YEAH Assignment 3

Recursion!

Some Logistics

- This assignment is broken down into **4 parts.** We think they're a little more involved than your previous assignments, so be sure to start early!

Some Logistics

- This assignment is broken down into **4 parts.** We think they're a little more involved than your previous assignments, so be sure to start early!
- **Pair programming is allowed on this assignment!**

Some Logistics

- This assignment is broken down into **4 parts.** We think they're a little more involved than your previous assignments, so be sure to start early!
- **Pair programming is allowed on this assignment!**
    - Be aware that you should be working together on **all** parts of this assignment. If you don't implement parts of this assignment, you'll be at a significant **disadvantage** on the exams!

Some Logistics

- This assignment is broken down into **4 parts.** We think they're a little more involved than your previous assignments, so be sure to start early!
- **Pair programming is allowed on this assignment!**
  - Be aware that you should be working together on **all** parts of this assignment. If you don't implement parts of this assignment, you'll be at a significant **disadvantage** on the exams!
- Small point: if you're on Windows and you're getting build errors right off the bat, you'll need to redownload the starter code!

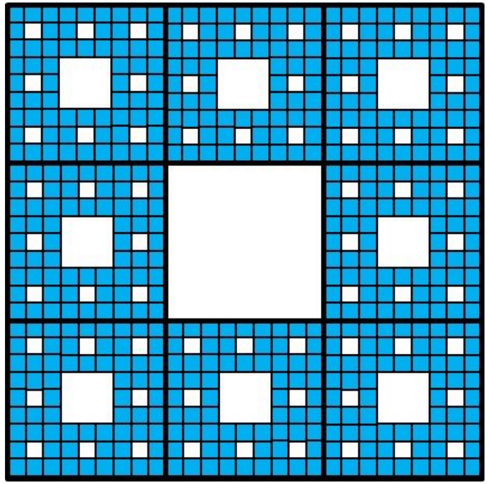
Let’s begin!

Part 1: The Sierpinski Triangle

- Remember our beautiful Sierpinski Carpet?

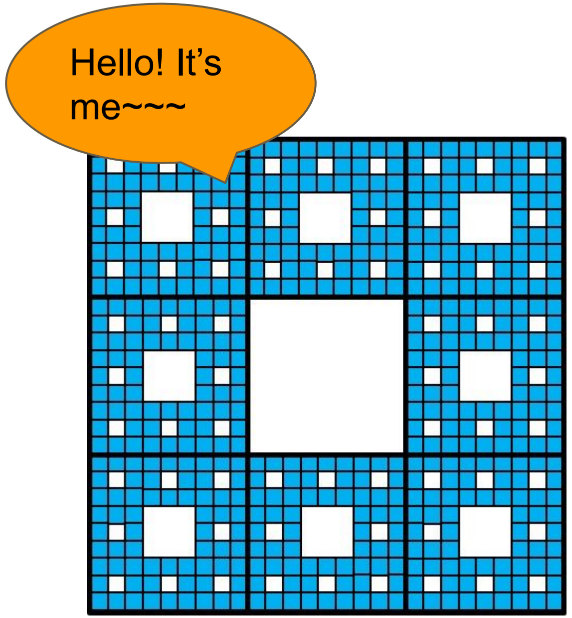
Part 1: The Sierpinski Triangle

- Remember our beautiful Sierpinski Carpet?



Part 1: The Sierpinski Triangle

- Remember our beautiful Sierpinski Carpet?



Part 1: The Sierpinski Triangle

- Remember our beautiful Sierpinski Carpet?
- We are beginning A3 with another fascinating discovery of Waclaw Sierpinski

Part 1: The Sierpinski Triangle

- Remember our beautiful Sierpinski Carpet?
- We are beginning A3 with another fascinating discovery of Waclaw Sierpinski

Waclaw Sierpinski, trusted distributor of CS 106B material

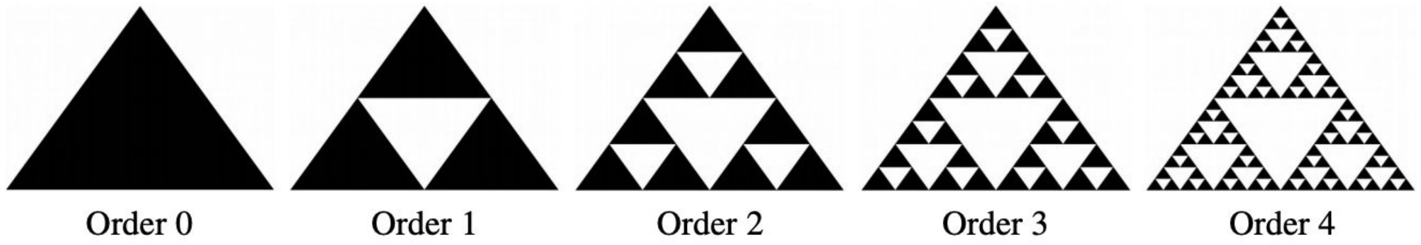
Part 1: The Sierpinski Triangle

- Remember our beautiful Sierpinski Carpet?
- We are beginning A3 with another fascinating discovery of Waclaw Sierpinski

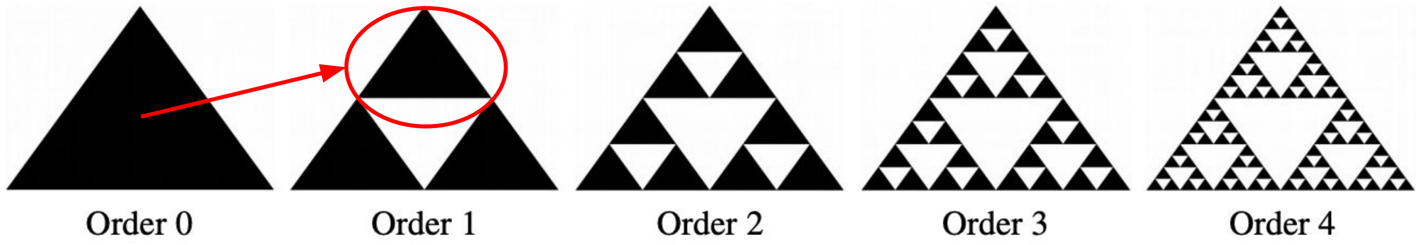
“I dedicate these shapes to Keith Schwarz” - Sierpinski, probably

Waclaw Sierpinski, trusted distributor of CS 106B material

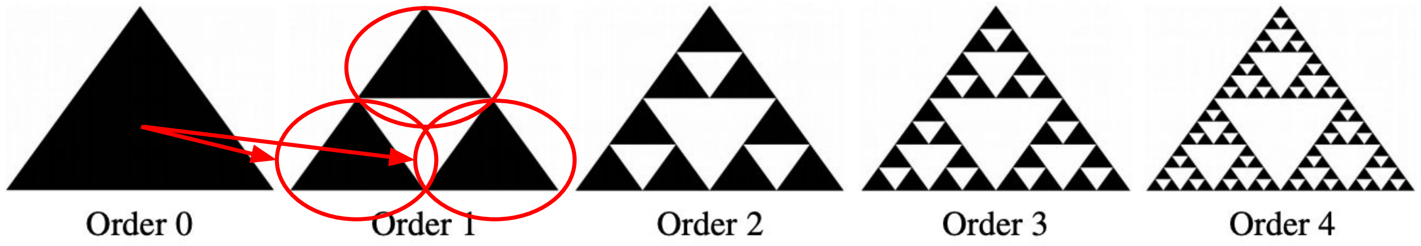
Part 1: The Sierpinski Triangle



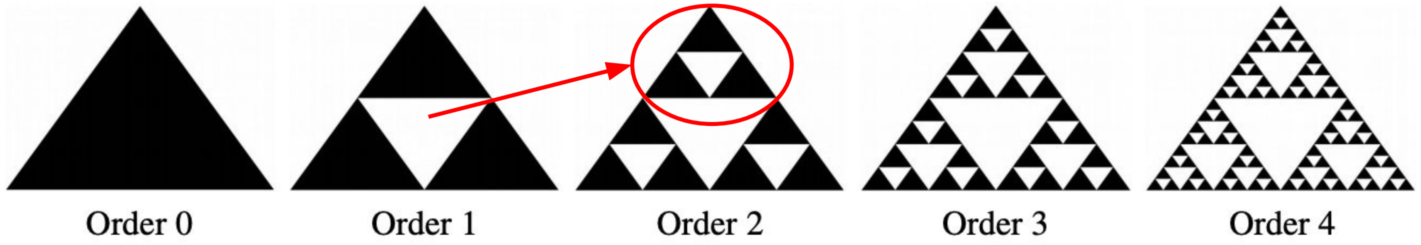
Part 1: The Sierpinski Triangle



Part 1: The Sierpinski Triangle

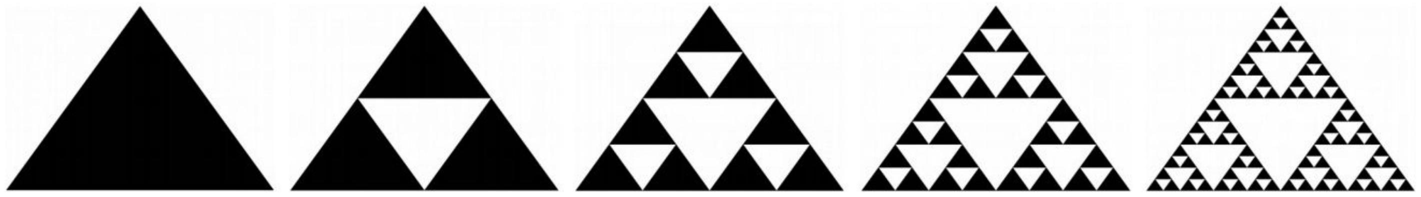


Part 1: The Sierpinski Triangle



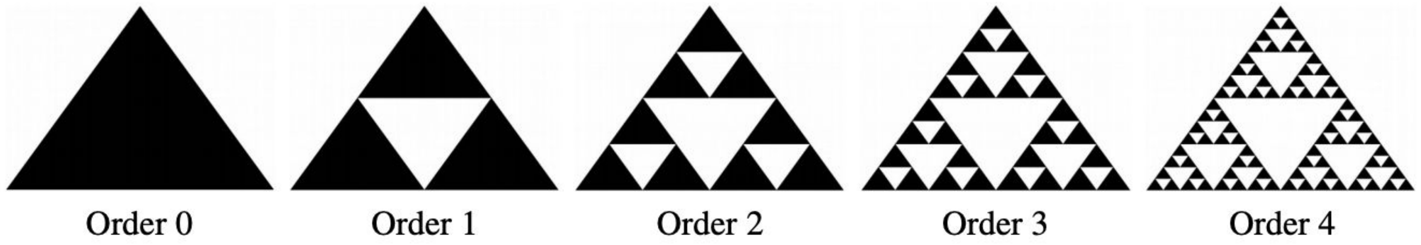
Part 1: The Sierpinski Triangle

- You are writing a program that draws *n-order* Sierpinski Triangles

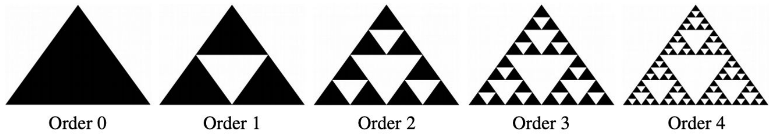


Part 1: The Sierpinski Triangle

- You are writing a program that draws *n-order* Sierpinski Triangles
- The Sierpinski triangle is defined *recursively*, meaning:
    - An order-0 Sierpinski triangle is a plain filled triangle.
    - An order-*n* Sierpinski triangle, where *n > 0*, consists of three Sierpinski triangles of order *n – 1*, each half as large as the main triangle, arranged so that they meet corner-to-corner.



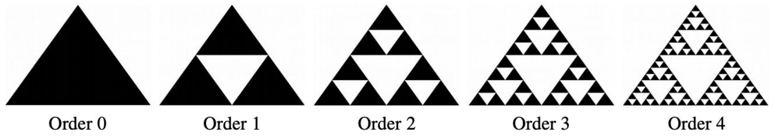
Part 1: The Sierpinski Triangle



- You are responsible for handling two functions
- The first function draws a black triangle on the canvas given the 3 vertices

```
void drawTriangle(GWindow& window,
                  double x0, double y0,
                  double x1, double y1,
                  double x2, double y2);
```

Part 1: The Sierpinski Triangle

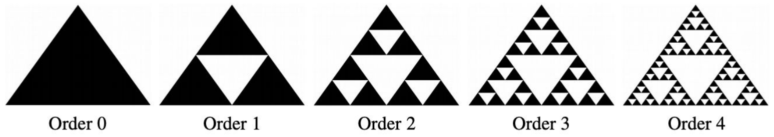


- You are responsible for handling two functions
- The first function draws a black triangle on the canvas given the 3 vertices

I'm already implemented for you :)

```
void drawTriangle(GWindow& window,
                  double x0, double y0,
                  double x1, double y1,
                  double x2, double y2);
```

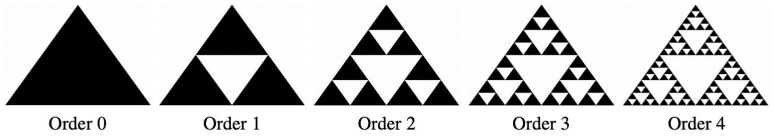
Part 1: The Sierpinski Triangle



- You are responsible for handling two functions
- The first function draws a black triangle on the canvas given the 3 vertices
- The second function is the recursive function you need to implement

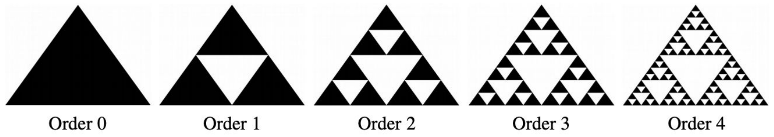
```
void drawSierpinskiTriangle(GWindow& window,
                            double x0, double y0,
                            double x1, double y1,
                            double x2, double y2,
                            int order)
```

Part 1: The Sierpinski Triangle



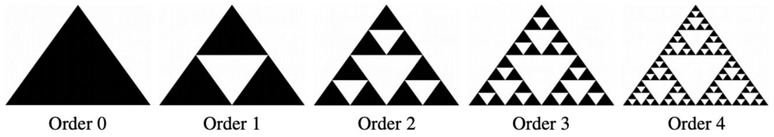
- A few implementation thoughts:
  - If order is negative you should throw an error!

Part 1: The Sierpinski Triangle



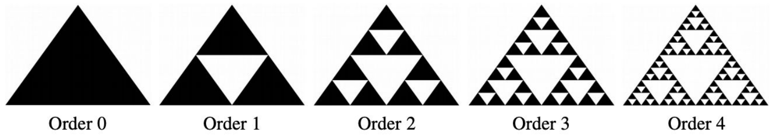
- A few implementation thoughts:
    - If the order is negative you should throw an error!
    - For any given recursive case, how many calls to drawSierpinskiTriangle() should you be making? At what locations?

Part 1: The Sierpinski Triangle



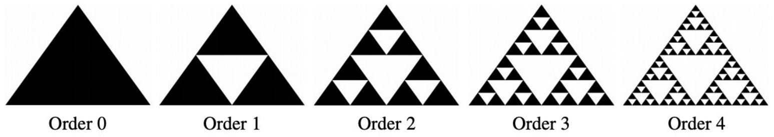
- A few implementation thoughts:
    - If the order is negative you should throw an error!
    - For any given recursive case, how many calls to drawSierpinskiTriangle() should you be making? At what locations?
    - An order-0 Sierpinski triangle is a plain filled triangle.

Part 1: The Sierpinski Triangle



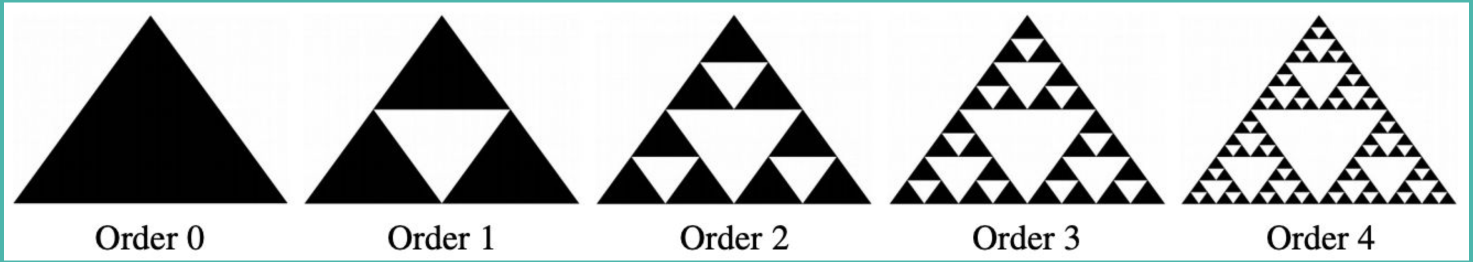
- A few implementation thoughts:
  - If the order is negative you should throw an error!
  - For any given recursive case, how many calls to drawSierpinskiTriangle() should you be making? At what locations?
  - An order-0 Sierpinski triangle is a plain filled triangle.
  - An order-n Sierpinski triangle, where n > 0, consists of three Sierpinski triangles of order n – 1, each half as large as the main triangle, arranged so that they meet corner-to-corner

Part 1: The Sierpinski Triangle



- A few implementation thoughts:
  - If the order is negative you should throw an error!
  - For any given recursive case, how many calls to drawSierpinskiTriangle() should you be making? At what locations?
  - An order-0 Sierpinski triangle is a plain filled triangle.
  - An order-n Sierpinski triangle, where n > 0, consists of three Sierpinski triangles of order n – 1, each half as large as the main triangle, arranged so that they meet corner-to-corner
  - Highly recommend drawing this one out and planning exactly where your points are going to be before coding. Even if you understand this problem, it's still easy to make math errors (trust me! - Trip, who gets this wrong every time he tries it)

Questions about Part 1?



Part 2: Human Pyramids

- We will now use recursion to examine why life is just not fair...

Part 2: Human Pyramids

- We will now use recursion to examine why life is JUST not fair...
- Have you ever made a human pyramid with your friends, and you were placed at the bottom center?

Part 2: Human Pyramids

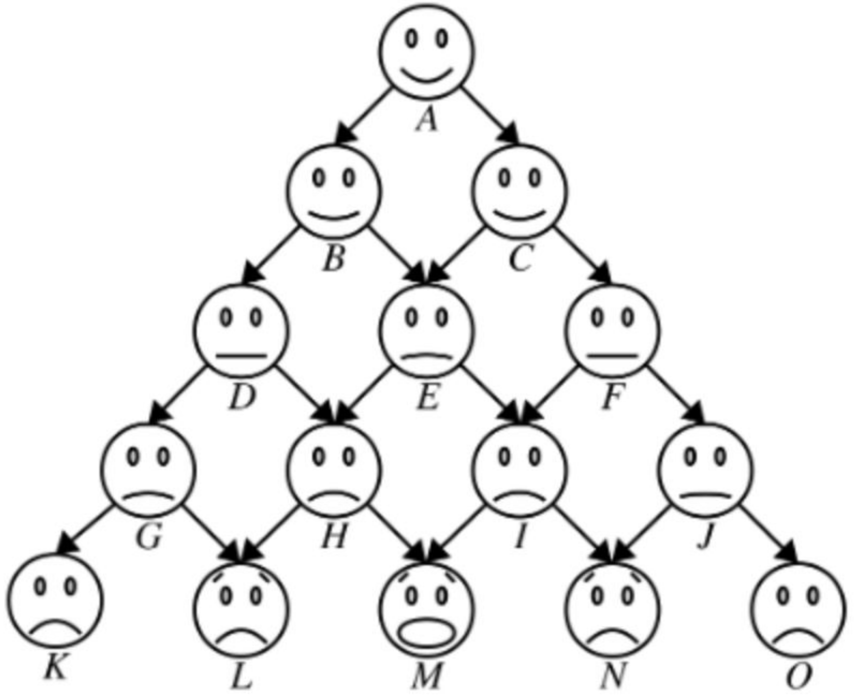
- We will now use recursion to examine why life is JUST not fair…
- Have you ever made a human pyramid with your friends, and you were placed at the bottom center?
- …

Part 2: Human Pyramids

- We will now use recursion to examine why life is JUST not fair…
- Have you ever made a human pyramid with your friends, and you were placed at the bottom center?
- …
- OUCH!

