Topic 7.0: Recursion

Learning Goals (Week 8):

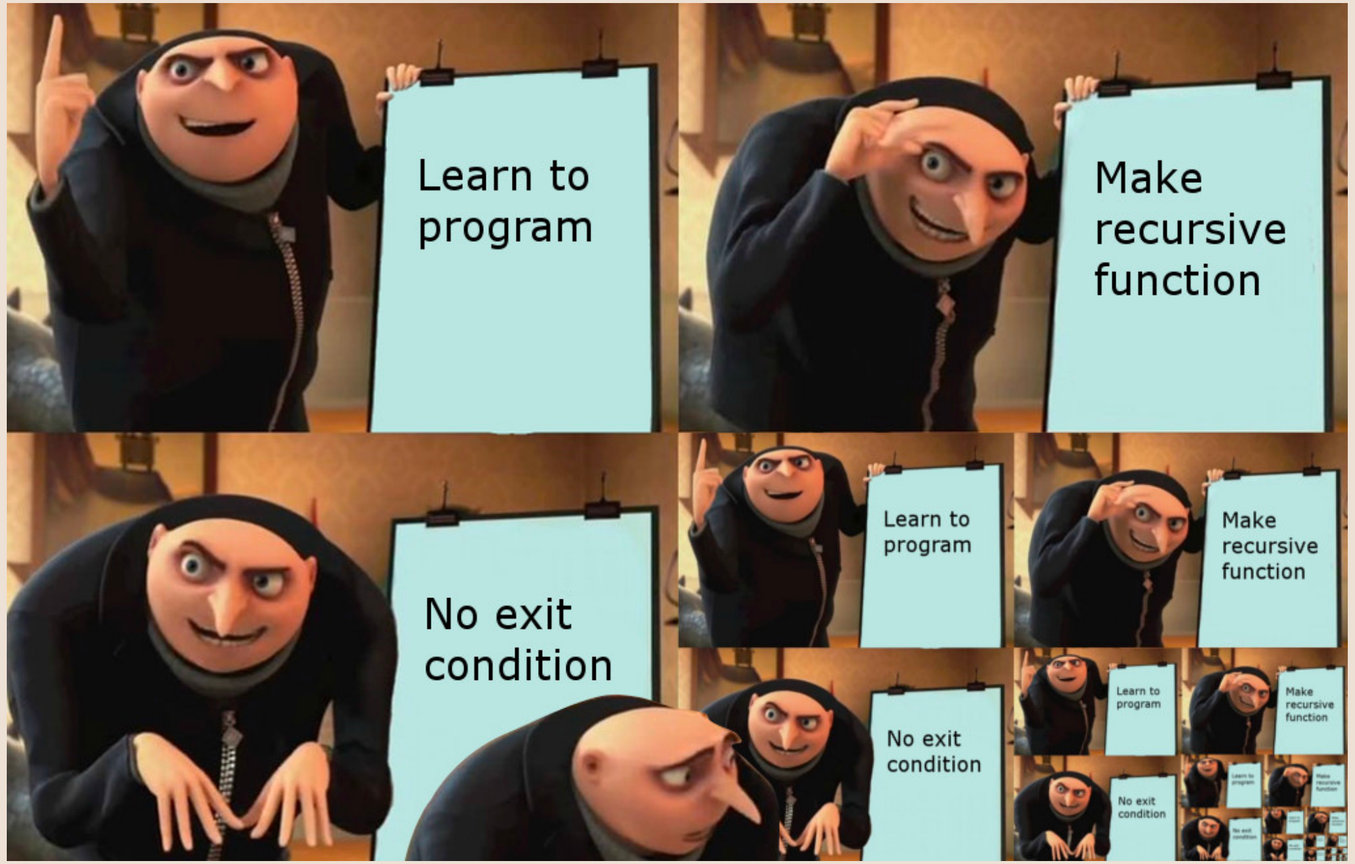
- Create and implement recursive solutions to simple problems such as simple mathematical calculations and list traversals.
- Write a recursive solution to a problem with and without a helper function.
- Identify and explain the base case and recursive step components of a recursive algorithm.

