YEAH Hours A3

Welcome to the world of **Recursion!**

-Professor Oak, CS106B alum and recursion pro

This week.... Recursion!

WHAT ARE YOU WORKING ON?
TRYING TO FIX THE PROBLEMS I CREATED WHEN I TRIED TO FIX THE PROBLEMS I CREATED WHEN I TRIED TO FIX THE PROBLEMS I CREATED WHEN...
Source, XKCD

- Recursion is the process by which a function calls itself.
- Recursive solutions consist of one or more **base case(s)**, which are specified terminating conditions for a recursive function, and **recursive case(s)**, which advance your recursive path one step towards your base case.
- Recursion can be tricky! So be ready to start this assignment early! The good news is, you shouldn't have to write much code at all!

A3 Logistics

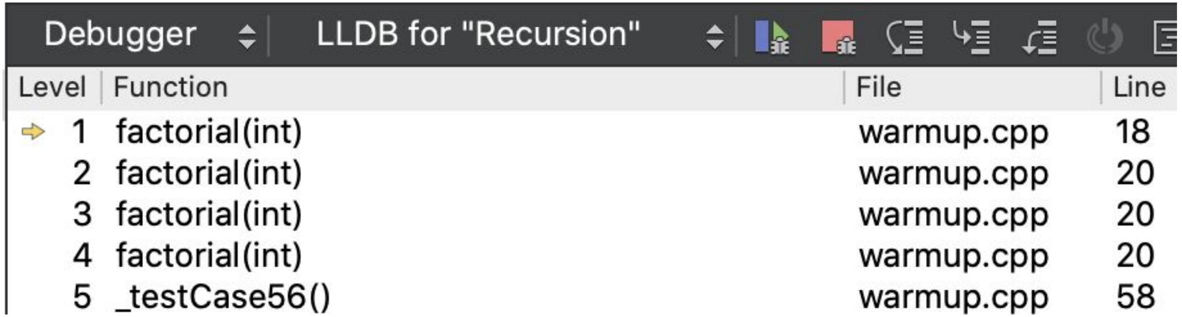
- Due Friday Oct 9th (11:59PDT)
- Late Submission Deadline Sunday Oct 11th (11:59PDT)
- As always, work must be done individually.

Let’s take a look at Assignment 3

- Your assignment consists of **five** fun and exciting parts!
    - 1. **Recursion Debugging Warmup**
    - 2. **Balanced** - A program that verifies that an expression has a balanced amount of parentheses / brackets
    - 3. **KarelGPS** - An old friend returns in a recursive adventure!
    - 4. **Sierpinski** - The ol’ faithful triangle drawing program that has brought thousands of 106B students into the recursive world
    - 5. **Merge** - Harness the power of recursion to write a blazingly-fast sorting algorithm!

Part 1: Recursion Warmup!

- In this part, you will be examining **two** recursive functions.
  - int **factorial**(int n) -> A function that computes n factorial (n!)
  - You will step through this function in the debugger with a focus on **the call stack**. The **call stack** is the list of function calls that brought you to your current line in the program. Does this call stack make sense for a recursive function?



Level	Function	File	Line
⇨ 1	factorial(int)	warmup.cpp	18
2	factorial(int)	warmup.cpp	20
3	factorial(int)	warmup.cpp	20
4	factorial(int)	warmup.cpp	20
5	_testCase56()	warmup.cpp	58

Part 1: Recursion Warmup!

- The **factorial**(int n) function is buggy! When given negative input, the function calls n * **factorial**(n-1) with no base case to stop it, so you end up totally blowing up your program memory. This is called **Stack Overflow.**
- You'll learn how to use the debugger to detect stack overflow (which can be common in recursive programs!)
- tldr; be careful with your base cases. If you don't account for a certain kind of input, stack overflow or other undesirable behavior is likely!

Part 1: Recursion Warmup!

- The second function you will look at is double **power**(int base, int exp)
  - This function returns the mathematical result base^(exp).
  - Sadly, there is a **bug** in the recursive **power**() function, specifically when exp is negative. It’s your job to write tests to uncover the bug!
  - ***Hint****:* The starter code gives you a great randomized test for **power**() on positive bases and exponents -- maybe you can modify/repurpose it to support negative numbers?
  - You’ll report your findings, as well as answer related questions in **shortanswer.txt**

Questions about part 1?

NEVER HAVE I FELT SO CLOSE TO ANOTHER SOUL
AND YET SO HELPLESSLY ALONE
AS WHEN I GOOGLE AN ERROR
AND THERE'S ONE RESULT
A THREAD BY SOMEONE WITH THE SAME PROBLEM
AND NO ANSWER
LAST POSTED TO IN 2003
WHO WERE YOU, DENVERCODER9?
WHAT DID YOU SEE?!

Source, XKCD

Part 2: Balanced

- Implement the function bool **isBalanced**(string str), which, given a string, returns whether the *bracketing operators* are properly balanced.
    - The bracketing operators you will be using are these: **[] {} ()**
    - We define balance as properly nested such that they would compile in a c++ program.
    - A correct example: `( ) { ( [ ] ) ( ( ) ) }` -- (spaced out for viewing)
    - An incorrect example: `{   (   }   )`
- **isBalanced**() won't actually have much code in it… you're simply going to call the following two functions ->

Part 2: Balanced

- You will need to implement two functions:
  - 1 - string **operatorsFrom**(string str);
    - Given string, remove all non-operator characters->[] {} ()
    - You **must** implement this recursively. To do so, you should process a single character at a time and recurse on the remainder of the string.
    - ***Hint***: Consider using str.substr(1) to easily get the rest of the string.
    - Remember to test this function **thoroughly** before moving on! Bugs here could go a long way...

