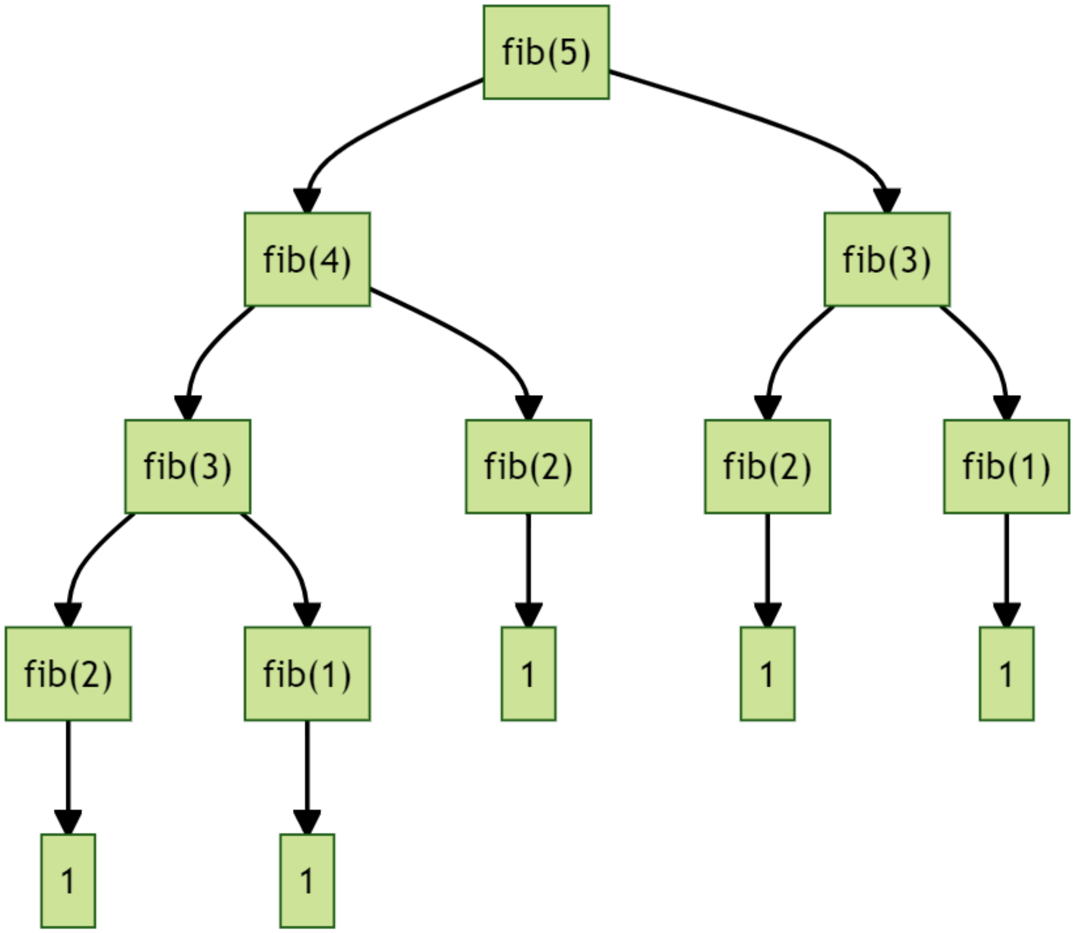
Multiple Branching Trees

- Most of our cases, the tree only had one branch that continued to branch
- The other branch was always the solution
- What if the tree branched out at both sides?

Fibonacci Sequence



- Seen throughout science, especially nature.
- the sum is equal to the previous two
- $fib(1) = 1$
- $fib(2) = 1$
- $fib(3) = fib(2) + fib(1) = 2$
- $fib(4) = fib(3) + fib(2) = 2 + 1$
- $fib(5) = fib(4) + fib(3) = 3 + 2$
- 1 1 2 3 5 8 13 21 ...
- What if we wanted $fib(n)$?
  - $fib(n) = fib(n-1) + fib(n-2)$
  - This looks naturally recursive...
  - The tree would be as follows:



Student Practice

- Write a method that calculates the fibonacci value based on n
- for example:
    - fib(5) would return 5
    - fib(6) would return 8
    - fib(7) would return 13
        - and so on
        - Both fib(2) and fib(1) return 1, fib(0) or lower returns 0.