Recursion

- But first:
  - How would you explain recursion to a 6 year old?
    - Explain it to someone 1 year younger than me and tell them to explain it to one person younger then them, etc
    - Continue until a 7 year old is explaining recursion to a 6 year old
  - **SOLVED**! (this is recursion)

Recursion

medium



reddit

“

Imagine you're trying to clean a messy room. You start by picking up a book and opening it, only to find a smaller, messier room inside. So, you step into that room to clean it up, but inside, you find another book with an even smaller, messier room. This keeps going, with you finding smaller and messier rooms within rooms, each time thinking, "This time, I'll definitely finish cleaning!" But just like in recursion, you need a stopping point, or else you'll be cleaning rooms within rooms forever.

chatGPT “Explain Recursion to me in a funny way”

Recursive Problem Solving

- Taking a big problem and solving it by solving smaller subsets of the problem first
- This is surprisingly easy to implement in code:
  - A method can call itself

Traditional Example (Not naturally recursive but we can do it....)

- Consider the following problem:
  - We want to compute a factorial for some number n (n!)
  - How do we do this?
    - 1 * 2 * 3 * ... * (n - 2) * (n -1) * n
    - Kind of looks like a loop, right?

```
public static long fact(int n) {
  long nfact = 1;
  for(int i=2; i<=n; i++) { // 1 * anything is itself
    nfact *= i;
  }
  return nfact;
}
```

This is the iterative solution (using a loop)

Traditional Example (Not naturally recursive but we can do it....)

- You can't really do a negative factorial
- 0! = 1
- 1! = 1
- 2! = 2 // 2 * 1
- 3! = 6 // 3 * 2 * 1

- **A pattern starts to form**
  - If n <= 1, the answer is 1
  - If n > 1 we need to multiply n by everything that comes before it

```
public static long fact(int n) {
  if(n<=1)
    return 1;
  else
    return n * fact(n-1);
}
```

recursive approach (i.e. method calls itself)

Traditional Example (Not naturally recursive but we can do it....)

- You can't really do a negative factorial
- 0! = 1
- 1! = 1
- 2! = 2 // 2 * 1
- 3! = 6 // 3 * 2 * 1

```
public static long fact(int n) {
  if(n<=1)
    return 1;
  else
    return n * fact(n-1);
}
```

- **A pattern starts to form**
  - If n <= 1, the answer is 1
  - If n > 1 we need to multiply n by everything that comes before it
- **And since clearly you are all wondering:**
  - **yes, we can make any loop into a recursive method call instead**

Breaking Problems Down

- Recursion is broken into 2 cases:
  - Base Case (Simple or “Smallest Subset of the Problem” Case)
  - Recursive Case (“Keep Making the Problem Smaller” Case)
- **Recursive Case**
  - calling ourself with a step-wise decline in problem size
- **Base Case**
  - A way to stop the method calls
- *But what if we don’t stop the method calls?*
  - We can’t call forever, if we do, it’s a bit like an infinite loop (with more explosions)
    - Infinite loops do not cause more memory usage because they are all in the same active method
    - Each recursive call is a new “activation record” (pancake) on the runtime stack. Eventually the stack gets too tall and topples over (StackOverflowError)

giphy

A Stack reminder

- How are programs run!
- Each method call can be thought of a bit like a pancake on the pancake stack
- If we make a method call inside a method (like System.out.println() inside our main) it is a bit like adding a pancake on to the stack before you finished eating the first one
- Each pancake is chocked full of the ingredients needed to make it yummy
    - Each method record on the runtime stack has all the variable information it needs
- When a method finishes running, we eat that pancake and it disappears from the stack
- Our plates can only hold so much before it topples over

A Stack reminder

- You don’t *really* need to keep a visualization of the runtime stack in your head with all it’s scoped variables, returns, calls, etc. We really can just visualize it like pancakes with a sprinkle of magic

Solving Recursive Problems:

- The main takeaways are:
  - Find a way to solve the problem by using the solution to a slightly smaller version of the same problem
  - Find an easy case (base case) that will always be reached if the problem keeps getting smaller
- Sometimes it is helpful to write out the formula to solving these problems for lots of different inputs, looking for that pattern
- If you can prove it works for the base case and one case up, it will probably work for all the cases. Think of it a bit like induction (for you math nerds)