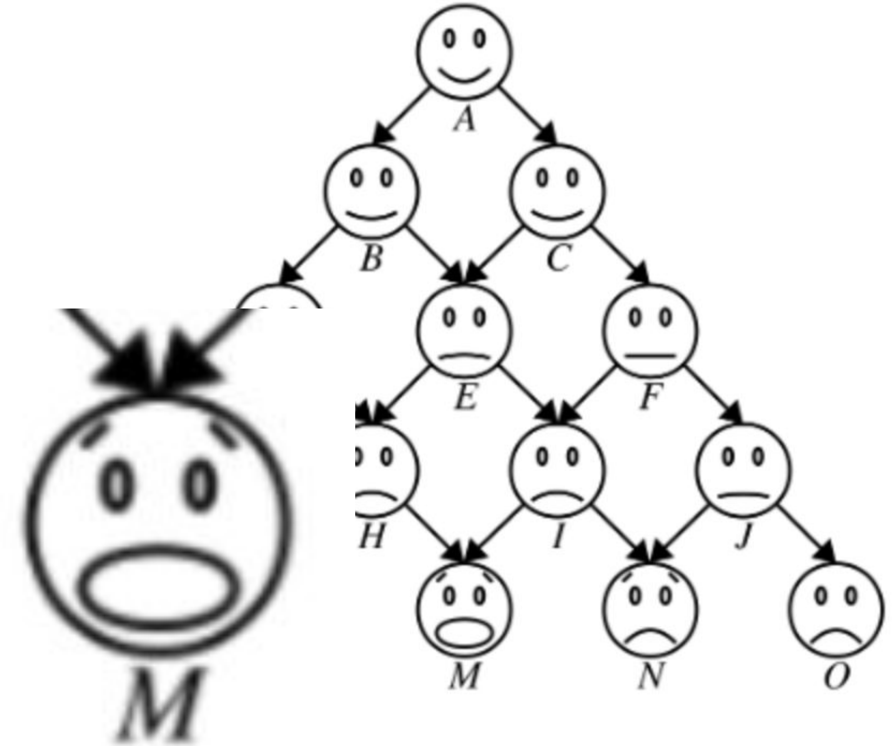
Part 2: Human Pyramids

- Here’s a human pyramid:



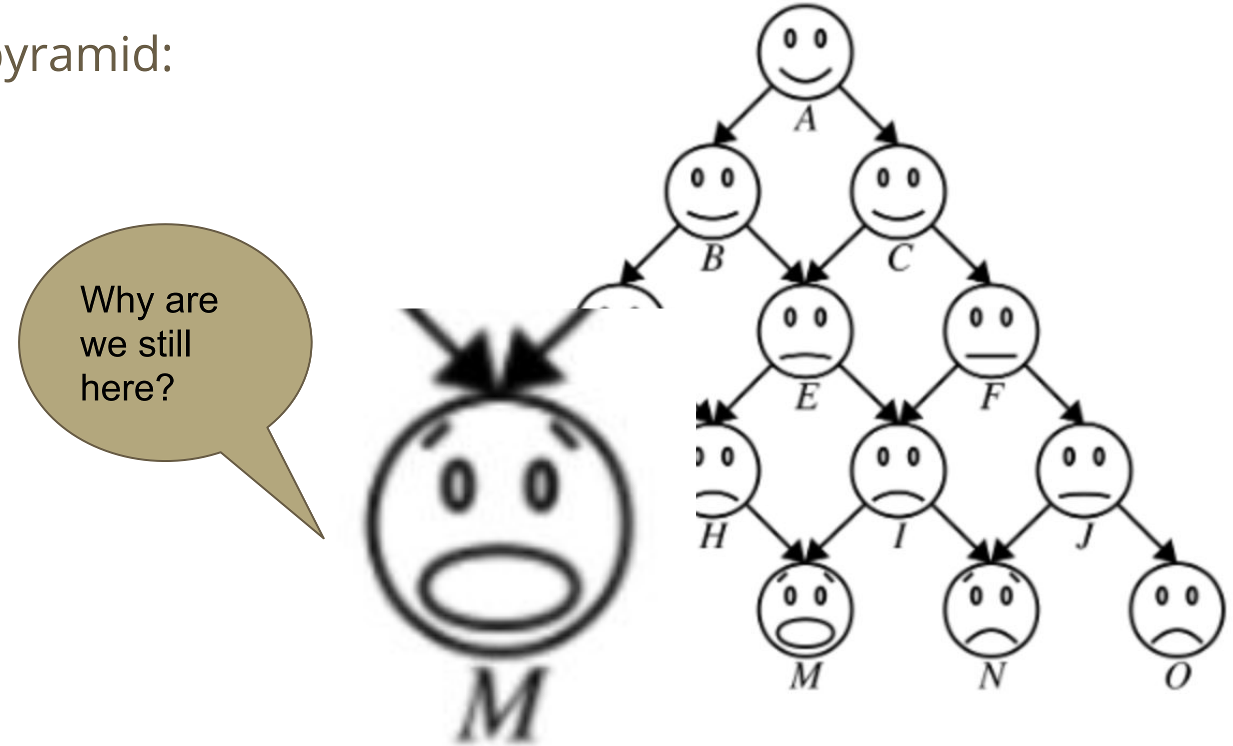
Part 2: Human Pyramids

- Here's a human pyramid:



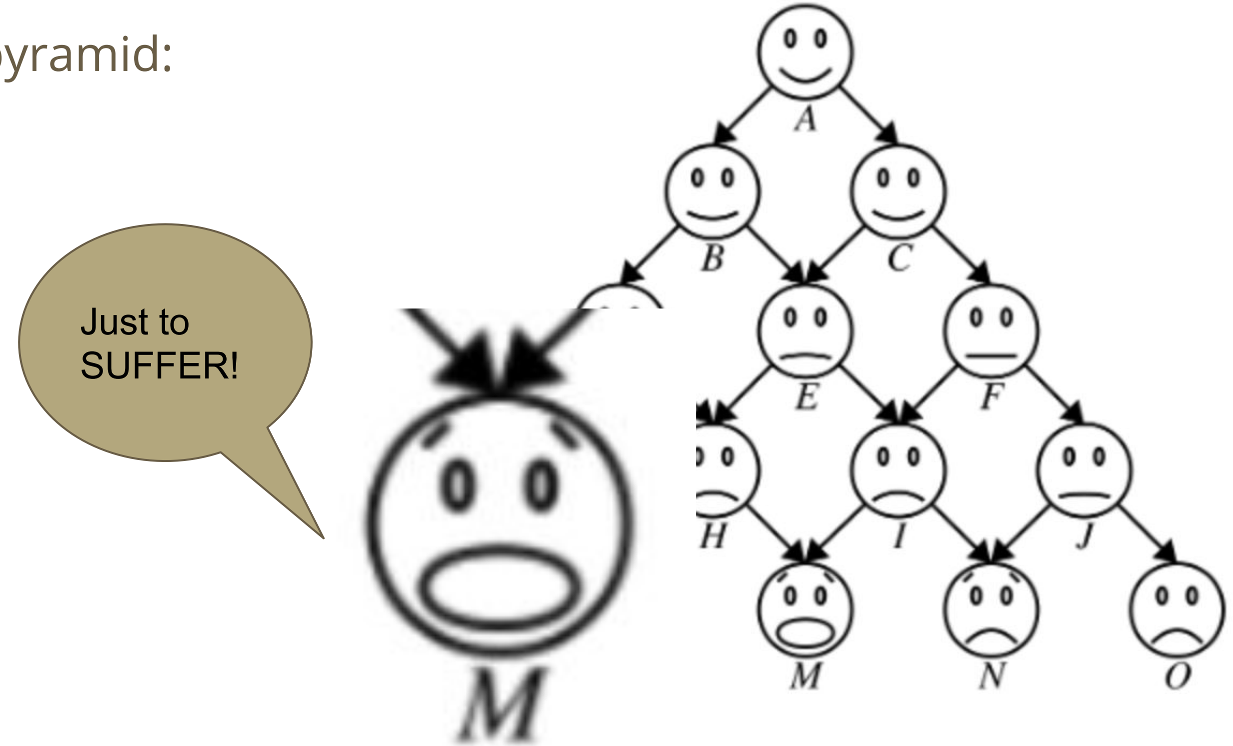
Part 2: Human Pyramids

- Here's a human pyramid:



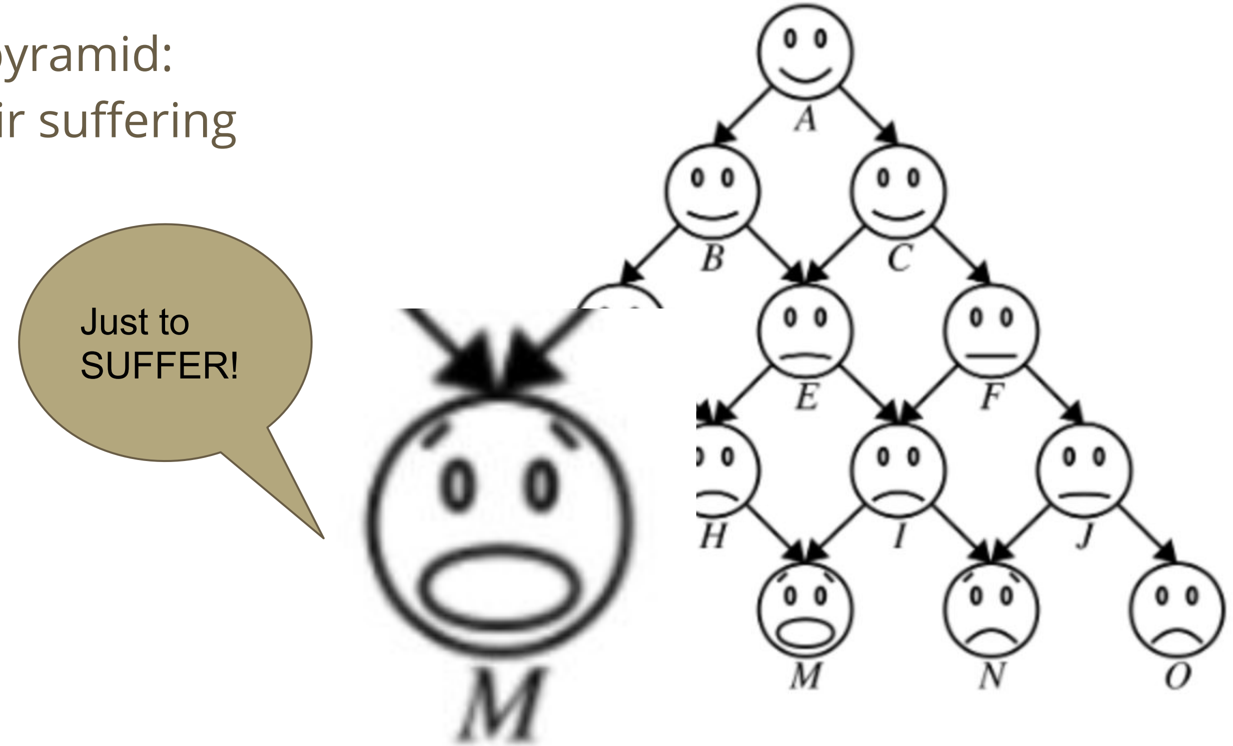
Part 2: Human Pyramids

- Here's a human pyramid:



Part 2: Human Pyramids

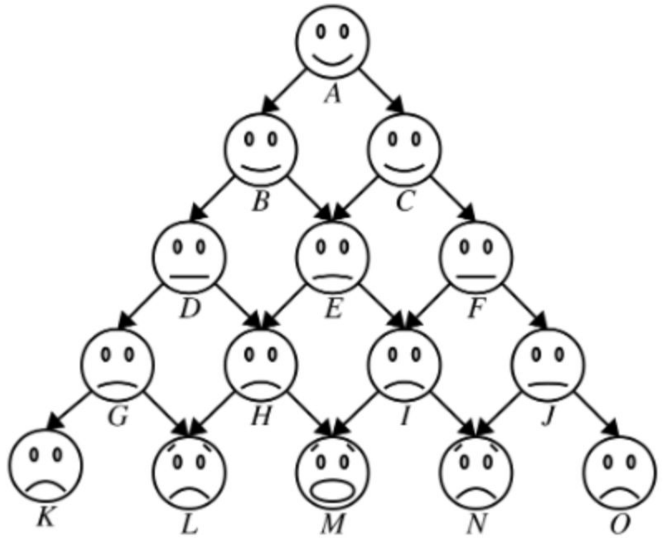
- Here's a human pyramid:
- Let's quantify their suffering



M

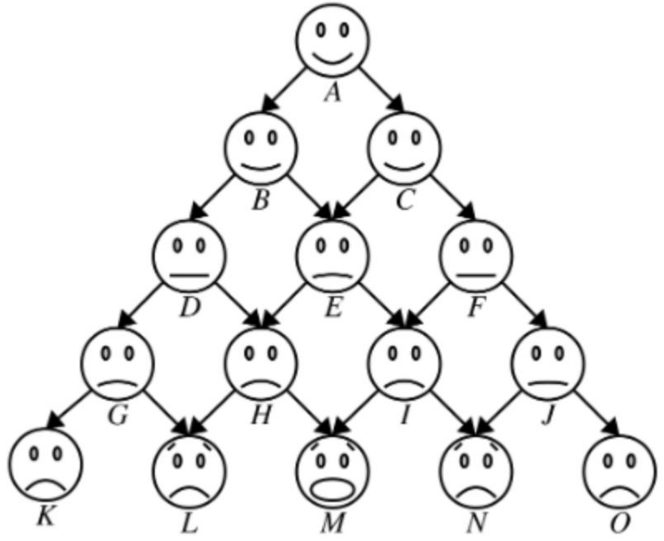
Part 2: Human Pyramids

- Problem set-up
  - Each person supports half the body weight of each of the people immediately above them, plus half of the weight that each of those people are supporting.
  - Each person weighs 160 pounds.



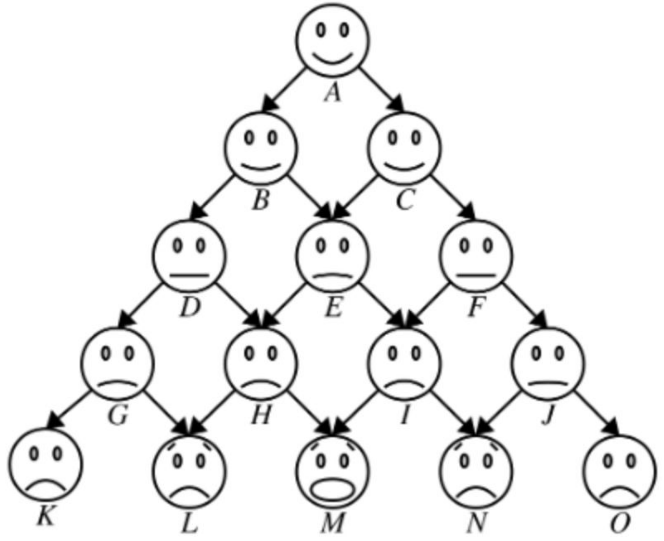
Part 2: Human Pyramids

- Let’s try a few examples!



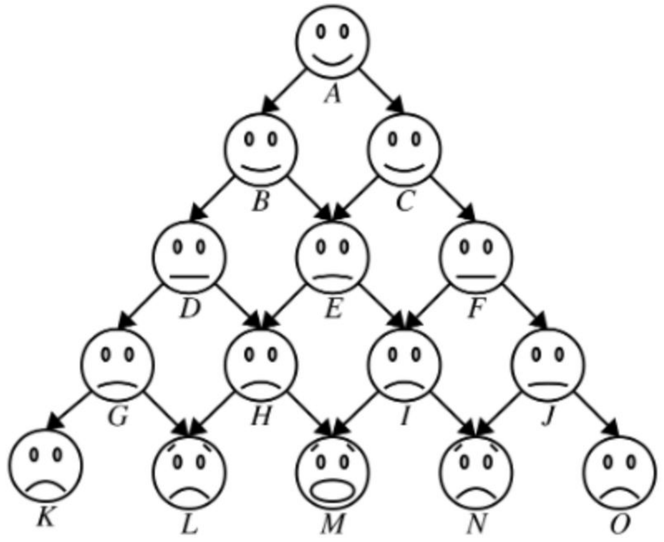
Part 2: Human Pyramids

- How much weight is **A** carrying?



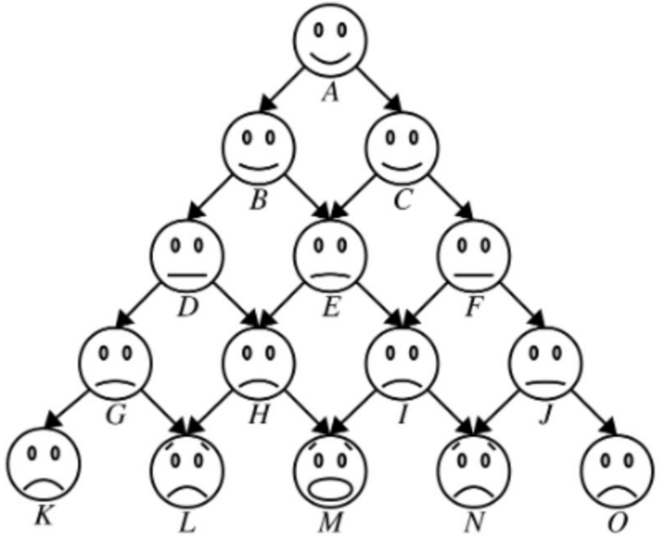
Part 2: Human Pyramids

- How much weight is **A** carrying?
  - **0** lbs; no calculation needed, there are no one above them **A**.



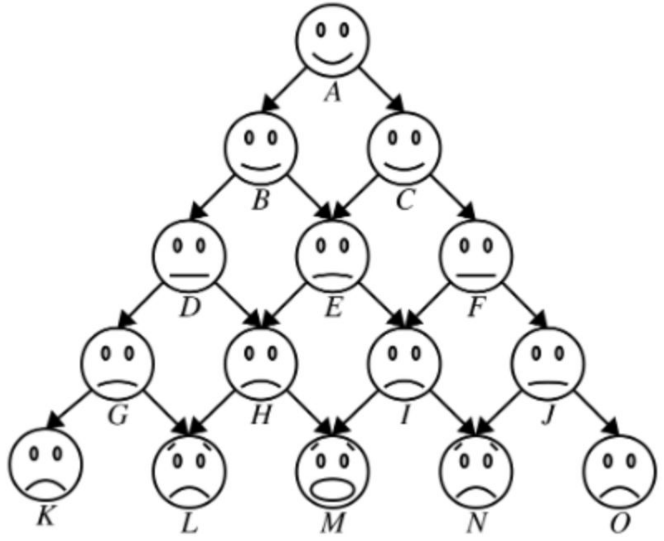
Part 2: Human Pyramids

- How much weight is **A** carrying?
  - **0** lbs; no calculation needed, there are no one above them **A**.
  - This problem almost… *trivial* ;)



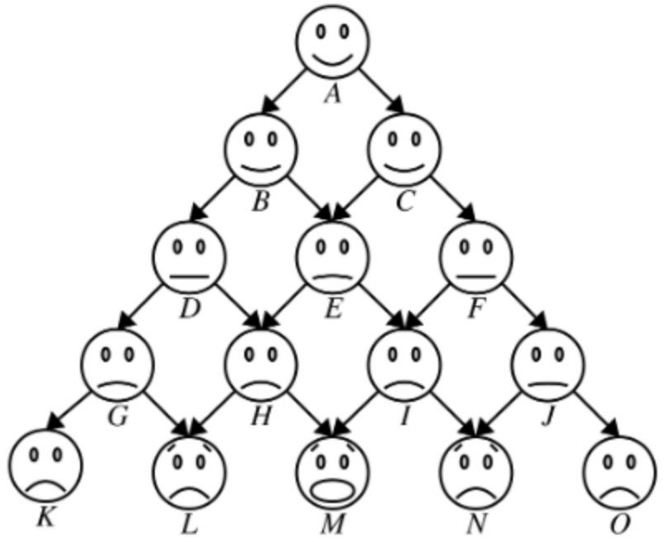
Part 2: Human Pyramids

- How much weight is **D** carrying?



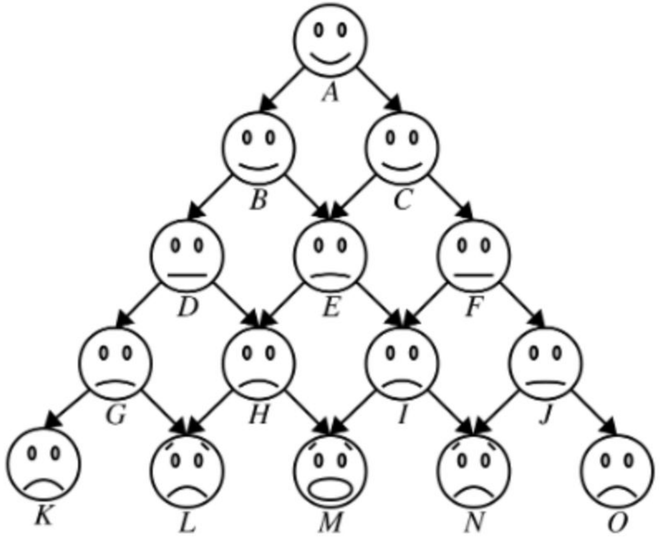
Part 2: Human Pyramids

- How much weight is **D** carrying?
  - **B** is above **D**, so **D** carries half of **B**'s weight.



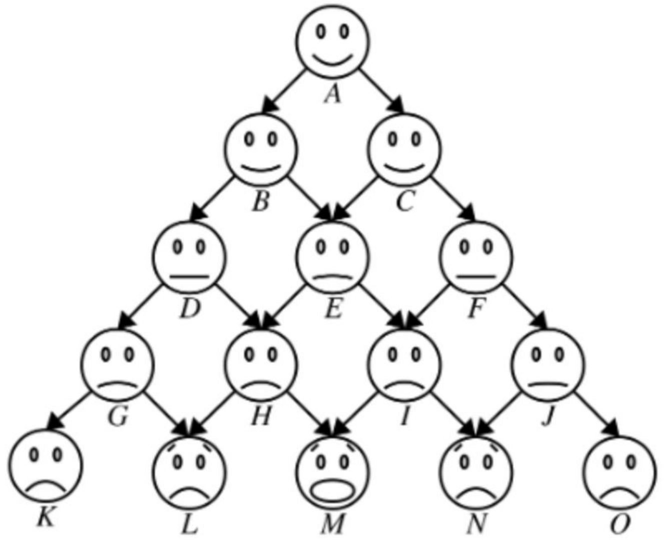
Part 2: Human Pyramids

- How much weight is **D** carrying?
  - **B** is above **D**, so **D** carries half of **B**'s weight.
    - That is 80 lbs.
  - **D** also carries half of the weight **B** is carrying.