In [37]:

```
public static long fib(int n) {
    if(n < 1) return 0;
    if(n == 1 || n == 2) return 1;
    long answer = fib(n-1) + fib(n-2);
    return answer;
}

// just an extra method to help view the full sequence
public static void fibShowRecursive(int n) {
    for(int i = 1; i <= n; i++) {
        System.out.print(fib(i) + " ");
    }
    System.out.println();
}
```

In [18]:

```
fibShowRecursive(20);
```

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765
```

Just because you can, doesn't mean you should

- Fibonacci is often used as a recursion example
- However, it is a *bad* example of recursion
  - Yes, it is elegant and lines up with the math
  - But, the implementation causes the 'stack' to explode exponentially
    - why? because for every one recursive call, there are two additional calls
    - As such to compute fib(5) we ended up with 9 method calls
    - To compute fib(6) we end up with 14 recursive calls!
- Let's see a way we can do it with a loop
  - Assuming we don't want to keep previous results

In [28]:

```
public static long fibLoop(int n) {
    long prev = 0;
    long next = 1;
    for(int i = 1; i < n; i++) {
        long sum = prev+next;
        prev = next;
        next = sum;
    }
    return next;
}

public static void fibShowLoop(int n) {
    for(int i = 1; i <= n; i++) {
        System.out.print(fibLoop(i) + " ");
    }
    System.out.println();
}
```

Now, let's run them side by side

In [30]:

```
fibShowLoop(10);
fibShowRecursive(10);
```

```
1 1 2 3 5 8 13 21 34 55
1 1 2 3 5 8 13 21 34 55
```

Not bad, let's increase the number

In [32]:

```
import java.time.Instant;

Instant start = java.time.Instant.now();
fibShowLoop(50);
Instant end = java.time.Instant.now();
System.out.println("Loop Done: " + java.time.Duration.between(start, end).toMillis());