Aside:

The magic of Recursion: Teaching my cat to climb down a ladder

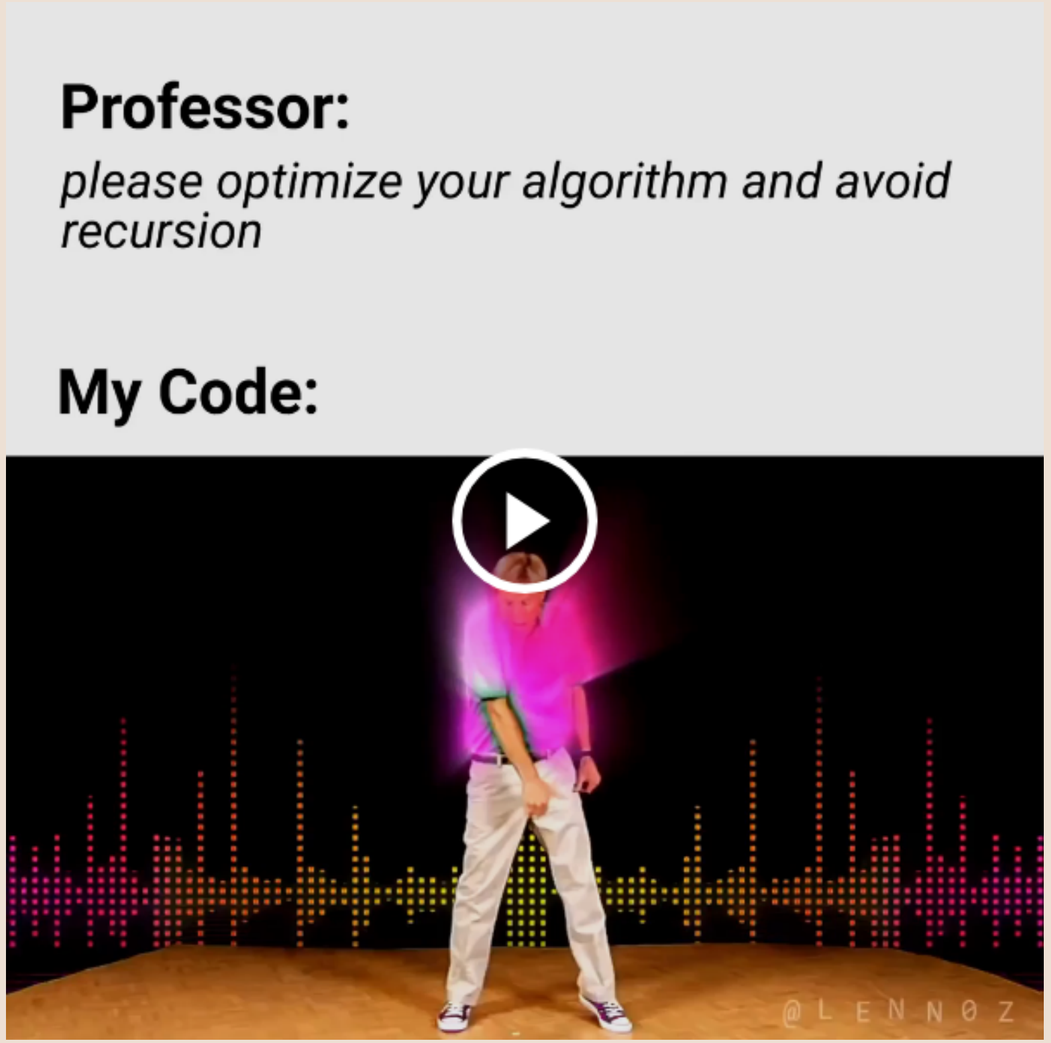
- Imagine I want to teach my cat to climb down a ladder with n steps...yikes
- Let's make this easy using recursion!
- **Base Case**
  - teach the cat to climb from the first step of the ladder to the floor. Place a treat on the floor, and the cat climbs down to get it. (YES DID IT!)
- **Recursive Step**
  - Next, consider the recursive step. Suppose the cat has learned to climb down **k** steps of the ladder. The task now is to teach it to climb down k + 1 steps.