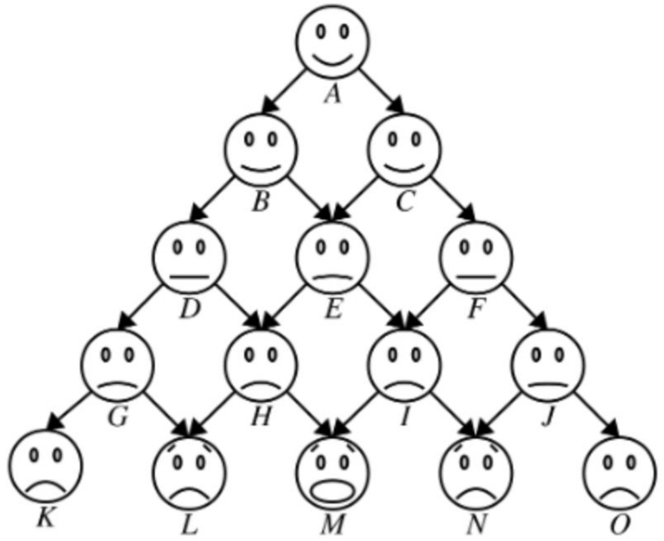
Part 2: Human Pyramids

- How much weight is **D** carrying?
    - **B** is above **D**, so **D** carries half of **B**'s weight.
        - That is 80 lbs.
    - **D** also carries half of the weight **B** is carrying.
        - How much weight is **B** carrying?



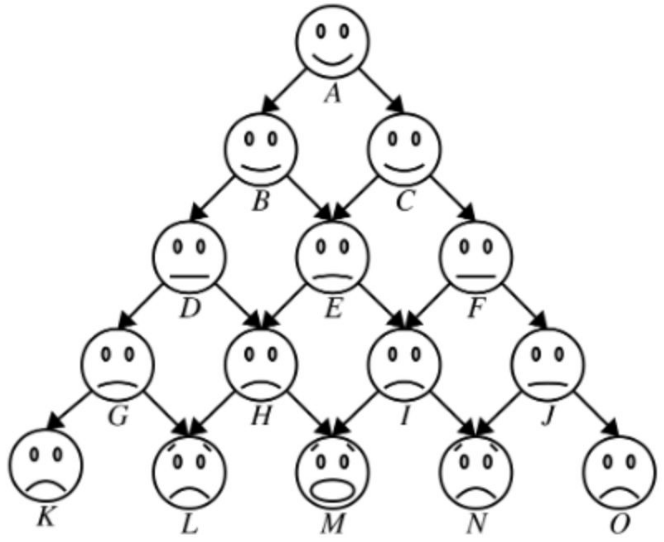
Part 2: Human Pyramids

- How much weight is **D** carrying?
  - **B** is above **D**, so **D** carries half of **B**'s weight.
    - That is 80 lbs.
  - **D** also carries half of the weight **B** is carrying.
    - How much weight is **B** carrying?
      - **A** is above **B**, so **B** carries half of **A**'s weight.



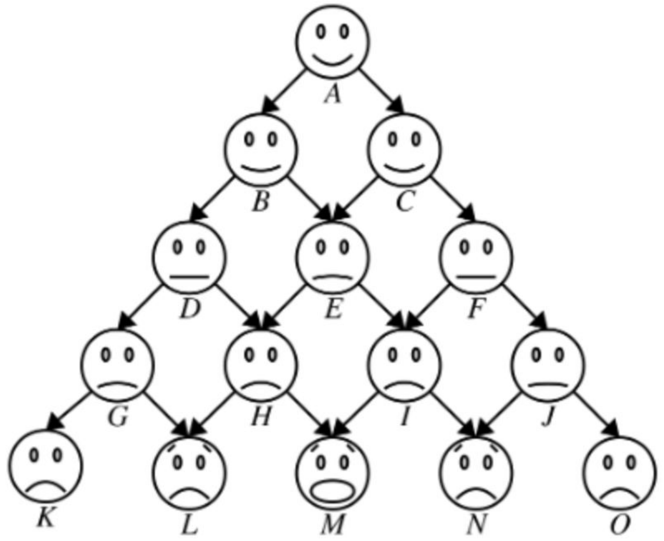
Part 2: Human Pyramids

- How much weight is **D** carrying?
    - **B** is above **D**, so **D** carries half of **B**'s weight.
        - That is 80 lbs.
    - **D** also carries half of the weight **B** is carrying.
        - How much weight is **B** carrying?
            - **A** is above **B**, so **B** carries half of **A**'s weight.
                - That is 80 lbs. And half of that is 40 lbs.



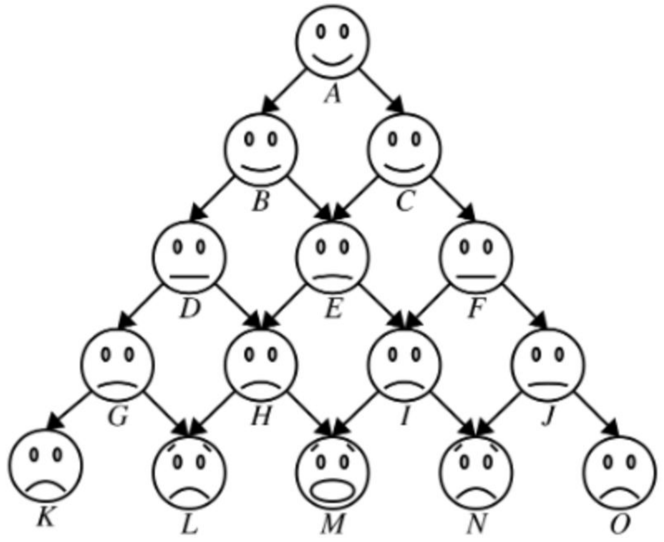
Part 2: Human Pyramids

- How much weight is **D** carrying?
    - **B** is above **D**, so **D** carries half of **B**'s weight.
        - That is 80 lbs.
    - **D** also carries half of the weight **B** is carrying.
        - How much weight is **B** carrying?
            - **A** is above **B**, so **B** carries half of **A**'s weight.
                - That is 80 lbs. And half of that is 40 lbs.
    - In total, **D** is carrying **120** lbs.



Part 2: Human Pyramids

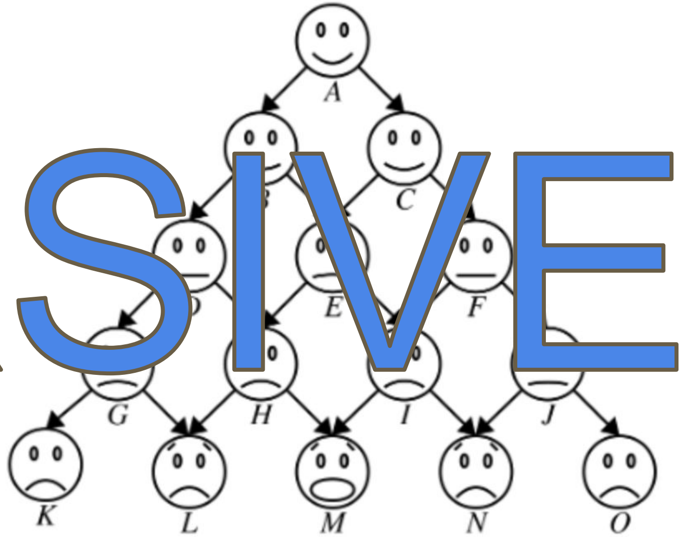
- How much weight is **D** carrying?
    - **B** is above **D**, so **D** carries half of **B**'s weight.
        - That is 80 lbs.
    - **D** also carries half of the weight **B** is carrying.
        - How much weight is **B** carrying?
            - **A** is above **B**, so **B** carries half of **A**'s weight.
                - That is 80 lbs. And half of that is 40 lbs.
    - In total, **D** is carrying **120** lbs.



Part 2: Human Pyramids

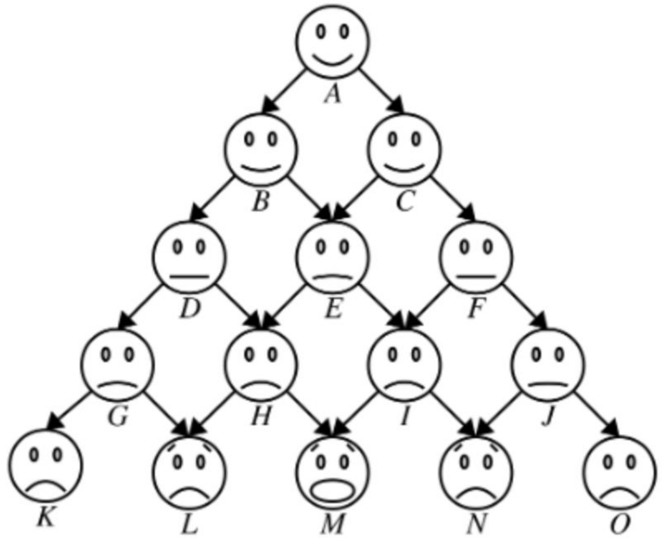
- How much weight is **D** carrying?
  - **B** is above **D**, so **D** carries half of **B**'s weight.
    - That is 80 lb
  - **D** also carries half of the weight **B** is carrying.
    - How mu n weight is carryin ?
      - **A** is above so carrie ha of **A**'s weight.
        - That is 80 lbs. And half of that is 40 lbs.
  - In total, **D** is carrying **120** lbs.

RECURSIVE



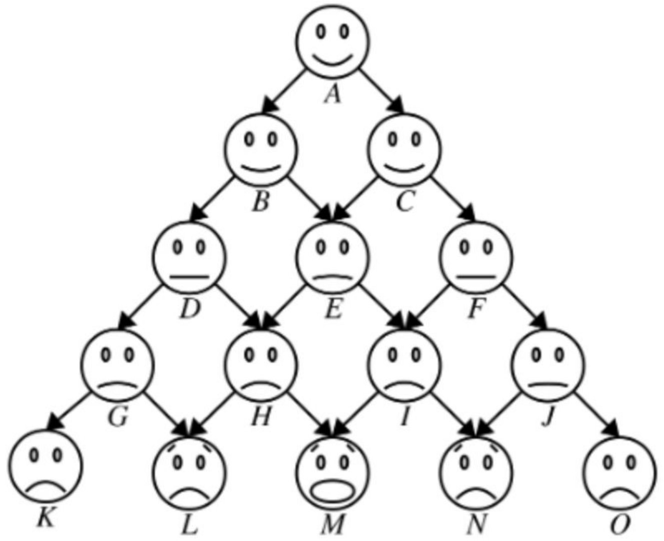
Part 2: Human Pyramids

- How much weight is **E** carrying?



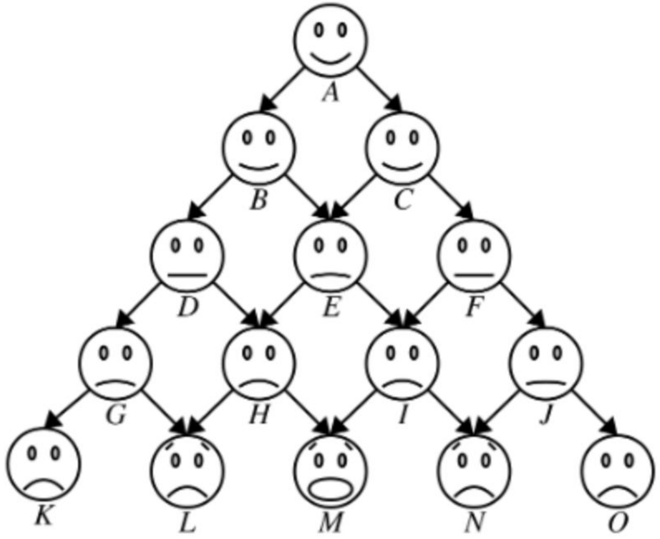
Part 2: Human Pyramids

- How much weight is **E** carrying?
    - To save some work, we know from the previous example that **B** yields 80 lbs (half of **B**'s weight) + 40 lbs (half of the weight **B** is carrying) = 120 lbs.



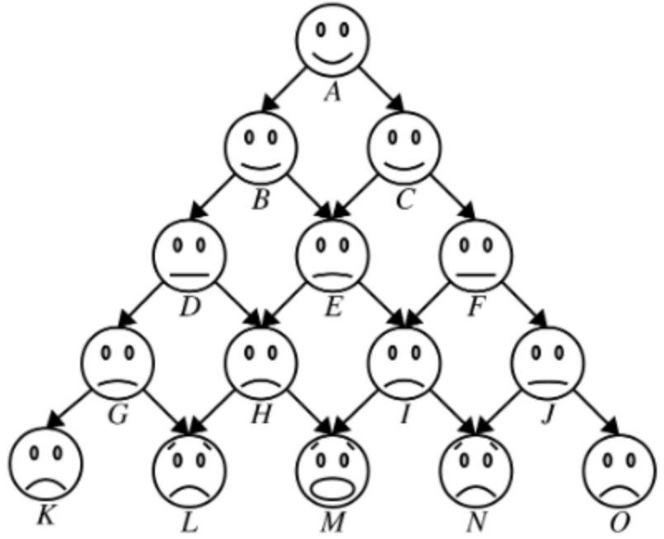
Part 2: Human Pyramids

- How much weight is **E** carrying?
  - To save some work, we know from the previous example that **B** yields 80 lbs (half of **B**'s weight) + 40 lbs (half of the weight **B** is carrying) = 120 lbs.
  - **C** is above **E**, so **E** carries half of **C**'s weight.



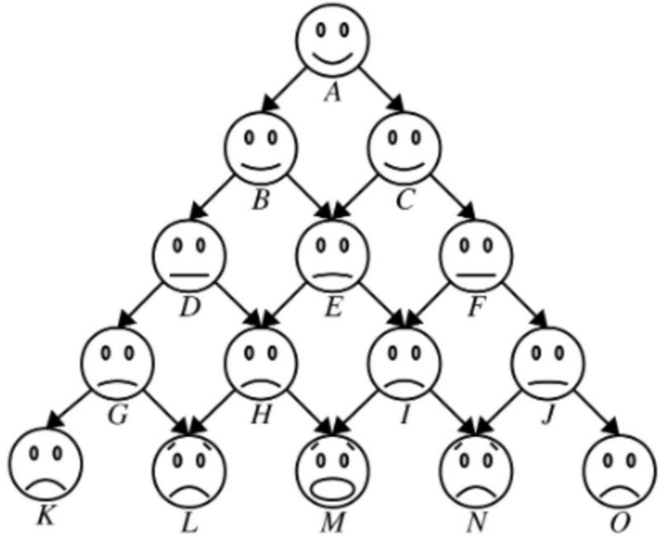
Part 2: Human Pyramids

- How much weight is **E** carrying?
  - To save some work, we know from the previous example that **B** yields 80 lbs (half of **B**'s weight) + 40 lbs (half of the weight **B** is carrying) = 120 lbs.
  - **C** is above **E**, so **E** carries half of **C**'s weight.
    - That is 80 lbs.



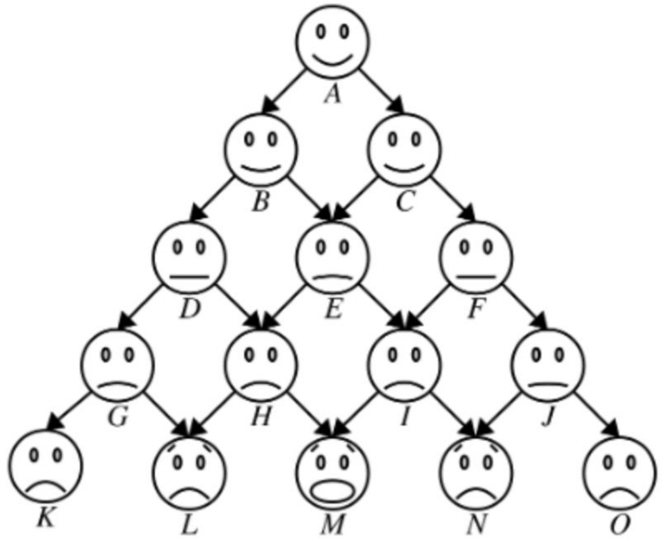
Part 2: Human Pyramids

- How much weight is **E** carrying?
  - To save some work, we know from the previous example that **B** yields 80 lbs (half of **B**'s weight) + 40 lbs (half of the weight **B** is carrying) = 120 lbs.
  - **C** is above **E**, so **E** carries half of **C**'s weight.
    - That is 80 lbs.
  - **E** also carries half of the weight **C** is carrying.



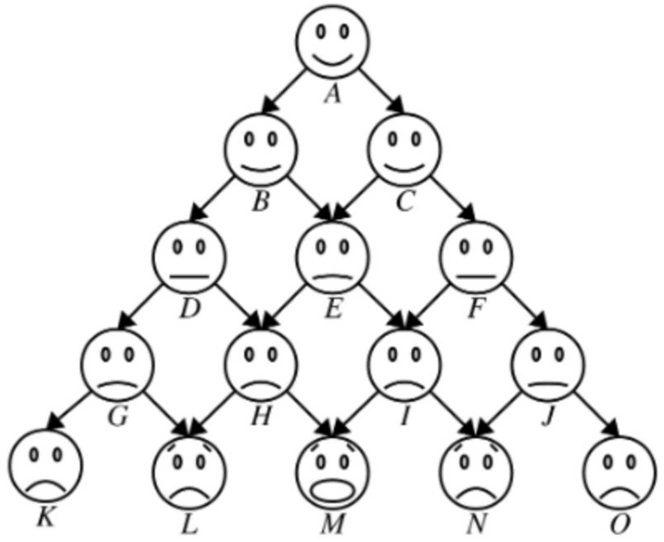
Part 2: Human Pyramids

- How much weight is **E** carrying?
    - To save some work, we know from the previous example that **B** yields 80 lbs (half of **B**'s weight) + 40 lbs (half of the weight **B** is carrying) = 120 lbs.
    - **C** is above **E**, so **E** carries half of **C**'s weight.
        - That is 80 lbs.
    - **E** also carries half of the weight **C** is carrying.
        - How much weight is **C** carrying?



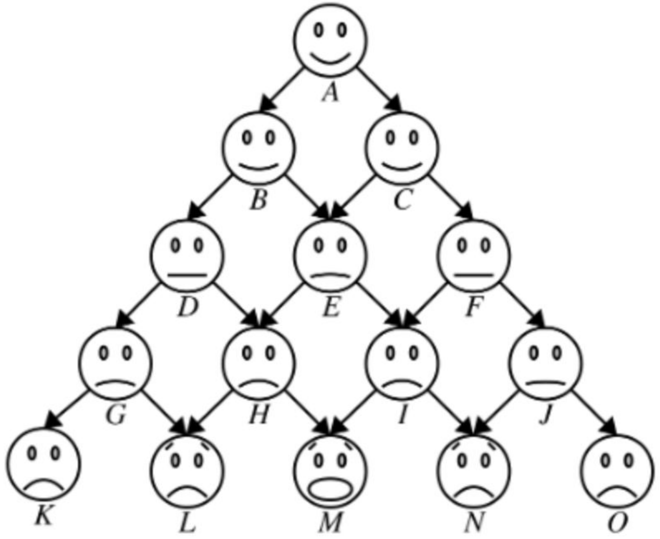
Part 2: Human Pyramids

- How much weight is **E** carrying?
  - To save some work, we know from the previous example that **B** yields 80 lbs (half of **B**'s weight) + 40 lbs (half of the weight **B** is carrying) = 120 lbs.
  - **C** is above **E**, so **E** carries half of **C**'s weight.
    - That is 80 lbs.
  - **E** also carries half of the weight **C** is carrying.
    - How much weight is **C** carrying?
      - **A** is above **C**, so **C** carries half of **A**'s weight.



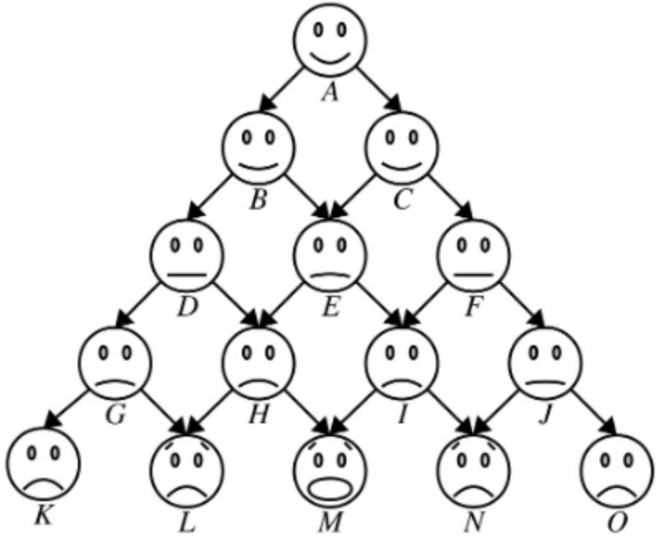
Part 2: Human Pyramids

- How much weight is **E** carrying?
  - To save some work, we know from the previous example that **B** yields 80 lbs (half of **B**'s weight) + 40 lbs (half of the weight **B** is carrying) = 120 lbs.
  - **C** is above **E**, so **E** carries half of **C**'s weight.
    - That is 80 lbs.
  - **E** also carries half of the weight **C** is carrying.
    - How much weight is **C** carrying?
      - **A** is above **C**, so **C** carries half of **A**'s weight.
        - That is 80 lbs. And half of that is 40 lbs.