start = java.time.Instant.now();
fibShowRecursive(50);
```

```
end = java.time.Instant.now();
System.out.println("Recursion Done: " + java.time.Duration.between(start, end).toMilli
```

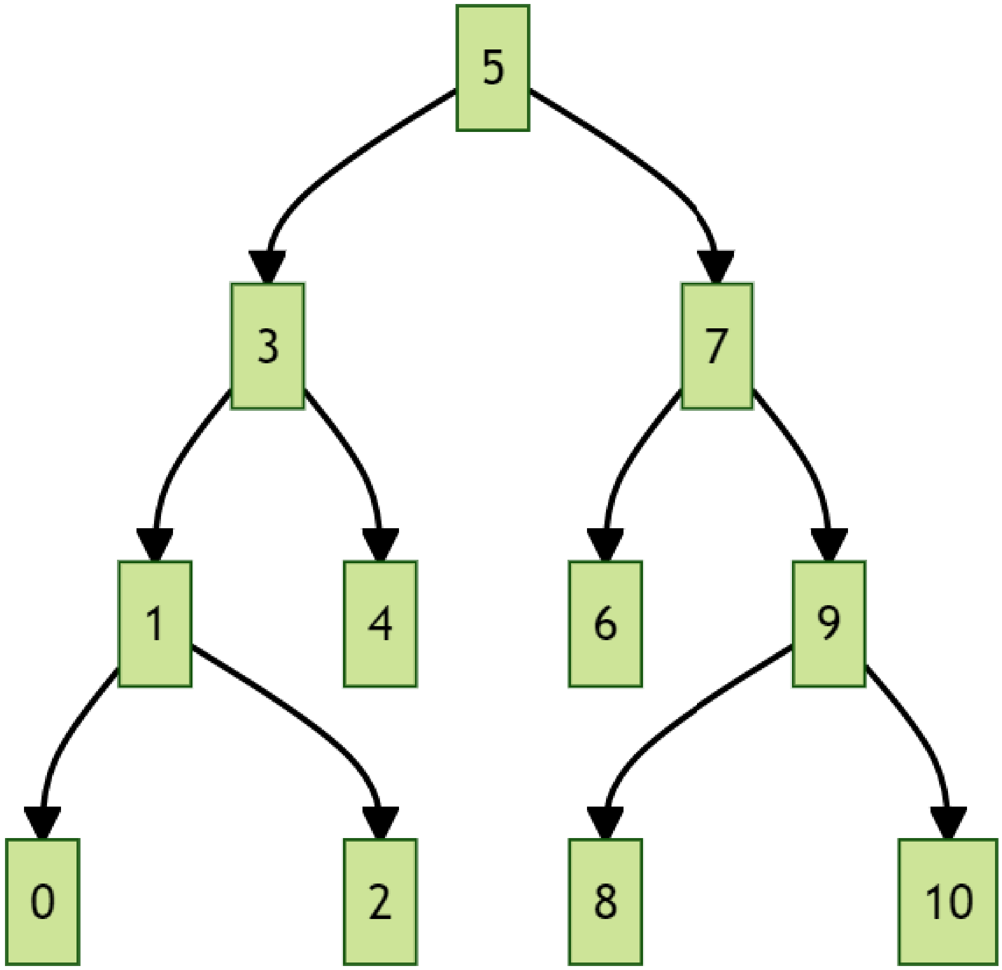
```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657
46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578 5702887 922746
5 14930352 24157817 39088169 63245986 102334155 165580141 267914296 433494437 7014087
33 1134903170 1836311903 2971215073 4807526976 7778742049 12586269025
Loop Done: 103
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657
46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578 5702887 922746
5 14930352 24157817 39088169 63245986 102334155 165580141 267914296 433494437 7014087
33 1134903170 1836311903 2971215073 4807526976 7778742049 12586269025
Recursion Done: 120183
```

Holy rusted metal batman!

103 milliseconds compared to 120,183 milliseconds!

When is it good to use recursion

- When you only have limited paths to follow
- When you don't know your loop depth
- When your data is *already* setup like a tree!
- For example:
  - Let's take a number sequence
  - [0, 1, 2 , 3, 4, 5, 6, 7, 8, 9, 10]
  - This sequence is in order
  - That means I could also structure the 'number' line as follows:



- To get from 5 to 1, it only takes 2 movements, instead of the typical 4 movements! We explore tree structures more more in CS 165.
- However, knowing this, we also know a quick way to **search** for a number
- Let's say we are looking for 7
  - We can say is the number less than, equal to, or greater than 5? Greater
  - Is the larger less than, equal to, or greater than 7? Equal -- FOUND
- This is a binary search, and inherently recursive

In [43]:
```
public static int binarySearch(int[] arr, int key) {
   return binarySearch(arr, key, 0, arr.length-1);
}
public static int binarySearch(int arr[], int key, int first, int last){
   int mid = (first + last) / 2;
   if(first > last) return -1; // not found
   if (arr[mid] == key ) return mid; // index found!
   if (arr[mid] < key ){
      first = mid + 1;
      return binarySearch(arr, key, first, last);
   }
   last = mid -1;
   return binarySearch(arr, key, first, last);
```

```java
}


int arr[] = {10,20,30,40,50};
int key = 20;
int loc = binarySearch(arr,key);
if(loc > 0) System.out.printf("Value found at %d, and the value is %d%n", loc, arr[loc
```

```
Value found at 1, and the value is 20
```

Overview

- We will explore this binary search concept more next lecture
- Recursion is a *very* powerful tool, especially in cases:
  - we don't know the depth of the loops
  - we are already setup in a tree structure
  - we only know the 'next' step in the process (but no more)
  - You will come back to it in both CS 220 and CS 165 - so don't worry if you don't have it mastered just yet!