- **MAGIC**
  - You don't need to start from scratch. Instead, you use the knowledge the cat has gained from climbing down k steps and apply it to the next step. Place a treat on the k+1 step, and encourage the cat to climb down one more step than it did previously.

```
function catDescent(int stairsLeft) {
  if(atBottom) {
    // tell cat they are a good kitty
  } else {
    catDescent(stairsLeft-1);
  }
}
```

Why are we doing this if we can just use loops...?

- Some problems are very difficult to solve iteratively and very simple to solve recursively.
- **just because we know about recursion doesn't mean we need to use it everywhere**
  - it's just a good tool in your toolbox
- Recursive code might be "physically" shorter code! (Doesn't mean it's actually shorter when executing though, example to come)



Bootstrap/Helper Methods

`public static int sumOf(int[] data)`

- Consider a method that takes in an array of ints, calculates the sum of the elements and returns that sum.

Bootstrap/Helper Methods

`public static int sumOf(int[] data)`

- Consider a method that takes in an array of ints, calculates the sum of the elements and returns that sum.

```
public static int sumOf(int[] data) {
    if data.length == 0 {
        return 0;
    } else {
        int n = data.length - 1;
        return data[n-1] + the sum of the first n-1 elements
    }
}
```

Bootstrap/Helper Methods

```
public static int sumOf(int[] data)
```

- Consider a method that takes in an array of ints, calculates the sum of the elements and returns that sum.

```
public static int sumOf(int[] data) {
    if data.length == 0 {
        return 0;
    } else {
        int n = data.length - 1;
        return data[n-1] + the sum of the first n-1 elements
                           sumOf(....?!?!?!.....);
    }
}
```

This method has no way of letting us specify the number of elements to sum

HELPER METHODS!

```
public static int sumOf(int[] data) {
    return sumOf(data, data.length-1);
}
```

This is the “interface” that we want to provide to the user. The user shouldn’t have to worry about how the recursion is implemented.

// helper methods mean main doesn’t have to change but we can implement the method with recursion anyway

```
private static int sumOf(int[] data, int n) {
    if n == 0 {
        return data[0];
    } else {
        return data[n] + sumOf(data, n-1);
    }
}
```

HELPER METHODS!

```
public static int sumOf(int[] data) {
    return sumOf(data, data.length-1);
}
```

This is the "interface" that we want to provide to the user. The user shouldn't have to worry about how the recursion is implemented.

// helper methods mean main doesn't have to change but we can implement the method with recursion anyway

```
private static int sumOf(int[] data, int n) {
    if n == 0 {
        return data[0];
    } else {
        return data[n] + sumOf(data, n-1);
    }
}
```

This is private since it should just be used internally (not by the "user". Only called by the Helper "interface" method

Pause & Practice (With Me)

- Let’s create a method convertToBinary(int n) that takes in a number n >=0 and converts it to a binary representation
- return this value as an Array<Integer> of 0s and 1s.
- **no helper method**

- Base Case: n == 0: (return an empty ArrayList)
- Recursive Case: (n>0)
  - Convert n/2 binary
  - if n is even add a 0 to the end of ArrayList
  - if n is odd add a 1 to the end of ArrayList
  - Hint: wait a second…n%2 can only return 0 or 1….

Pause & Practice

1. **Sum Natural Numbers** - Write a recursive method in Java that calculates the sum of the first **n** natural numbers. Natural numbers start from 1. For example, if **n** is 5, the sum would be 1 + 2 + 3 + 4 + 5 = 15.
2. **Reverse a String** - Write a recursive method in Java to reverse a string. For example, if the input string is "hello", the output should be "olleh".
3. **Count the Digits in a Number** - Write a recursive method in Java to count the number of digits in a given integer. For example, if the input is 12345, the output should be 5.
4. **Power of a Number** - Write a recursive method in Java to compute the power of a number. Given a base b and an exponent e, calculate b raised to the power of e. For example, power(2, 4) should return 16, since 2^4 = 16.