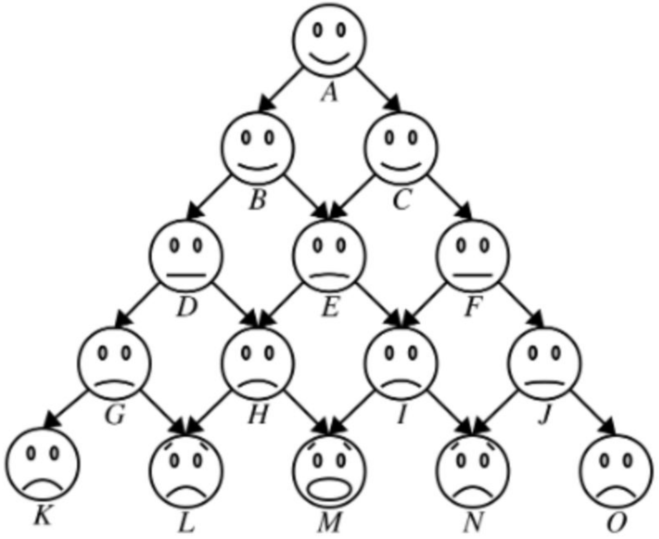
Part 2: Human Pyramids

- How much weight is **E** carrying?
  - To save some work, we know from the previous example that **B** yields 80 lbs (half of **B**'s weight) + 40 lbs (half of the weight **B** is carrying) = 120 lbs.
  - **C** is above **E**, so **E** carries half of **C**'s weight.
    - That is 80 lbs.
  - **E** also carries half of the weight **C** is carrying.
    - How much weight is **C** carrying?
      - **A** is above **C**, so **C** carries half of **A**'s weight.
        - That is 80 lbs. And half of that is 40 lbs.
  - In total, **E** is carrying **240** lbs.



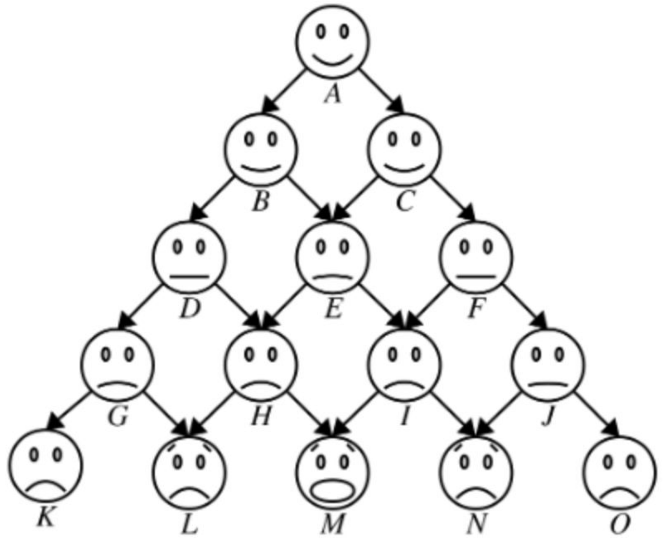
Part 2: Human Pyramids

- How much weight is **E** carrying?
    - To save some work, we know from the previous example that **B** yields 80 lbs (half of **B**'s weight) + 40 lbs (half of the weight **B** is carrying) = 120 lbs.
    - **C** is above **E**, so **E** carries half of **C**'s weight.
        - That is 80 lbs.
    - **E** also carries half of the weight **C** is carrying.
        - How much weight is **C** carrying?
            - **A** is above **C**, so **C** carries half of **A**'s weight.
                - That is 80 lbs. And half of that is 40 lbs.
    - In total, **E** is carrying **240** lbs.
- Despite the fact that **D** and **E** are on the same row, **E** is carrying double the weight **D** is carrying!



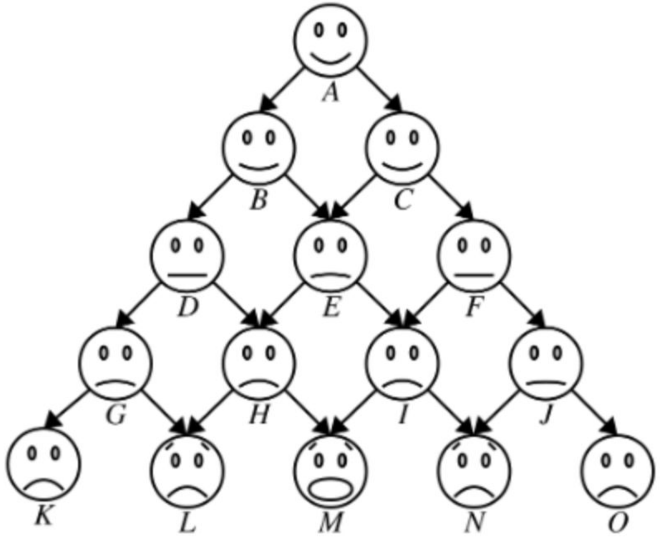
Part 2: Human Pyramids

- How much weight is **M** carrying?



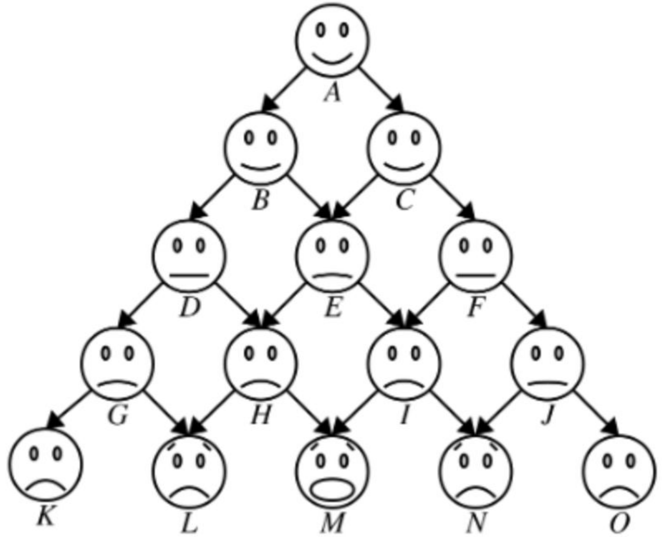
Part 2: Human Pyramids

- How much weight is **M** carrying?



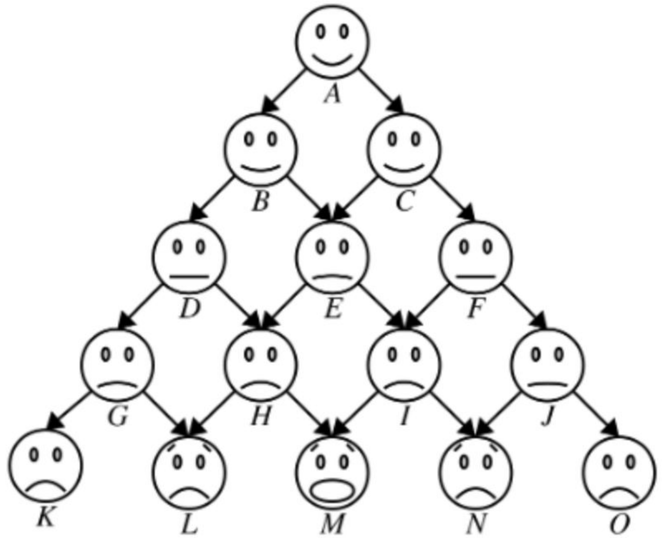
Part 2: Human Pyramids

- How much weight is **M** carrying?
- Sorry I'm just kidding, we are not doing this LOL...



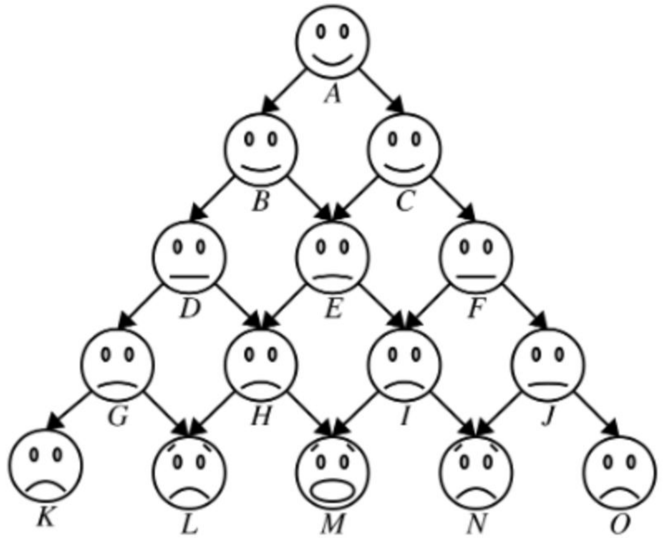
Part 2: Human Pyramids

- How much weight is **M** carrying?
- Sorry I'm just kidding, we are not doing this LOL...
- The point is, this problem fairly computationally heavy after just several rows.



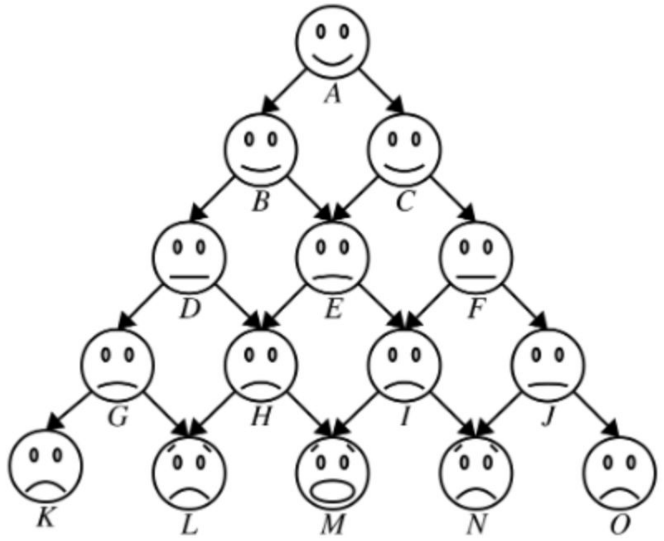
Part 2: Human Pyramids

- Let’s get into the code!



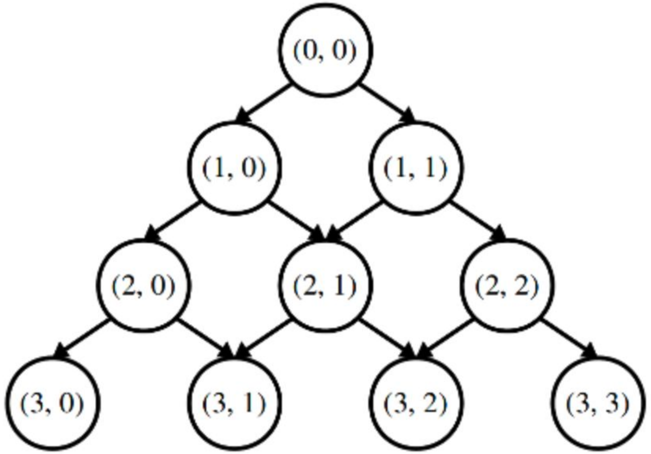
Part 2: Human Pyramids

- Let's get into the code!
- double weightOnBackOf(int row, int col, int pyramidHeight);



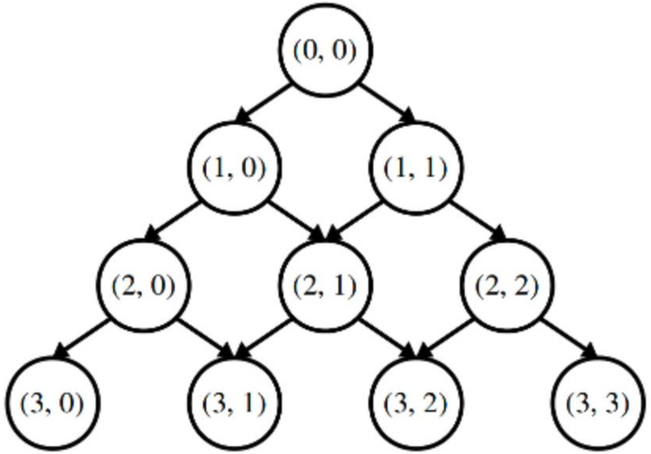
Part 2: Human Pyramids

- Let's get into the code!
- `double weightOnBackOf(int row, int col, int pyramidHeight);`
- Coordinate system (row, col) →


$$\begin{tikzcd}[column sep=tiny]
 & & & (0,0) \arrow[dl] \arrow[dr] & & & \\
 & & (1,0) \arrow[dl] \arrow[dr] & & (1,1) \arrow[dl] \arrow[dr] & & \\
 & (2,0) \arrow[dl] \arrow[dr] & & (2,1) \arrow[dl] \arrow[dr] & & (2,2) \arrow[dl] \arrow[dr] & \\
(3,0) & & (3,1) & & (3,2) & & (3,3)
\end{tikzcd}$$

Part 2: Human Pyramids

- Let’s get into the code!
- double weightOnBackOf(int row, int col, int pyramidHeight);
- Coordinate system (row, col) →
- “pyramidHeight” refers to the number of rows



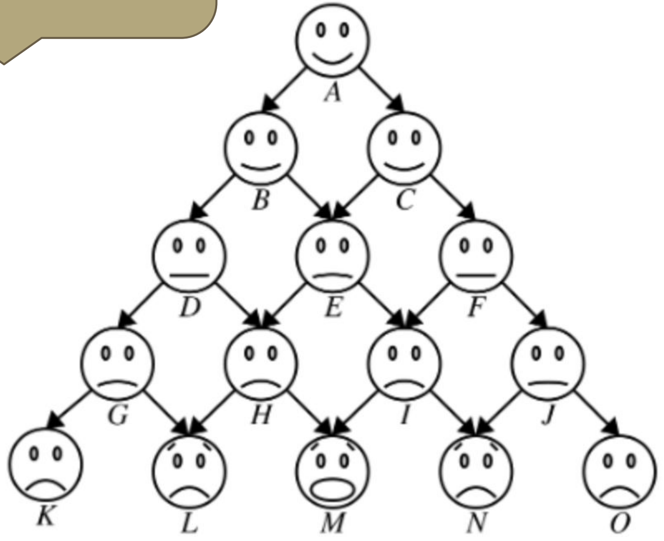
Part 2: Human Pyramids

- Quiz time!

Part 2: Human Pyramids

- Quiz time!

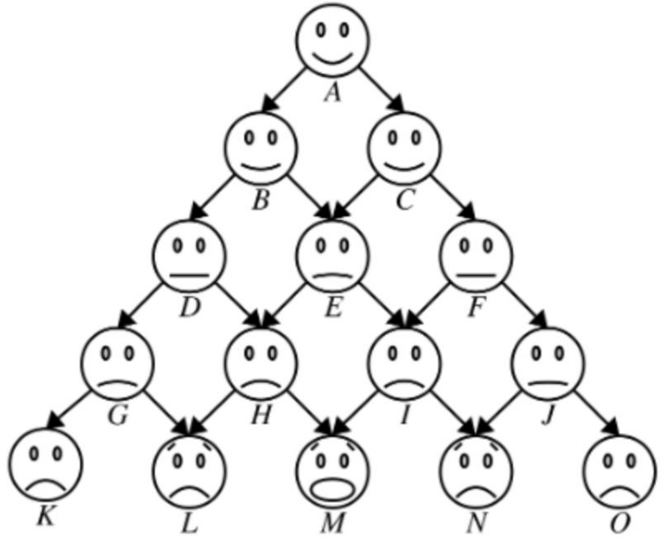
Hey I’m back!



```
double weightOnBackOf(int row, int col, int pyramidHeight);
```

Part 2: Human Pyramids

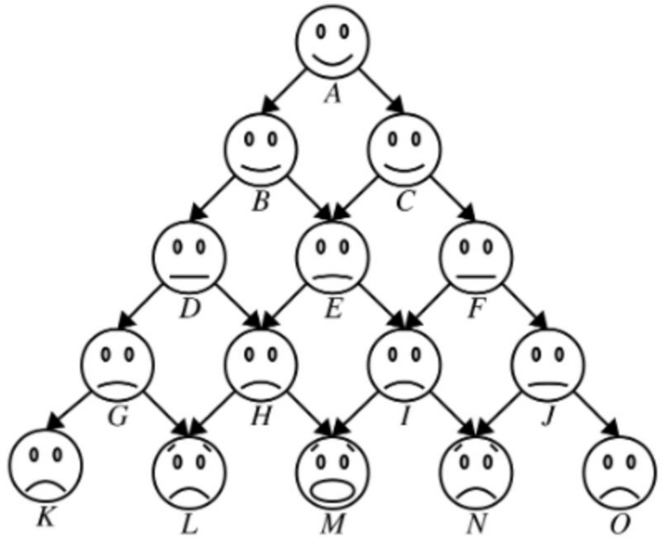
- What is the function call to get the weight on back of **H**?



```
double weightOnBackOf(int row, int col, int pyramidHeight);
```

Part 2: Human Pyramids

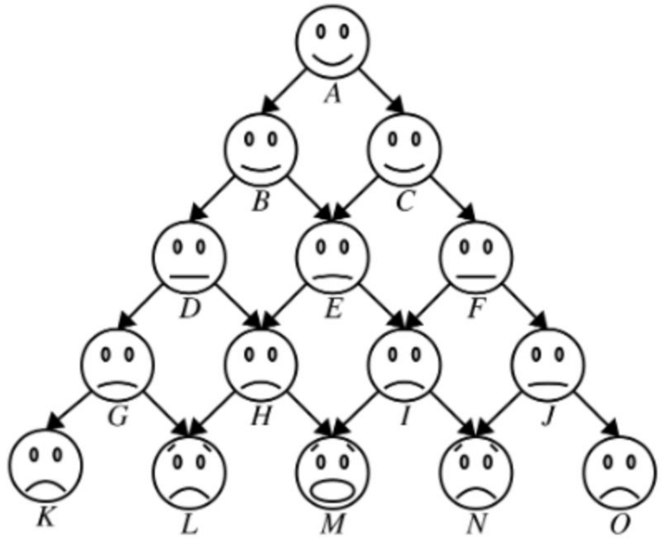
- What is the function call to get the weight on back of **H**?
    - weightOnBackOf(3, 1, 5)



```
double weightOnBackOf(int row, int col, int pyramidHeight);
```

Part 2: Human Pyramids

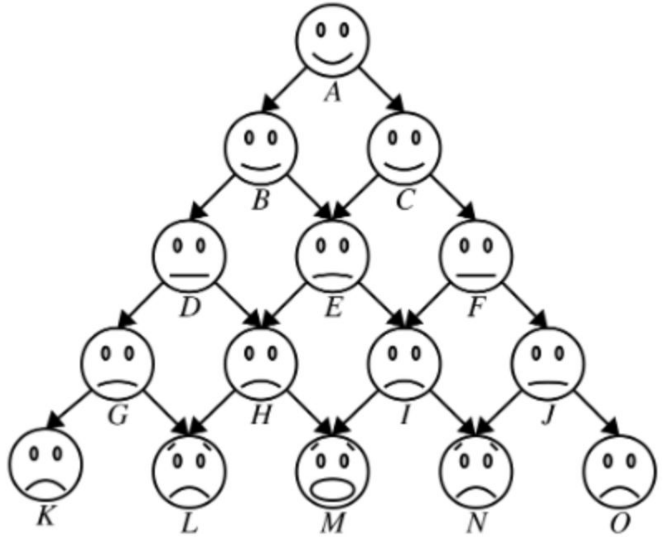
- What is the function call to get the weight on back of **H**?
    - weightOnBackOf(3, 1, 5)
- Which person is weightOnBackOf(4, 3, 5) referring to?



```
double weightOnBackOf(int row, int col, int pyramidHeight);
```

Part 2: Human Pyramids

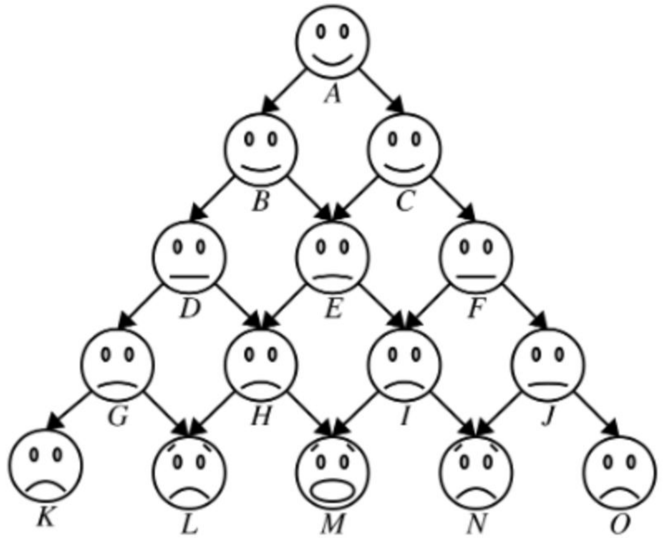
- What is the function call to get the weight on back of **H**?
  - weightOnBackOf(3, 1, 5)
- Which person is weightOnBackOf(4, 3, 5) referring to?
  - Person **N**



```
double weightOnBackOf(int row, int col, int pyramidHeight);
```

Part 2: Human Pyramids

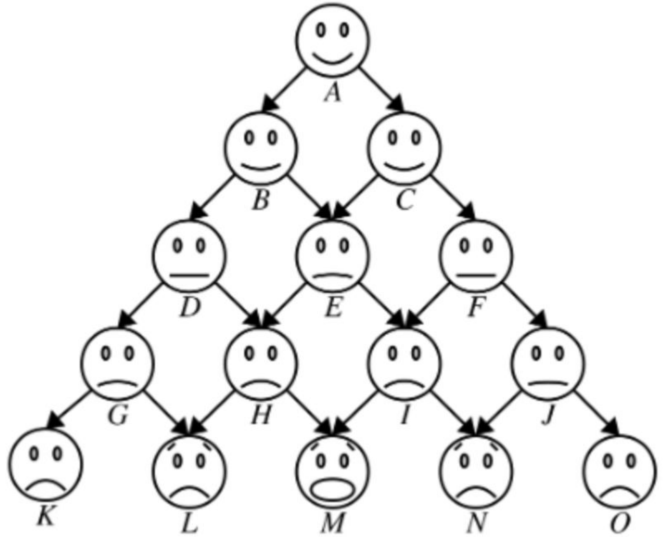
- What is the function call to get the weight on back of **H**?
    - weightOnBackOf(3, 1, 5)
- Which person is weightOnBackOf(4, 3, 5) referring to?
    - Person **N**
- Which person is weightOnBackOf(2, 4, 5) referring to?



```
double weightOnBackOf(int row, int col, int pyramidHeight);
```

Part 2: Human Pyramids

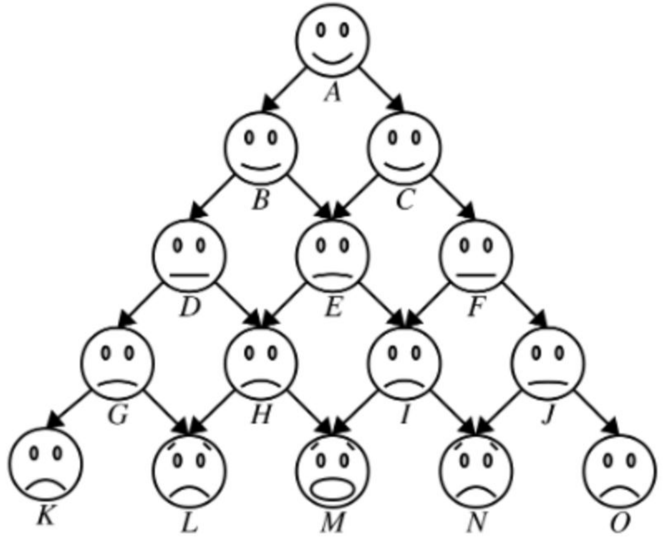
- What is the function call to get the weight on back of **H**?
  - weightOnBackOf(3, 1, 5)
- Which person is weightOnBackOf(4, 3, 5) referring to?
  - Person **N**
- Which person is weightOnBackOf(2, 4, 5) referring to?
  - No one :(



```
double weightOnBackOf(int row, int col, int pyramidHeight);
```

Part 2: Human Pyramids

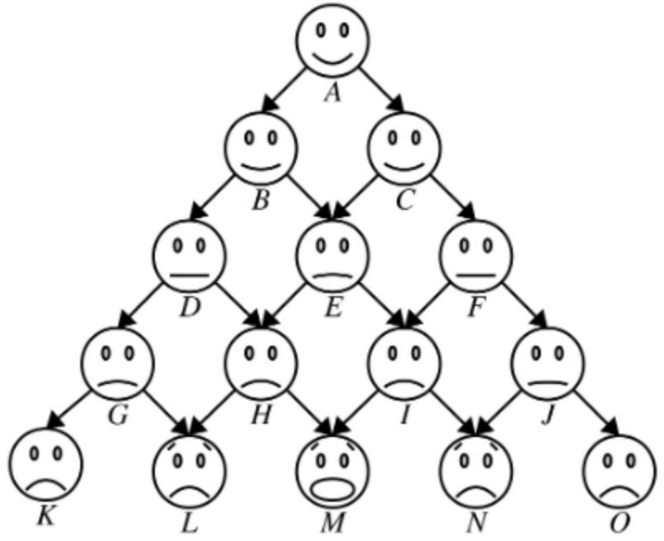
- What is the function call to get the weight on back of **H**?
  - weightOnBackOf(3, 1, 5)
- Which person is weightOnBackOf(4, 3, 5) referring to?
  - Person **N**
- Which person is weightOnBackOf(2, 4, 5) referring to?
  - No one :(
- weightOnBackOf(6, 1, 5)?



```
double weightOnBackOf(int row, int col, int pyramidHeight);
```

Part 2: Human Pyramids

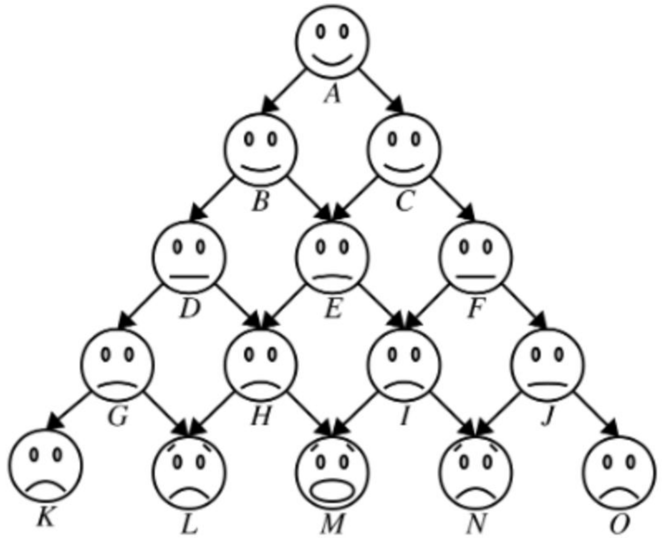
- What is the function call to get the weight on back of **H**?
    - weightOnBackOf(3, 1, 5)
- Which person is weightOnBackOf(4, 3, 5) referring to?
    - Person **N**
- Which person is weightOnBackOf(2, 4, 5) referring to?
    - No one :(
- weightOnBackOf(6, 1, 5)?
    - No one :(



```
double weightOnBackOf(int row, int col, int pyramidHeight);
```

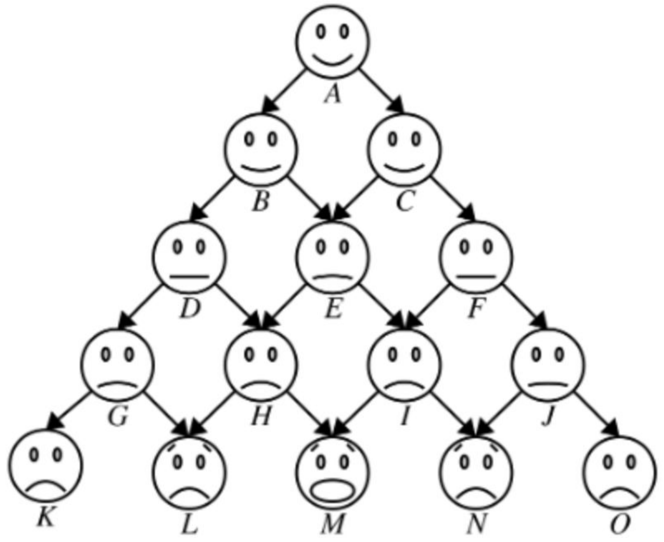
Part 2: Human Pyramids

- Some implementation thoughts:



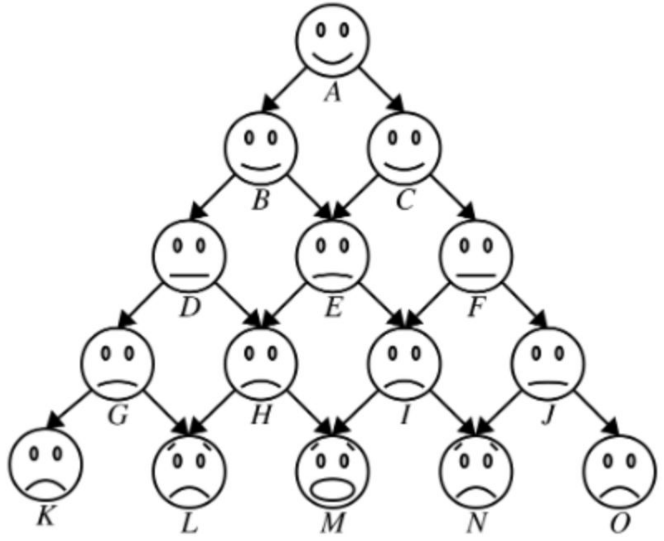
Part 2: Human Pyramids

- Some implementation thoughts:
  - Throw an error for invalid function calls (think through the possible cases thoroughly).



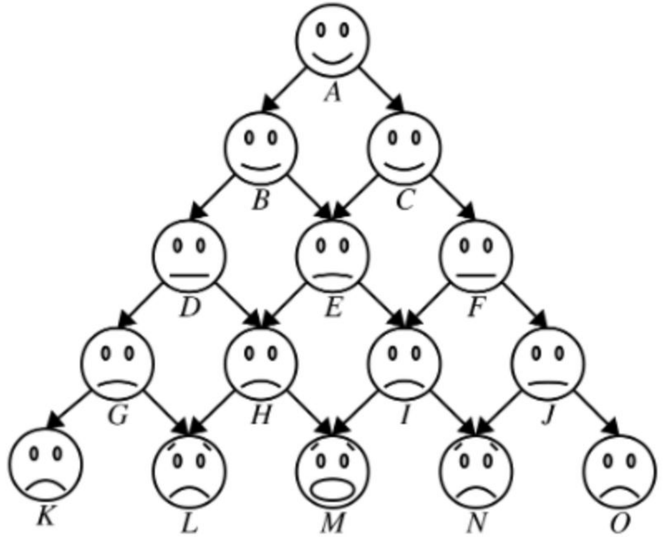
Part 2: Human Pyramids

- Some implementation thoughts:
  - Throw an error for invalid function calls (think through the possible cases thoroughly).
  - The top person carries no weight.



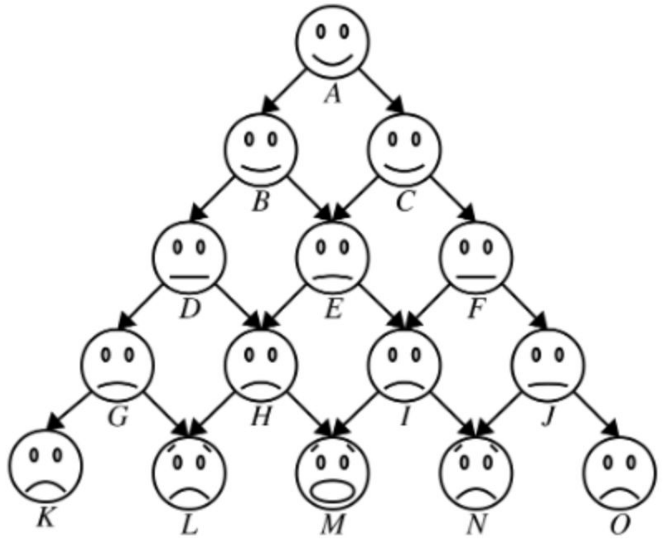
Part 2: Human Pyramids

- Some implementation thoughts:
  - Throw an error for invalid function calls (think through the possible cases thoroughly).
  - The top person carries no weight.
  - Some people are directly supporting two people, some people are directly supporting one person. How should you distinguish between the two cases?



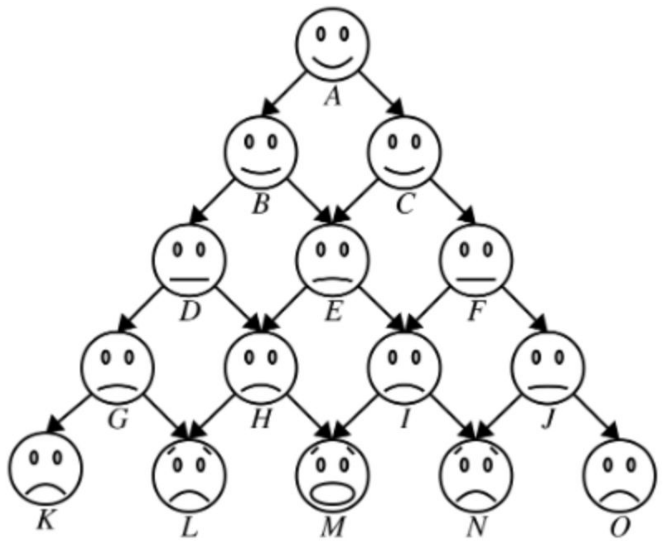
Part 2: Human Pyramids

- Some implementation thoughts:
  - Throw an error for invalid function calls (think through the possible cases thoroughly).
  - The top person carries no weight.
  - Some people are directly supporting two people, some people are directly supporting one person. How should you distinguish between the two cases?
  - Remember that the function returns a **double**!



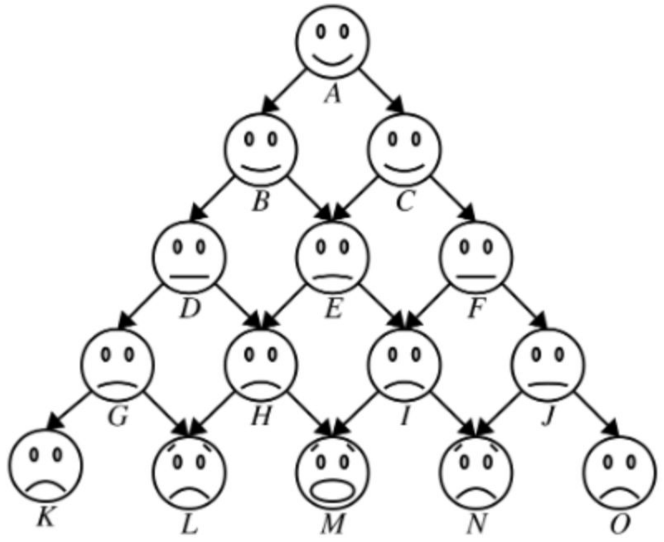
Part 2: Human Pyramids

- Some implementation thoughts:
    - Throw an error for invalid function calls (think through the possible cases thoroughly).
    - The top person carries no weight.
    - Some people are directly supporting two people, some people are directly supporting one person. How should you distinguish between the two cases?
    - Remember that the function returns a **double**!
    - Test your solution before moving on!



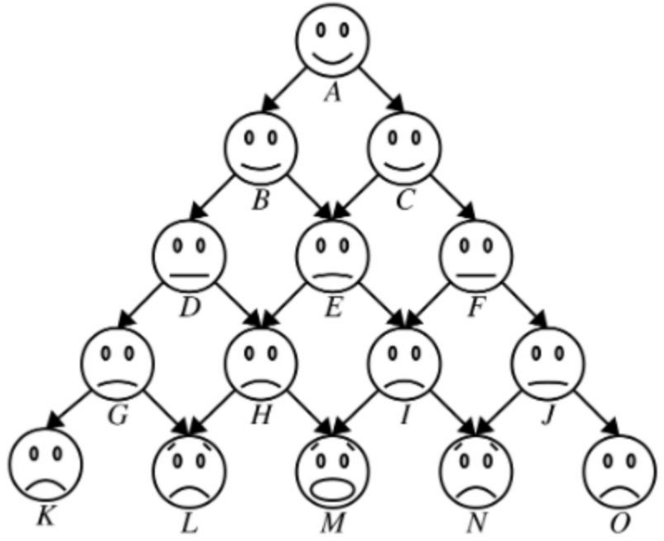
Part 2: Human Pyramids

- Questions before we move on to part 2 of part 2?



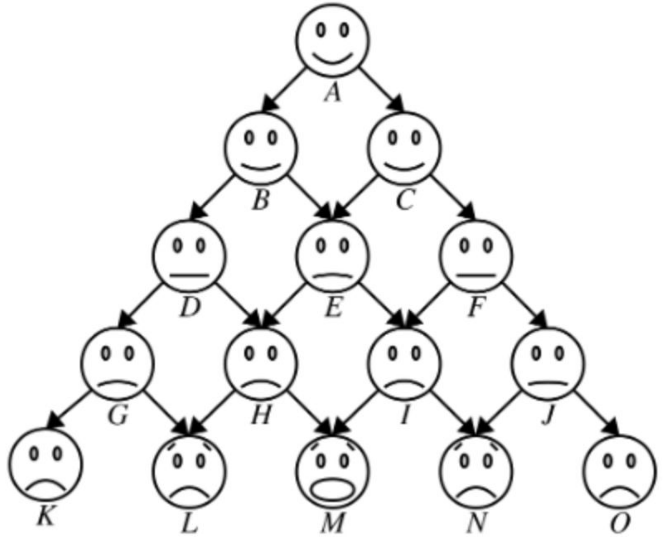
Part 2: Human Pyramids - Milestone 2

- Let's examine a ***HUGE*** efficiency flaw in our first implementation.



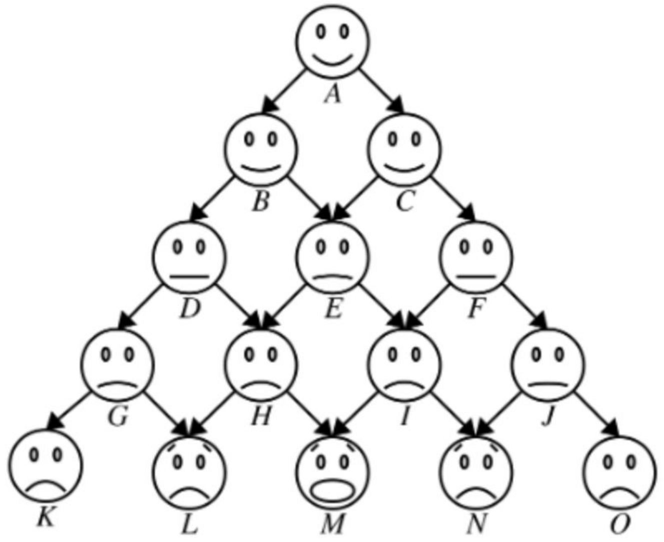
Part 2: Human Pyramids - Milestone 2

- Let’s examine a ***HUGE*** efficiency flaw in our first implementation.
- Say we’re interested in finding out how much weight **M** is carrying.



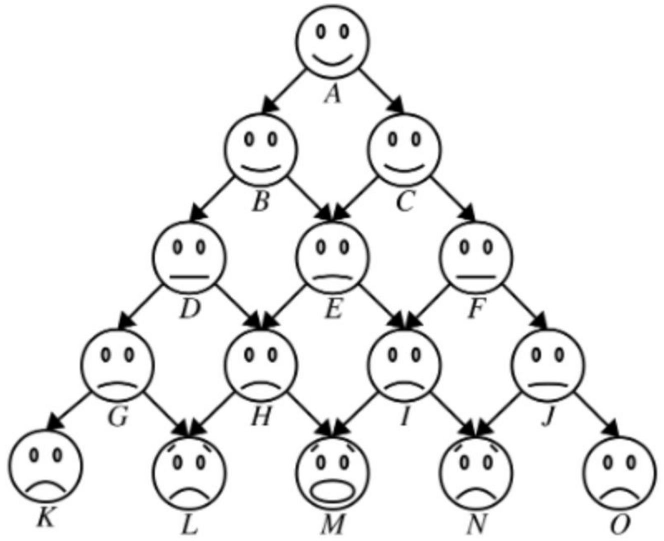
Part 2: Human Pyramids - Milestone 2

- Let's examine a ***HUGE*** efficiency flaw in our first implementation.
- Say we're interested in finding out how much weight **M** is carrying.
- How many times do we need to call the function with respect to **M**?



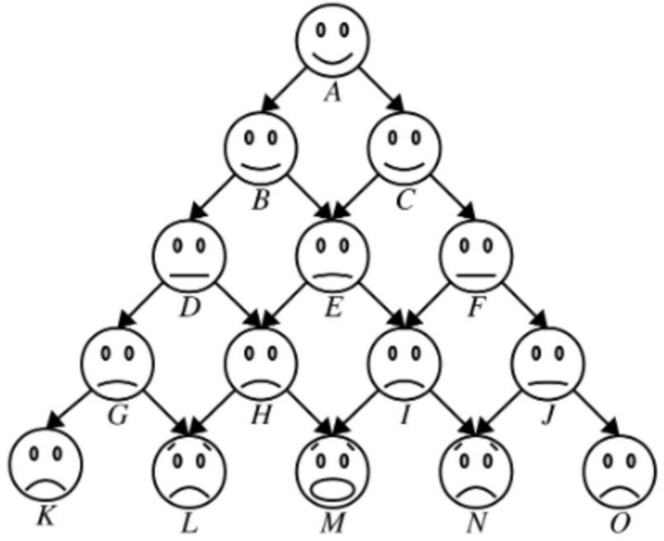
Part 2: Human Pyramids - Milestone 2

- Let's examine a ***HUGE*** efficiency flaw in our first implementation.
- Say we're interested in finding out how much weight **M** is carrying.
- How many times do we need to call the function with respect to **M**?
    - Just once



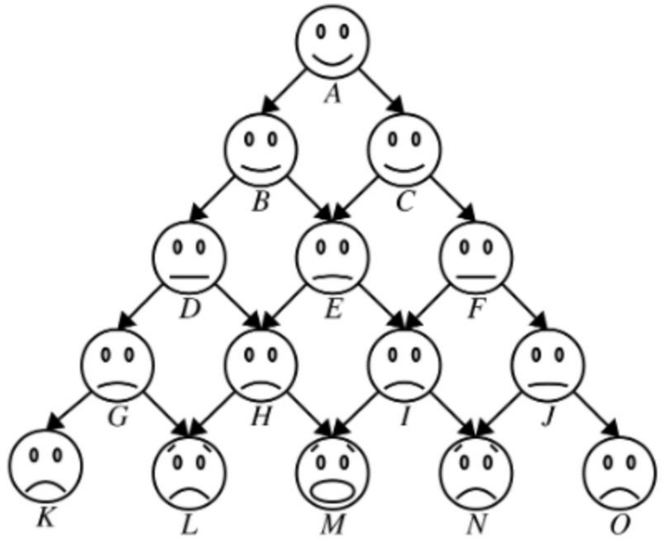
Part 2: Human Pyramids - Milestone 2

- Let's examine a ***HUGE*** efficiency flaw in our first implementation.
- Say we're interested in finding out how much weight **M** is carrying.
- How many times do we need to call the function with respect to **M**?
    - Just once
- What about **H**?



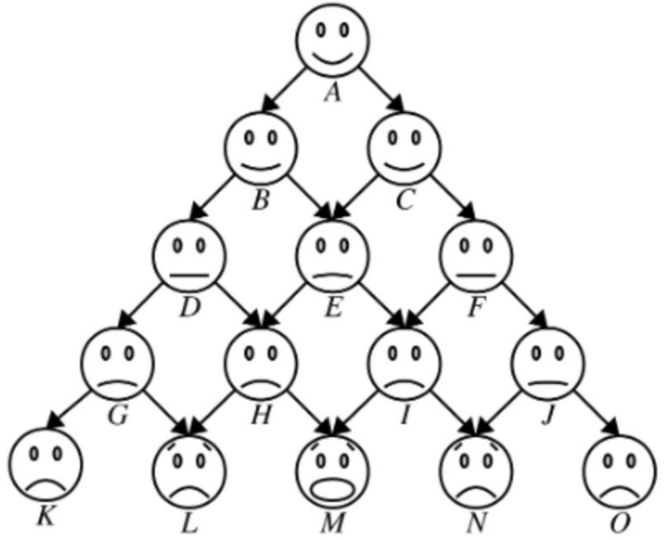
Part 2: Human Pyramids - Milestone 2

- Let's examine a ***HUGE*** efficiency flaw in our first implementation.
- Say we're interested in finding out how much weight **M** is carrying.
- How many times do we need to call the function with respect to **M**?
    - Just once
- What about **H**?
    - Once



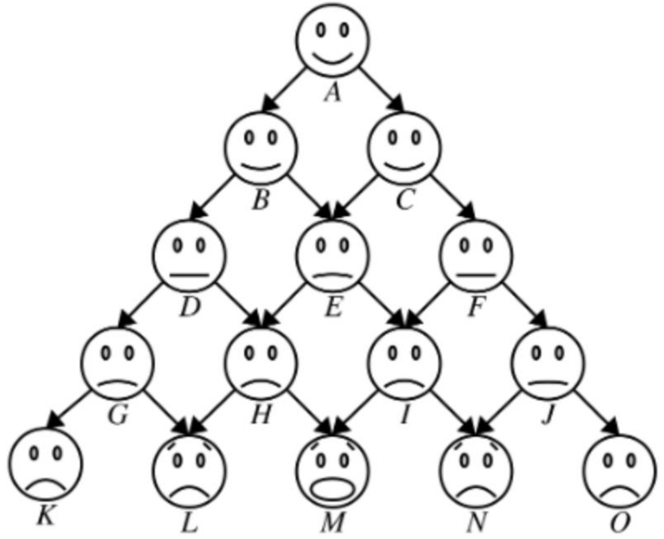
Part 2: Human Pyramids - Milestone 2

- Let's examine a ***HUGE*** efficiency flaw in our first implementation.
- Say we're interested in finding out how much weight **M** is carrying.
- How many times do we need to call the function with respect to **M**?
    - Just once
- What about **H**?
    - Once
- **E**?



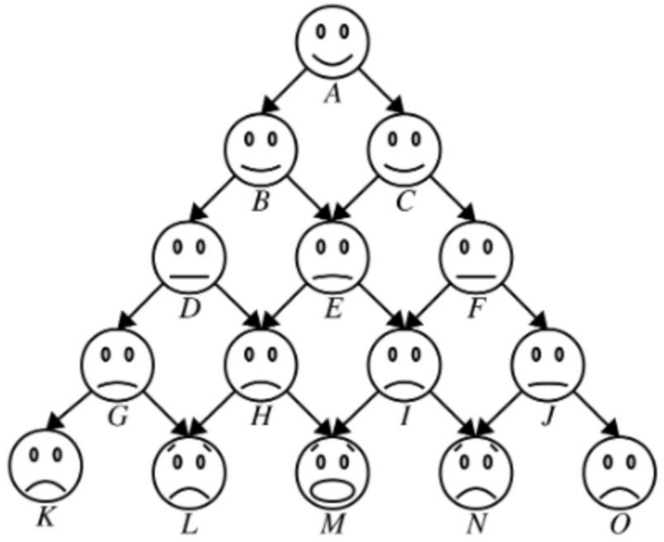
Part 2: Human Pyramids - Milestone 2

- Let’s examine a ***HUGE*** efficiency flaw in our first implementation.
- Say we’re interested in finding out how much weight **M** is carrying.
- How many times do we need to call the function with respect to **M**?
  - Just once
- What about **H**?
  - Once
- **E**?
  - Twice



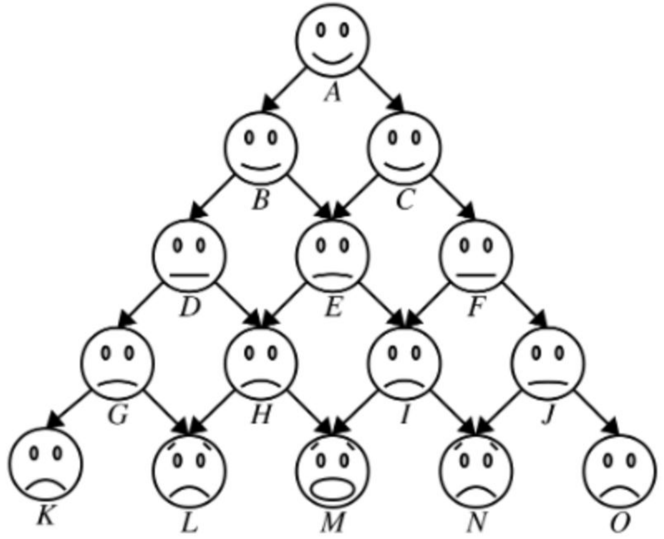
Part 2: Human Pyramids - Milestone 2

- Let's examine a ***HUGE*** efficiency flaw in our first implementation.
- Say we're interested in finding out how much weight **M** is carrying.
- How many times do we need to call the function with respect to **M**?
  - Just once
- What about **H**?
  - Once
- **E**?
  - Twice
- And **A** will be called 6 times!



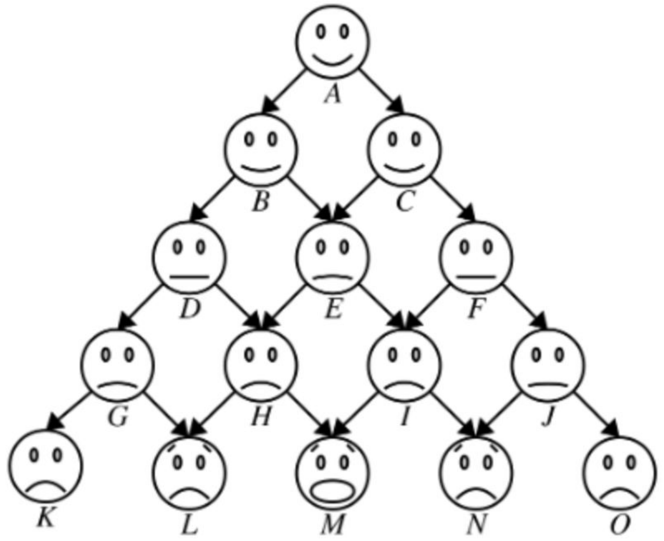
Part 2: Human Pyramids - Milestone 2

- This actually resembles a Pascal's Triangle (off topic, just slightly)



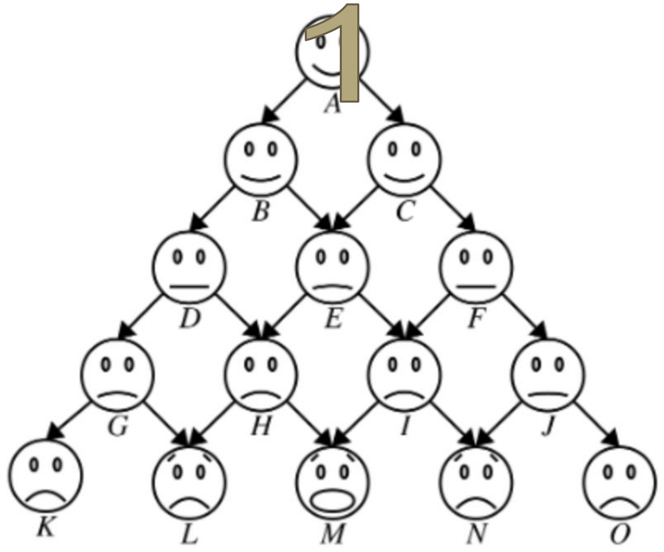
Part 2: Human Pyramids - Milestone 2

- This actually resembles a Pascal's Triangle (off topic, just slightly)
- Let the number on a person denote how many times this person needs to call **A**.



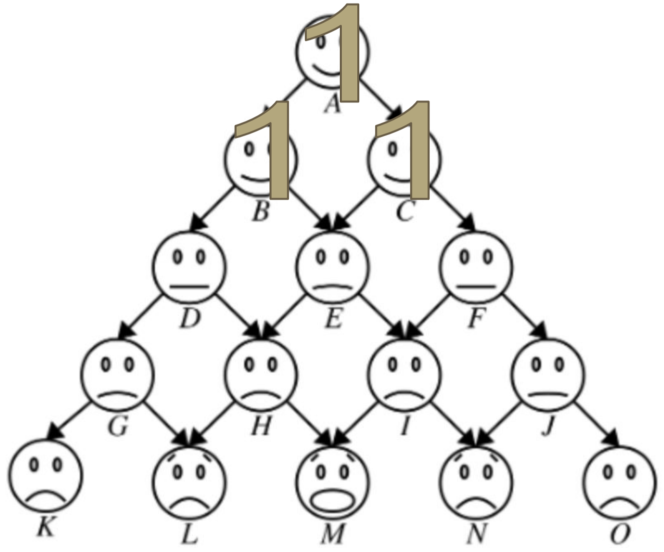
Part 2: Human Pyramids - Milestone 2

- This actually resembles a Pascal's Triangle (off topic, just slightly)
- Let the number on a person denote how many times this person needs to call **A**.



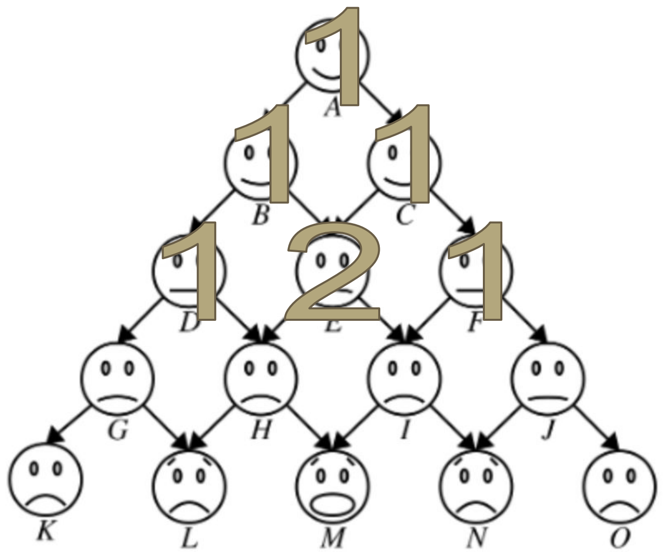
Part 2: Human Pyramids - Milestone 2

- This actually resembles a Pascal's Triangle (off topic, just slightly)
- Let the number on a person denote how many times this person needs to call **A**.



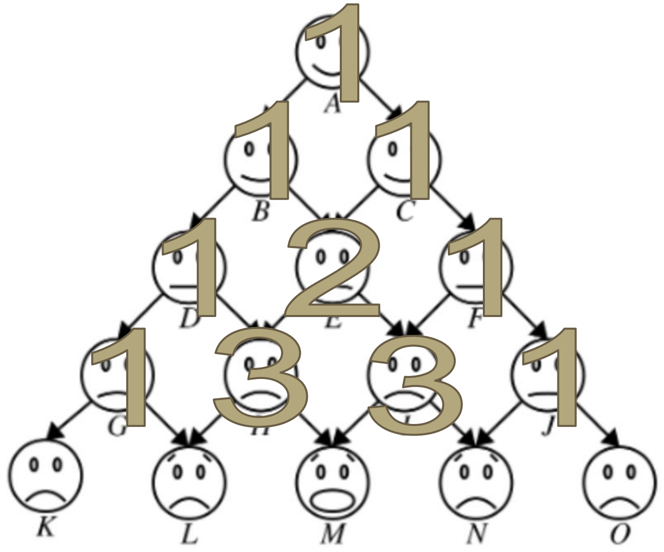
Part 2: Human Pyramids - Milestone 2

- This actually resembles a Pascal's Triangle (off topic, just slightly)
- Let the number on a person denote how many times this person needs to call **A**.



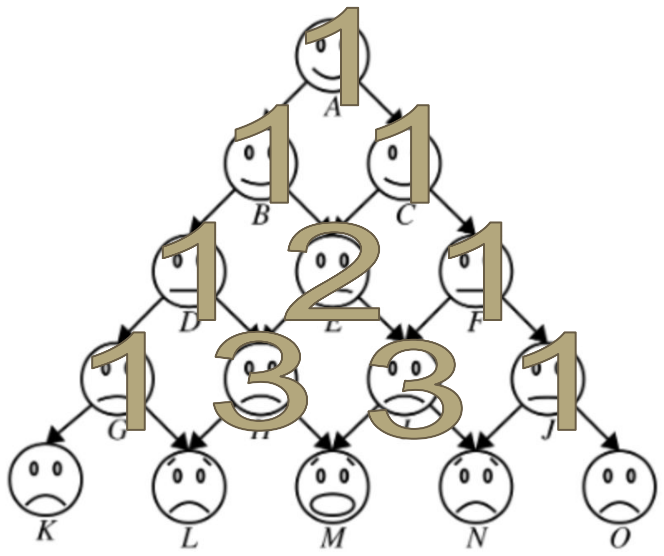
Part 2: Human Pyramids - Milestone 2

- This actually resembles a Pascal's Triangle (off topic, just slightly)
- Let the number on a person denote how many times this person needs to call **A**.



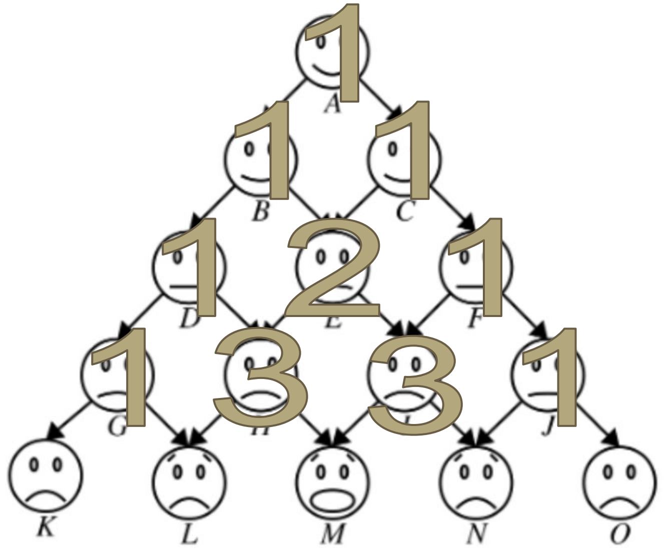
Part 2: Human Pyramids - Milestone 2

- This actually resembles a Pascal's Triangle (off topic, just slightly)
- Let the number on a person denote how many times this person needs to call **A**.
- You are expecting the last row filled out, but I'm out of animation budget :|



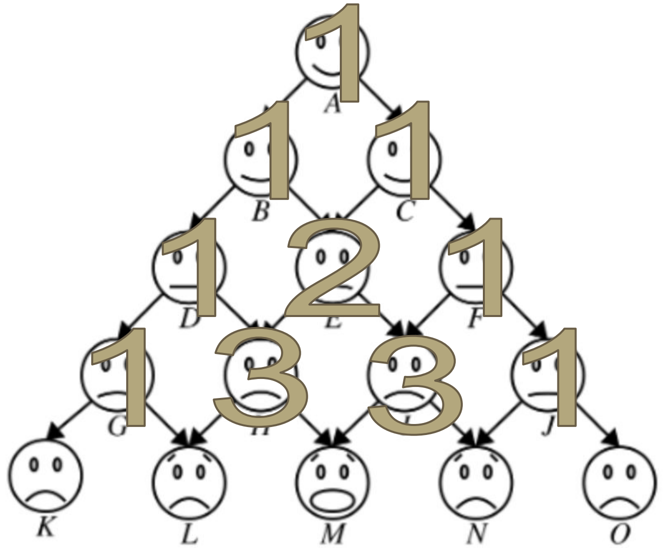
Part 2: Human Pyramids - Milestone 2

- Observation: As we get further into the recursive stack, the more copies of the **SAME** function calls we will be creating.



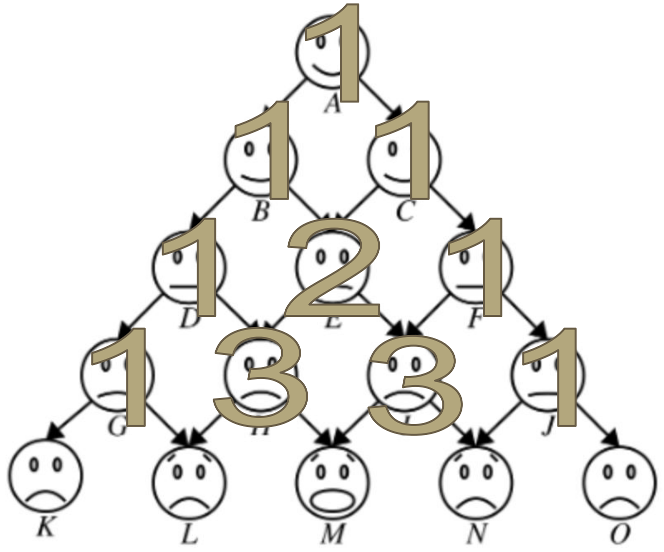
Part 2: Human Pyramids - Milestone 2

- Observation: As we get further into the recursive stack, the more copies of the **SAME** function calls we will be creating.
- Just the 10th row and we have surpassed 100 in the Pascal's Triangle.



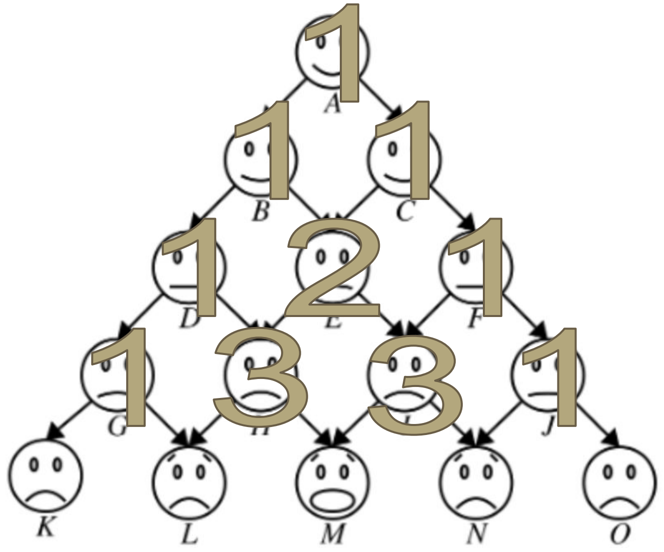
Part 2: Human Pyramids - Milestone 2

- Observation: As we get further into the recursive stack, the more copies of the **SAME** function calls we will be creating.
- Just the 10th row and we have surpassed 100 in the Pascal's Triangle.
- 14th row and we have surmounted 1,000.



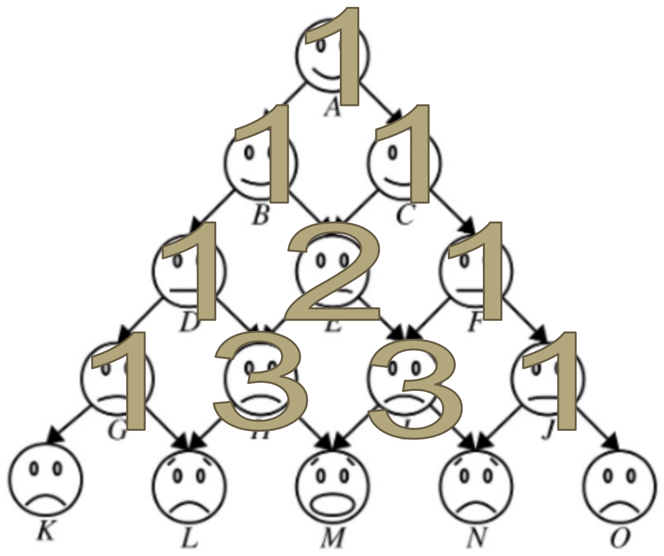
Part 2: Human Pyramids - Milestone 2

- Observation: As we get further into the recursive stack, the more copies of the **SAME** function calls we will be creating.
- Just the 10th row and we have surpassed 100 in the Pascal's Triangle.
- 14th row and we have surmounted 1,000.
- 17th row > 10,000.



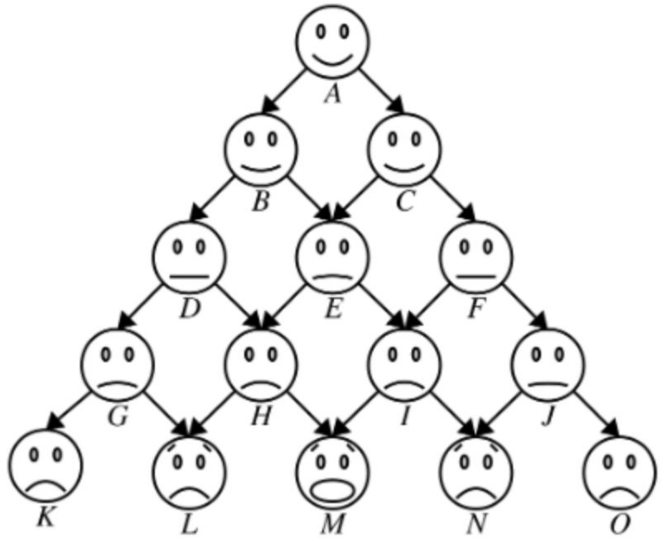
Part 2: Human Pyramids - Milestone 2

- Observation: As we get further into the recursive stack, the more copies of the **SAME** function calls we will be creating.
- Just the 10th row and we have surpassed 100 in the Pascal's Triangle.
- 14th row and we have surmounted 1,000.
- 17th row > 10,000.
- Conclusion: This grows really really fast!



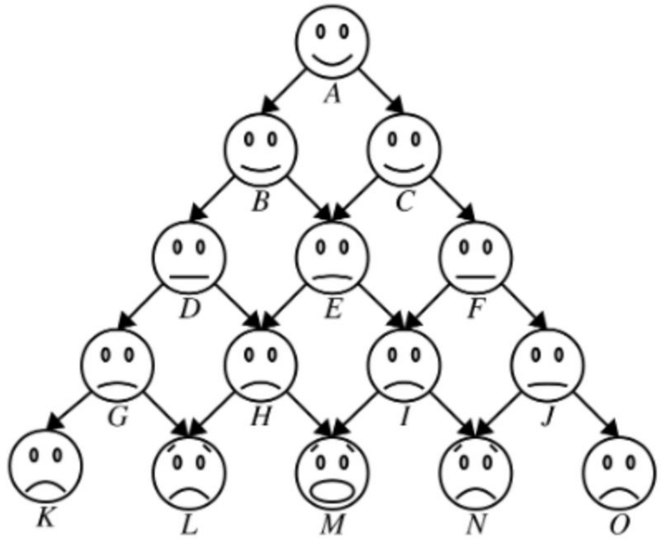
Part 2: Human Pyramids - Milestone 2

- We need a solution.



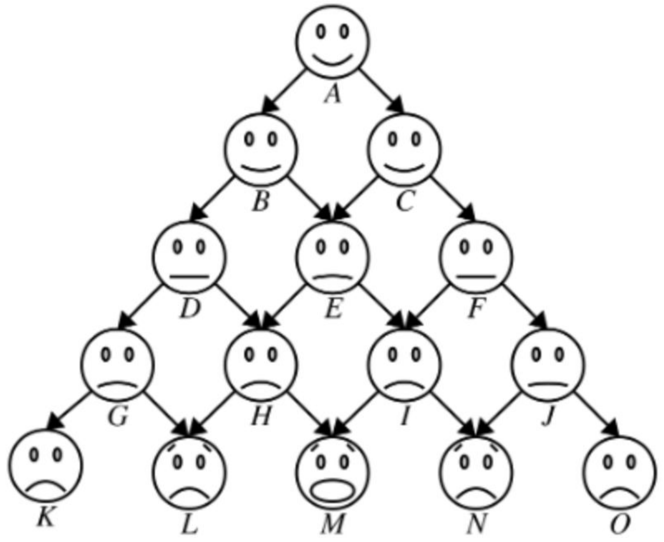
Part 2: Human Pyramids - Milestone 2

- We need a solution.



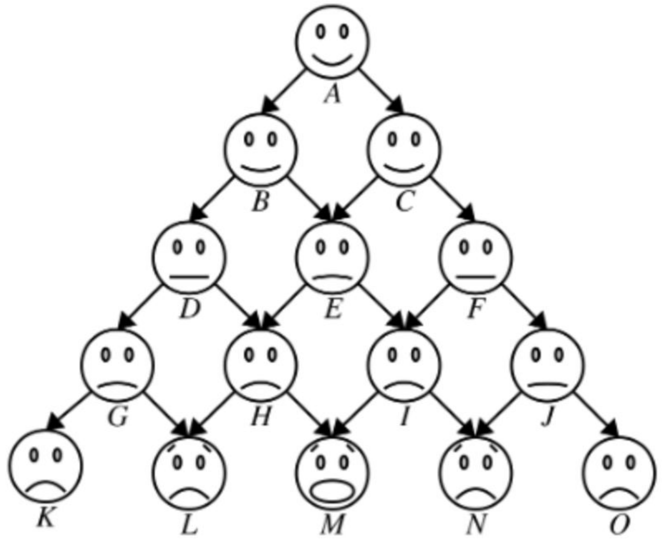
Part 2: Human Pyramids

- How much weight is **E** carrying?
  - To save some work, we know from the previous example that **B** yields 80 lbs (half of **B**'s weight) + 40 lbs (half of the weight **B** is carrying) = 120 lbs.



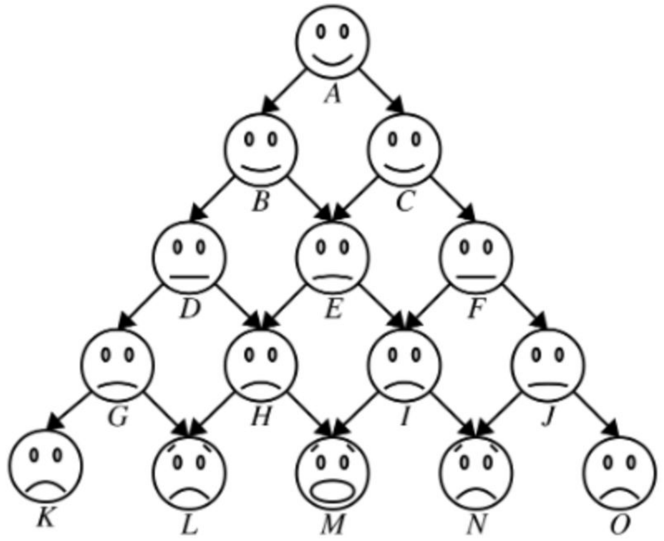
Part 2: Human Pyramids

- How much weight is **E** carrying?
  - To save some work, we know from the previous example that **B** yields 80 lbs (half of **B**'s weight) + 40 lbs (half of the weight **B** is carrying) = 120 lbs.



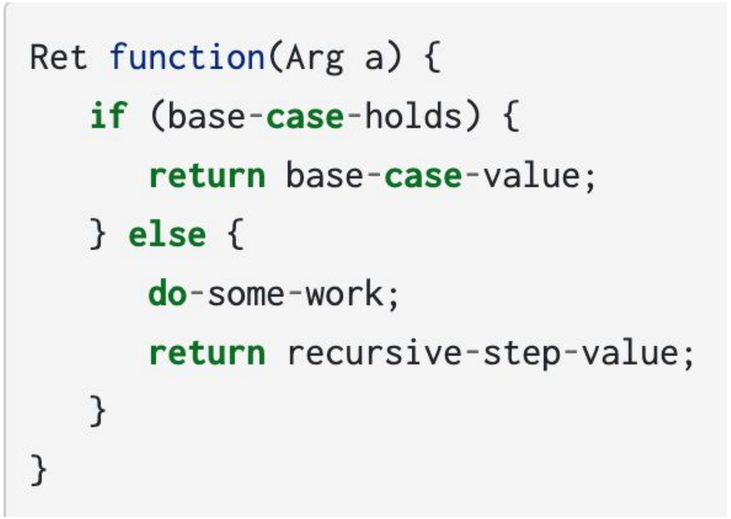
Part 2: Human Pyramids - Milestone 2

- ***Memoization***: using an auxiliary table to keep track of all the recursive calls that have been made before and what value was returned for each of them.



Part 2: Human Pyramids - Milestone 2

Standard Recursion:



```
Ret function(Arg a) {
    if (base-case-holds) {
        return base-case-value;
    } else {
        do-some-work;
        return recursive-step-value;
    }
}
```

Part 2: Human Pyramids - Milestone 2

With Memoization:

```
Ret functionRec(Arg a, Table& table) {
   if (base-case-holds) {
      return base-case-value;
   } else if (table contains a) {
      return table[a];
   } else {
      do-some-work;
      table[a] = recursive-step-value;
      return recursive-step-value;
   }
}

Ret function(Arg a) {
    Table table;
    return functionRec(a, table);
}
```

Part 2: Human Pyramids - Milestone 2

With Memoization:

```
Ret functionRec(Arg a, Table& table) {
   if (base-case-holds) {
      return base-case-value;
   } else if (table contains a) {
      return table[a];
   } else {
      do-some-work;
      table[a] = recursive-step-value;
      return recursive-step-value;
   }
}

Ret function(Arg a) {
    Table table;
    return functionRec(a, table);
}
```

Wrapper function

Part 2: Human Pyramids - Milestone 2

With Memoization:

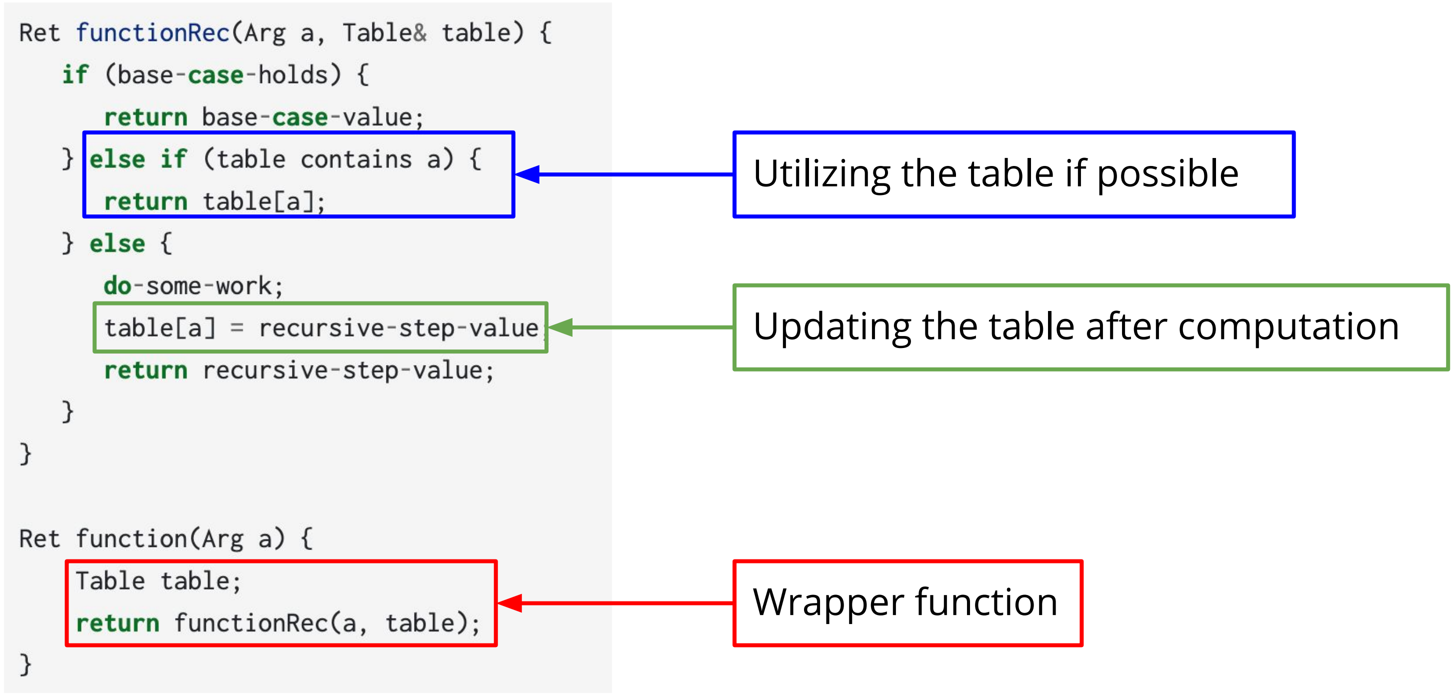
```
Ret functionRec(Arg a, Table& table) {
   if (base-case-holds) {
      return base-case-value;
   } else if (table contains a) {
      return table[a];
   } else {
      do-some-work;
      table[a] = recursive-step-value;
      return recursive-step-value;
   }
}

Ret function(Arg a) {
    Table table;
    return functionRec(a, table);
}
```

Utilizing the table if possible

Wrapper function

Part 2: Human Pyramids - Milestone 2

With Memoization:



```
Ret functionRec(Arg a, Table& table) {
    if (base-case-holds) {
        return base-case-value;
    } else if (table contains a) {
        return table[a];
    } else {
        do-some-work;
        table[a] = recursive-step-value
        return recursive-step-value;
    }
}

Ret function(Arg a) {
    Table table;
    return functionRec(a, table);
}
```

Utilizing the table if possible
Updating the table after computation
Wrapper function

Part 2: Human Pyramids - Milestone 2

- Note that there isn't a variable of type **Table**

```
Ret functionRec(Arg a, Table& table) {
   if (base-case-holds) {
      return base-case-value;
   } else if (table contains a) {
      return table[a];
   } else {
      do-some-work;
      table[a] = recursive-step-value;
      return recursive-step-value;
   }
}

Ret function(Arg a) {
    Table table;
    return functionRec(a, table);
}
```

Part 2: Human Pyramids - Milestone 2

- Note that there isn't a variable of type **Table**
    - **Table** is generally referring some data structure we can use to store previous computations.

```
Ret functionRec(Arg a, Table& table) {
   if (base-case-holds) {
      return base-case-value;
   } else if (table contains a) {
      return table[a];
   } else {
      do-some-work;
      table[a] = recursive-step-value;
      return recursive-step-value;
   }
}


Ret function(Arg a) {
    Table table;
    return functionRec(a, table);
}
```

Part 2: Human Pyramids - Milestone 2

- Note that there isn't a variable of type **Table**
  - **Table** is generally referring some data structure we can use to store previous computations.
  - Its type can differ from problem to problem.

```
Ret functionRec(Arg a, Table& table) {
   if (base-case-holds) {
      return base-case-value;
   } else if (table contains a) {
      return table[a];
   } else {
      do-some-work;
      table[a] = recursive-step-value;
      return recursive-step-value;
   }
}

Ret function(Arg a) {
    Table table;
    return functionRec(a, table);
}
```

Part 2: Human Pyramids - Milestone 2

- Note that there isn't a variable of type **Table**
    - **Table** is generally referring some data structure we can use to store previous computations.
    - Its type can differ from problem to problem.
- Question to ask yourself: What should the type of **Table** be such that:

```
Ret functionRec(Arg a, Table& table) {
    if (base-case-holds) {
        return base-case-value;
    } else if (table contains a) {
        return table[a];
    } else {
        do-some-work;
        table[a] = recursive-step-value;
        return recursive-step-value;
    }
}

Ret function(Arg a) {
    Table table;
    return functionRec(a, table);
}
```

Part 2: Human Pyramids - Milestone 2

- Note that there isn't a variable of type **Table**
  - **Table** is generally referring some data structure we can use to store previous computations.
  - Its type can differ from problem to problem.
- Question to ask yourself: What should the type of **Table** be such that:
  - It is easy and fast to look up necessary values.

```
Ret functionRec(Arg a, Table& table) {
    if (base-case-holds) {
        return base-case-value;
    } else if (table contains a) {
        return table[a];
    } else {
        do-some-work;
        table[a] = recursive-step-value;
        return recursive-step-value;
    }
}

Ret function(Arg a) {
    Table table;
    return functionRec(a, table);
}
```

Part 2: Human Pyramids - Milestone 2

- Note that there isn't a variable of type **Table**
    - **Table** is generally referring some data structure we can use to store previous computations.
    - Its type can differ from problem to problem.
- Question to ask yourself: What should the type of **Table** be such that:
    - It is easy and fast to look up necessary values.
    - Can be updated during program execution.

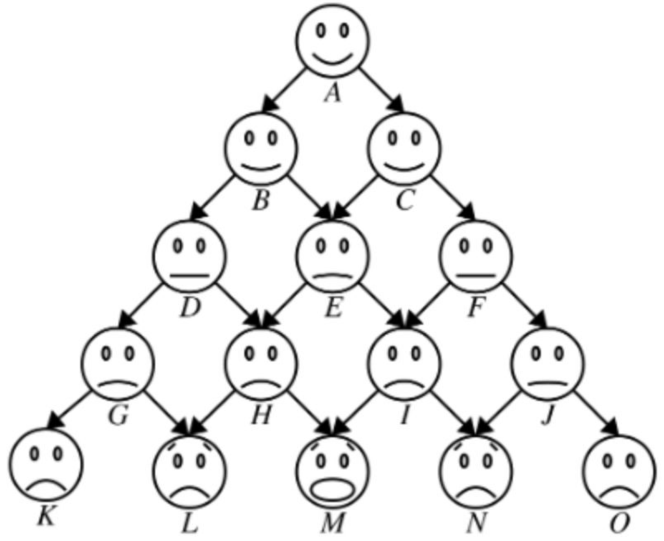
```
Ret functionRec(Arg a, Table& table) {
    if (base-case-holds) {
        return base-case-value;
    } else if (table contains a) {
        return table[a];
    } else {
        do-some-work;
        table[a] = recursive-step-value;
        return recursive-step-value;
    }
}

Ret function(Arg a) {
    Table table;
    return functionRec(a, table);
}
```

Part 2: Human Pyramids - Milestone 2

- Note that there isn't a variable of type **Table**
    - **Table** is generally referring some data structure we can use to store previous computations.
    - Its type can differ from problem to problem.
- Question to ask yourself: What should the type of **Table** be such that:
    - It is easy and fast to look up necessary values.
    - Can be updated during program execution.
    - Is efficient and elegant.

```
Ret functionRec(Arg a, Table& table) {
    if (base-case-holds) {
        return base-case-value;
    } else if (table contains a) {
        return table[a];
    } else {
        do-some-work;
        table[a] = recursive-step-value;
        return recursive-step-value;
    }
}

Ret function(Arg a) {
    Table table;
    return functionRec(a, table);
}
```

Questions about Part 2?



Context Switch Time!

Definition: A *context switch* occurs when a code routine is “switched off” the CPU so that another routine can begin / resume.

Part 3: What Are YOU Doing?

- Have you ever considered an iconic phrase like:

Part 3: What Are YOU Doing?

- Have you ever considered an iconic phrase like:

“To be or not to be”

Part 3: What Are YOU Doing?

- Have you ever considered an iconic phrase like:

“To be or not to be”

and wondered what it would be like to capitalize every possible subset of the words?

Part 3: What Are YOU Doing?

- Have you ever considered an iconic phrase like:

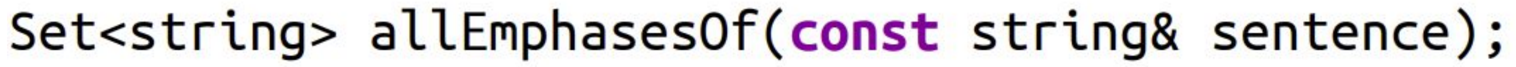
  “*To be or not to be*”

  and wondered what it would be like to capitalize every possible subset of the words?

  Me neither!

Part 3: What Are YOU Doing?

- Nonetheless, it's your job to implement the following function:



```
Set<string> allEmphasesOf(const string& sentence);
```

Part 3: What Are YOU Doing?

- Nonetheless, it's your job to implement the following function:

```
Set<string> allEmphasesOf(const string& sentence);
```

- Given an arbitrary `sentence`, you'll need to return all possible combinations of emphases like so:

Part 3: What Are YOU Doing?

- Nonetheless, it's your job to implement the following function:

```
Set<string> allEmphasesOf(const string& sentence);
```

- Given an arbitrary `sentence`, you'll need to return all possible combinations of emphases like so:

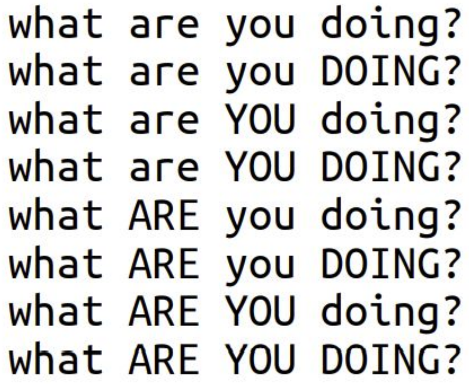
```
what are you doing?
what are you DOING?
what are YOU doing?
what are YOU DOING?
what ARE you doing?
what ARE you DOING?
what ARE YOU doing?
what ARE YOU DOING?
```

What are you doing?

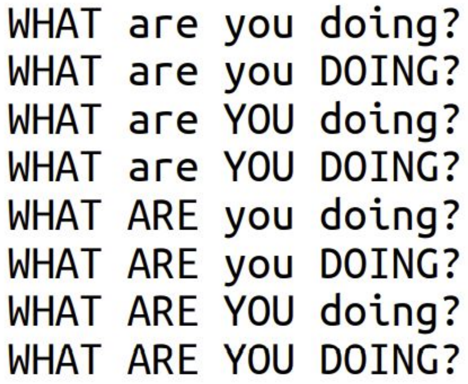
```
WHAT are you doing?
WHAT are you DOING?
WHAT are YOU doing?
WHAT are YOU DOING?
WHAT ARE you doing?
WHAT ARE you DOING?
WHAT ARE YOU doing?
WHAT ARE YOU DOING?
```

Part 3: What Are YOU Doing?

- Let’s think about this critically for a second:



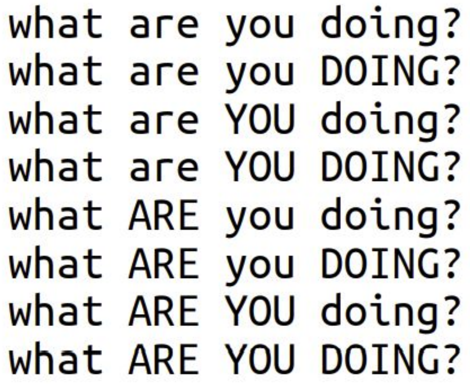
```
what are you doing?
what are you DOING?
what are YOU doing?
what are YOU DOING?
what ARE you doing?
what ARE you DOING?
what ARE YOU doing?
what ARE YOU DOING?
```



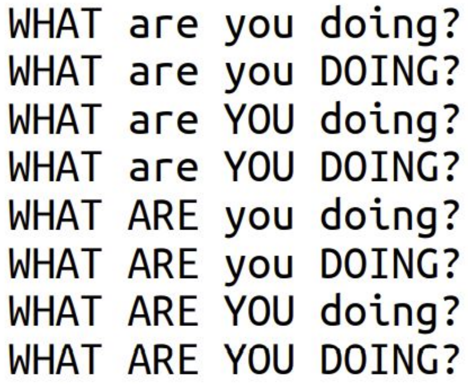
```
WHAT are you doing?
WHAT are you DOING?
WHAT are YOU doing?
WHAT are YOU DOING?
WHAT ARE you doing?
WHAT ARE you DOING?
WHAT ARE YOU doing?
WHAT ARE YOU DOING?
```

Part 3: What Are YOU Doing?

- Let’s think about this critically for a second:



```
what are you doing?
what are you DOING?
what are YOU doing?
what are YOU DOING?
what ARE you doing?
what ARE you DOING?
what ARE YOU doing?
what ARE YOU DOING?
```

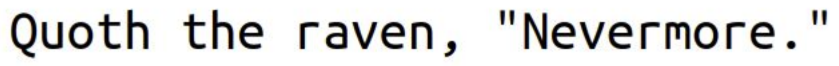


```
WHAT are you doing?
WHAT are you DOING?
WHAT are YOU doing?
WHAT are YOU DOING?
WHAT ARE you doing?
WHAT ARE you DOING?
WHAT ARE YOU doing?
WHAT ARE YOU DOING?
```

- It looks like at the upper left, *all tokens are lowercase,* and on the bottom right, all *tokens are uppercase*! Hmm, does this look like patterns you’ve seen before?

Part 3: What Are YOU Doing?

- Granted, you might sometimes have stranger-looking questions like this:



```
Quoth the raven, "Nevermore."
```

Part 3: What Are YOU Doing?

- Granted, you might sometimes have stranger-looking questions like this:

```
Quoth the raven, "Nevermore."
```

- We've provided you with the following function to help **tokenize** the given sentence.

```
Vector<string> tokenize(const string& sentence);
```

Part 3: What Are YOU Doing?

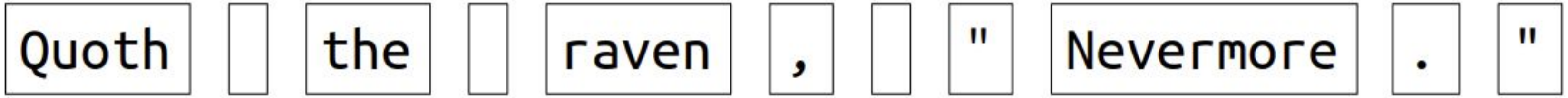
- Granted, you might sometimes have stranger-looking questions like this:

```
Quoth the raven, "Nevermore."
```

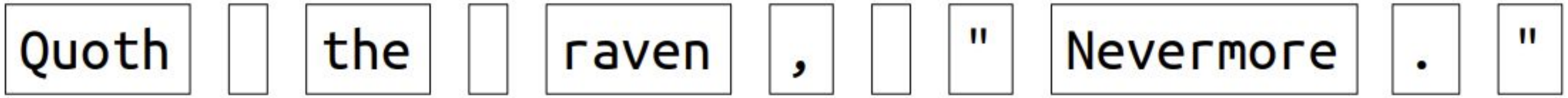
- We've provided you with the following function to help **tokenize** the given sentence.

```
Vector<string> tokenize(const string& sentence);
```

  That can turn the above sentence into the following vector:

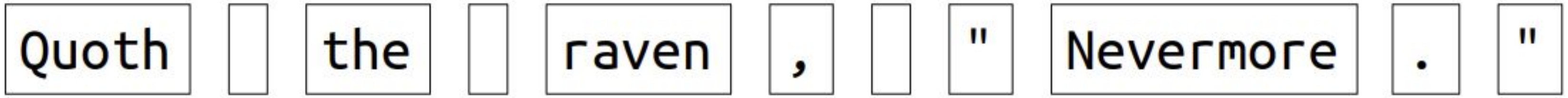


Part 3: What Are YOU Doing?



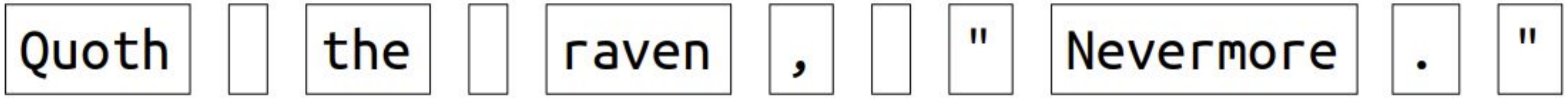
- The good news is, you can determine whether an individual token is a word or not by checking whether the first character is alphabetical!

Part 3: What Are YOU Doing?



- The good news is, you can determine whether an individual token is a word or not by checking whether the first character is alphabetical!
    - That might look something like this: `isalpha(tokenizedString[someIndex][0])`

Part 3: What Are YOU Doing?



- The good news is, you can determine whether an individual token is a word or not by checking whether the first character is alphabetical!
    - That might look something like this: `isalpha(tokenizedString[someIndex][0])`
    - This way you can avoid trying to capitalize / lowercase strings that aren't words! To repeat, **you should ignore non-words!**

Part 3: What Are YOU Doing?

- Some notes about this problem:

Part 3: What Are YOU Doing?

- Some notes about this problem:
  - Think long and hard about what kind of recursion you’re doing here. Are you dealing with subsets? Permutations? Combinations? Something else? Be sure you can answer this question before writing code.

Part 3: What Are YOU Doing?

- Some notes about this problem:
    - Think long and hard about what kind of recursion you’re doing here. Are you dealing with subsets? Permutations? Combinations? Something else? Be sure you can answer this question before writing code.
    - Helper functions are encouraged to complete this part of the assignment!

Part 3: What Are YOU Doing?

- Some notes about this problem:
  - Think long and hard about what kind of recursion you're doing here. Are you dealing with subsets? Permutations? Combinations? Something else? Be sure you can answer this question before writing code.
  - Helper functions are encouraged to complete this part of the assignment!
  - You should ignore the original casings of words, but be aware that you must BOTH toUpperCase() AND toLowerCase() words in the sentence! Students sometimes think you only need to use toUpperCase()

Part 3: What Are YOU Doing?

- Some notes about this problem:
  - Think long and hard about what kind of recursion you’re doing here. Are you dealing with subsets? Permutations? Combinations? Something else? Be sure you can answer this question before writing code.
  - Helper functions are encouraged to complete this part of the assignment!
  - You should ignore the original casings of words, but be aware that you must BOTH toUpperCase() AND toLowerCase() words in the sentence! Students sometimes think you only need to use toUpperCase()
  - You shouldn’t need to call the `tokenize()` function more than once to complete this function!

Part 3: What Are YOU Doing?

- Some notes about this problem:
    - Think long and hard about what kind of recursion you're doing here. Are you dealing with subsets? Permutations? Combinations? Something else? Be sure you can answer this question before writing code.
    - Helper functions are encouraged to complete this part of the assignment!
    - You should ignore the original casings of words, but be aware that you must BOTH toUpperCase() AND toLowerCase() words in the sentence! Students sometimes think you only need to use toUpperCase()
    - You shouldn't need to call the `tokenize()` function more than once to complete this function!
    - Here's how to properly use `toUpperCase()`:

```
string str = "hello there!"
str = toUpperCase(str);
```

Part 3: What Are YOU Doing?

- Some notes about this problem:
  - Think long and hard about what kind of recursion you're doing here. Are you dealing with subsets? Permutations? Combinations? Something else? Be sure you can answer this question before writing code.
  - Helper functions are encouraged to complete this part of the assignment!
  - You should ignore the original casings of words, but be aware that you must BOTH toUpperCase() AND toLowerCase() words in the sentence! Students sometimes think you only need to use toUpperCase()
  - You shouldn't need to call the `tokenize()` function more than once to complete this function!
  - Here's how to properly use `toUpperCase()`:

```
string str = "hello there!"
str = toUpperCase(str);
```

Any questions?

Part 4: Shift Scheduling

- This is it! You’ve made it to the last part of the assignment!

Part 4: Shift Scheduling

- This is it! You’ve made it to the last part of the assignment!
- In this final challenge, you’ll be tasked with creating a **schedule** for a worker that **optimizes profit**.

Part 4: Shift Scheduling

- This is it! You've made it to the last part of the assignment!
- In this final challenge, you'll be tasked with creating a **schedule** for a worker that **optimizes profit**.
  - Wait a minute, does this sound ethical?

Part 4: Shift Scheduling

- This is it! You've made it to the last part of the assignment!
- In this final challenge, you'll be tasked with creating a **schedule** for a worker that **optimizes profit**.
    - Wait a minute, does this sound ethical?

Part 4: Shift Scheduling

- Optimizing a shift schedule for profit is **absolutely not ethical!** Always be conscientious about *what* you're optimizing.

Part 4: Shift Scheduling

- Optimizing a shift schedule for profit is **absolutely not ethical!** Always be conscientious about *what* you're optimizing.
- Despite this… we're going to ask you to do just this in *Shift Scheduling*

Part 4: Shift Scheduling

- Here's what you'll be implementing:

```
Set<Shift> highestValueScheduleFor(const Set<Shift>& shifts, int maxHours);
```

Part 4: Shift Scheduling

- Here's what you'll be implementing:

```
Set<Shift> highestValueScheduleFor(const Set<Shift>& shifts, int maxHours);
```

- Your job is to return the collection of `shift`'s that maximizes profit given a full collection of `shifts` and a maximum number of hours an individual can work.

Part 4: Shift Scheduling

- Here's what you'll be implementing:

```
Set<Shift> highestValueScheduleFor(const Set<Shift>& shifts, int maxHours);
```

- Your job is to return the collection of `shift`'s that maximizes profit given a full collection of `shifts` and a maximum number of hours an individual can work.
- Here are a few functions we've written for you that use `shift`:

```
int lengthOf(const Shift& shift);     // Returns the length of a shift.
int valueOf(const Shift& shift);      // Returns the value of this shift.
bool overlapsWith(const Shift& one,   // Returns whether two shifts overlap.
                  const Shift& two);
```

Part 4: Shift Scheduling

- Here’s how we want you to approach this problem:

Part 4: Shift Scheduling

- Here’s how we want you to approach this problem:
    - Take some shift (any you’d like!) from the collection of given `shifts`.

Part 4: Shift Scheduling

- Here’s how we want you to approach this problem:
    - Take some shift (any you’d like!) from the collection of given `shifts`.

Part 4: Shift Scheduling

- Here's how we want you to approach this problem:
  - Take some shift (any you'd like!) from the collection of given `shifts`.
  - Determine whether the employee can actually take this shift:

Part 4: Shift Scheduling

- Here's how we want you to approach this problem:
  - Take some shift (any you'd like!) from the collection of given `shifts`.
  - Determine whether the employee can actually take this shift:
    - Does this shift's length put the worker over the max number of hours they're allowed to work?

Part 4: Shift Scheduling

- Here’s how we want you to approach this problem:
    - Take some shift (any you’d like!) from the collection of given `shifts`.
    - Determine whether the employee can actually take this shift:
        - Does this shift’s length put the worker over the max number of hours they’re allowed to work?
        - Does this shift overlap with a shift the worker is already working?

Part 4: Shift Scheduling

- Here's how we want you to approach this problem:
  - Take some shift (any you'd like!) from the collection of given `shifts`.
  - Determine whether the employee can actually take this shift:
    - Does this shift's length put the worker over the max number of hours they're allowed to work?
    - Does this shift overlap with a shift the worker is already working?
  - If this shift is invalid for either reason, discard it, and repeat the process on a new shift (does this sound self-similar?)

Part 4: Shift Scheduling

- Here's how we want you to approach this problem:
  - Take some shift (any you'd like!) from the collection of given `shifts`.
  - Determine whether the employee can actually take this shift:
    - Does this shift's length put the worker over the max number of hours they're allowed to work?
    - Does this shift overlap with a shift the worker is already working?
  - If this shift is invalid for either reason, discard it, and repeat the process on a new shift (does this sound self-similar?)
  - If the shift is valid, you have **two** options:

Part 4: Shift Scheduling

- Here's how we want you to approach this problem:
  - Take some shift (any you'd like!) from the collection of given `shifts`.
  - Determine whether the employee can actually take this shift:
    - Does this shift's length put the worker over the max number of hours they're allowed to work?
    - Does this shift overlap with a shift the worker is already working?
  - If this shift is invalid for either reason, discard it, and repeat the process on a new shift (does this sound self-similar?)
  - If the shift is valid, you have **two** options:
    - See what happens if you **include** this shift, and continue for a new shift

Part 4: Shift Scheduling

- Here’s how we want you to approach this problem:
  - Take some shift (any you’d like!) from the collection of given `shifts`.
  - Determine whether the employee can actually take this shift:
    - Does this shift’s length put the worker over the max number of hours they’re allowed to work?
    - Does this shift overlap with a shift the worker is already working?
  - If this shift is invalid for either reason, discard it, and repeat the process on a new shift (does this sound self-similar?)
  - If the shift is valid, you have **two** options:
    - See what happens if you **include** this shift, and continue for a new shift
    - See what happens if you **exclude** this shift, and continue for a new shift

Part 4: Shift Scheduling

- Here's how we want you to approach this problem:
  - Take some shift (any you'd like!) from the collection of given `shifts`.
  - Determine whether the employee can actually take this shift:
    - Does this shift's length put the worker over the max number of hours they're allowed to work?
    - Does this shift overlap with a shift the worker is already working?
  - If this shift is invalid for either reason, discard it, and repeat the process on a new shift (does this sound self-similar?)
  - If the shift is valid, you have **two** options:
    - See what happens if you **include** this shift, and continue for a new shift
    - See what happens if you **exclude** this shift, and continue for a new shift
  - We want you to try **both**, and return the collection of shifts that has the **greater** aggregate value.

Part 4: Shift Scheduling

- Here's how we want you to approach this problem:
  - Take some shift (any you'd like!) from the collection of given `shifts`.
  - Determine whether the employee can actually take this shift:
    - Does this shift's length put the worker over the max number of hours they're allowed to work?
    - Does this shift overlap with a shift the worker is already working?
  - If this shift is invalid for either reason, discard it, and repeat the process on a new shift (does this sound self-similar?)
  - If the shift is valid, you have **two** options:
    - See what happens if you **include** this shift, and continue for a new shift
    - See what happens if you **exclude** this shift, and continue for a new shift
  - We want you to try **both**, and return the collection of shifts that has the **greater** aggregate value.

```
int lengthOf(const Shift& shift);     // Returns the length of a shift.
int valueOf(const Shift& shift);      // Returns the value of this shift.
bool overlapsWith(const Shift& one,   // Returns whether two shifts overlap.
                  const Shift& two);
```

Part 4: Shift Scheduling

- Here’s how we **DON’T** want you to approach this problem:

Part 4: Shift Scheduling

- Here's how we **DON'T** want you to approach this problem:
  - Recursively enumerate ALL possible combinations of shifts, then loop through the conglomerate and find the most valuable schedule that is feasible

Part 4: Shift Scheduling

- Here's how we **DON'T** want you to approach this problem:
  - Recursively enumerate ALL possible combinations of shifts, then loop through the conglomerate and find the most valuable schedule that is feasible
  - Why is this such a bad idea?

Part 4: Shift Scheduling

- Here are a few tips about this problem:

Part 4: Shift Scheduling

- Here are a few tips about this problem:
    - Follow the step-by-step approach we've given you here/on the handout!

Part 4: Shift Scheduling

- Here are a few tips about this problem:
  - Follow the step-by-step approach we’ve given you here/on the handout!
  - You shouldn’t actually need to look at the internals of the `shift` struct! Simply use the three functions we’ve provided for you :)

Part 4: Shift Scheduling

- Here are a few tips about this problem:
  - Follow the step-by-step approach we've given you here/on the handout!
  - You shouldn't actually need to look at the internals of the `shift` struct! Simply use the three functions we've provided for you :)
  - An optimized collection of shifts might not use up all of the hours of a worker!

Part 4: Shift Scheduling

- Here are a few tips about this problem:
  - Follow the step-by-step approach we've given you here/on the handout!
  - You shouldn't actually need to look at the internals of the `shift` struct! Simply use the three functions we've provided for you :)
  - An optimized collection of shifts might not use up all of the hours of a worker!
  - Workers can have a `maxHours` value of 0, but you should raise an `error()` if that value is ever negative!

Part 4: Shift Scheduling

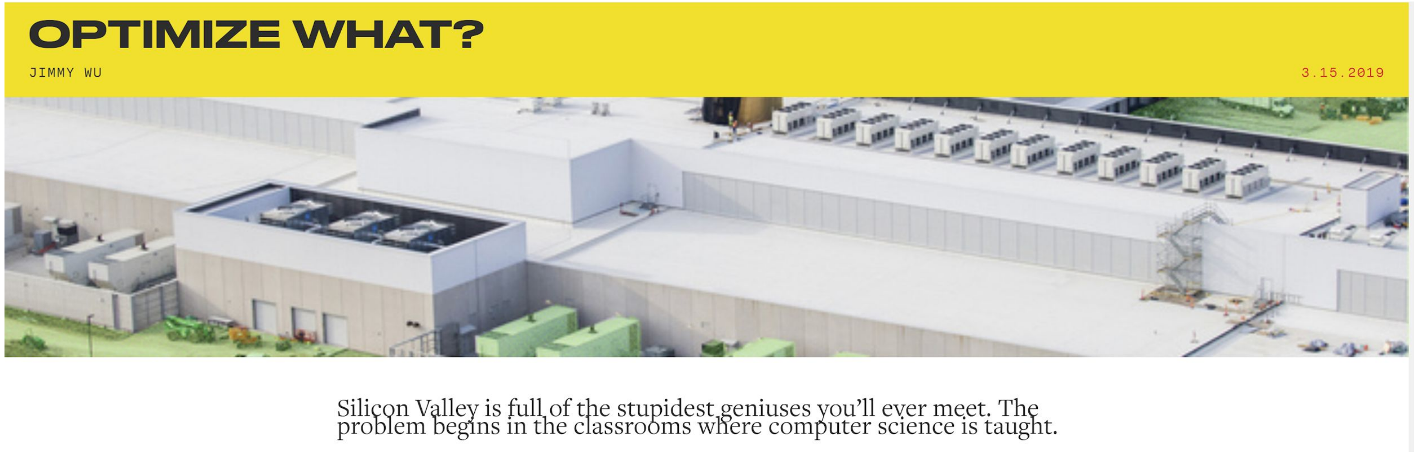
- Here are a few tips about this problem:
  - Follow the step-by-step approach we've given you here/on the handout!
  - You shouldn't actually need to look at the internals of the `shift` struct! Simply use the three functions we've provided for you :)
  - An optimized collection of shifts might not use up all of the hours of a worker!
  - Workers can have a `maxHours` value of 0, but you should raise an `error()` if that value is ever negative!
  - Remember to think long and hard about the implications of your work -- take a look at some of the schedules your algorithms produce -- you might be optimizing profit, but at what cost?

Part 4: Shift Scheduling

- Here are a few tips about this problem:
    - Follow the step-by-step approach we've given you here/on the handout!
    - You shouldn't actually need to look at the internals of the `shift` struct! Simply use the three functions we've provided for you :)
    - An optimized collection of shifts might not use up all of the hours of a worker!
    - Workers can have a `maxHours` value of 0, but you should raise an `error()` if that value is ever negative!
    - Remember to think long and hard about the implications of your work -- take a look at some of the schedules your algorithms produce -- you might be optimizing profit, but at what cost?
    - When making your recursive calls, ensure that **every** variable is being updated correctly! Ensure you can justify each modification you make to your parameters.

Any questions about part 4?

I'd recommend taking a look through Jimmy Wu's article *"Optimize What?"* It's an eye-opening piece about how a-seemingly innocuous CS education can be a *very* dangerous thing for society.



OPTIMIZE WHAT?

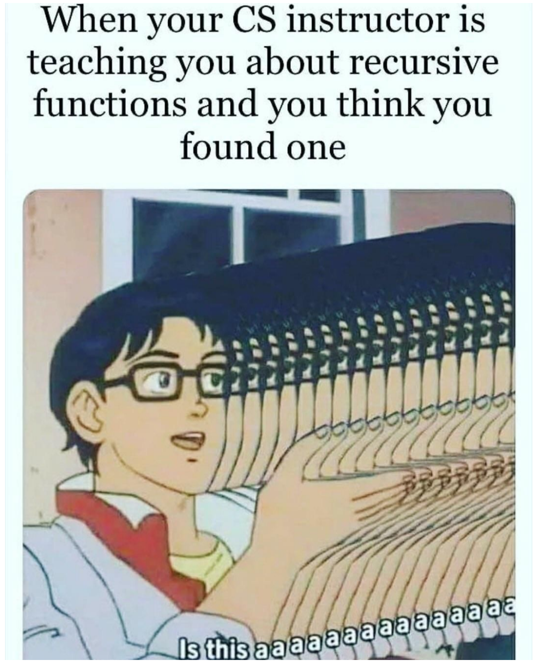
JIMMY WU

3.15.2019

Silicon Valley is full of the stupidest geniuses you'll ever meet. The problem begins in the classrooms where computer science is taught.

Congrats!

This was a **big** assignment, but we believe in you! Remember that you have lots of support!


When your CS instructor is teaching you about recursive functions and you think you found one
Is this aaaaaaaaaaaaaaaa