Part 2: Balanced

- You will need to implement two functions:
  - 1 - string **operatorsFrom**(string str);
    - Given string, remove all non-operator characters->[] {} ()
    - You **must** implement this recursively. To do so, you should process a single character at a time and recurse on the remainder of the string.
    - ***Hint***: Consider using str.substr(1) to easily get the rest of the string.
    - Remember to test this function **thoroughly** before moving on! Bugs here could go a long way...

Tip just for YEAH viewers: this kind of recursion is very common with strings. You'll see a similar thing done with a string reverse() function

Part 2: Balanced

- You will need to implement two functions:
  - 2 - `bool operatorsAreMatched(string ops);`
    - This is the function that truly implements the isBalanced() process.
    - The handout gives you some really really good advice here: a string is balanced iff (if and only if ;))
      - The string is empty.
      - The string contains `"()"`, `"[]"`, or `"{}"` as a substring and the rest of the string is balanced after removing that substring.
    - From these givens, we can derive the following...

Part 2: Balanced

- If your string is non-empty, you have work to do…
  - Look for an instance of “{}” “[]” or “()” in your string.
    - If you find one, remove it, and see if the remainder of the string is balanced!
    - If you don’t find one, your string isn’t balanced :(.
- If your string is empty, your string is balanced!

```
operatorsAreMatched("[(){}]") -> "()" exists, remove it and check whether rest is matched
  operatorsAreMatched("[{}]") -> "{}" exists, remove it and check whether rest is matched
    operatorsAreMatched("[]") -> "[]" exists, remove it and check whether rest is matched
      operatorsAreMatched("") -> true
```

Part 2: Balanced

- A few notes on this problem
  - Both helper functions should be implemented **recursively**, and you should not have **any** loops or data structures in your solution.
  - Be sure you thoroughly test these two helpers -- if you get errors from isBalanced(), it’s more difficult to see where your bug came from!
  - You can assume that operatorsAreMatched() always takes in strings of only operators, but if the input string does not, the function should return false, as expected. (i.e. don’t throw an error)

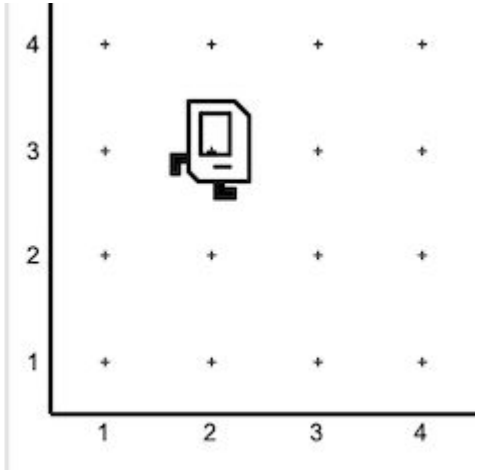
Questions about part 2?

(AN UNMATCHED LEFT PARENTHESIS CREATES AN UNRESOLVED TENSION THAT WILL STAY WITH YOU ALL DAY.

Source, XKCD

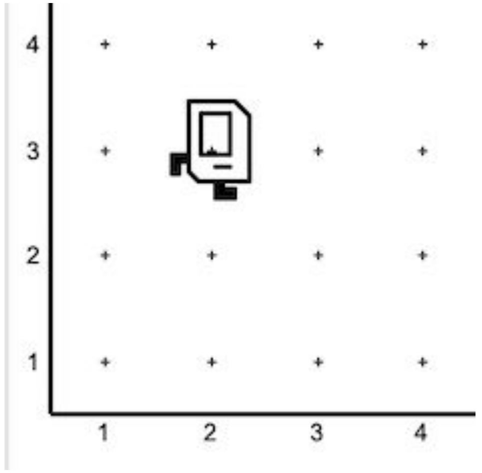
Part 3: KarelGPS

- For those of you who don't know Karel -- this is Karel!



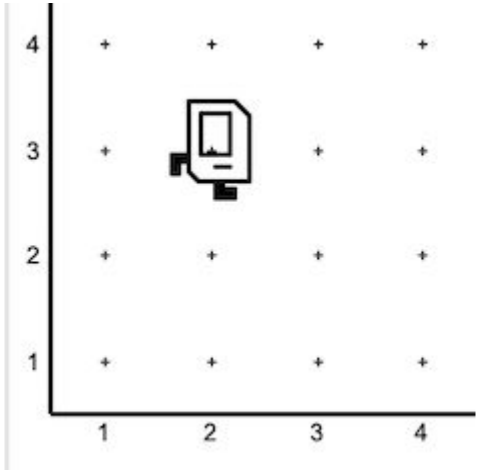
Part 3: KarelGPS

- For those of you who don’t know Karel -- this is Karel!
- Karel lives in an x-y coordinate system, where rows are called “streets” and columns are called “avenues”



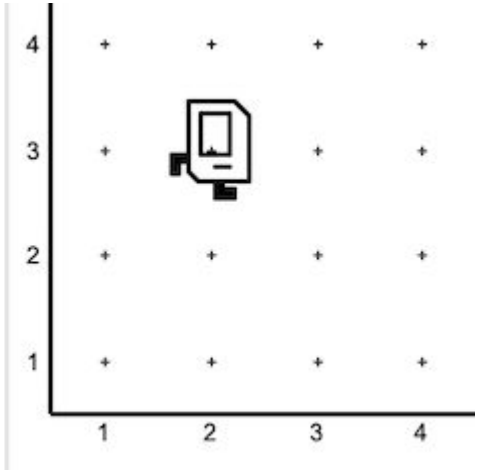
Part 3: KarelGPS

- For those of you who don’t know Karel -- this is Karel!
- Karel lives in an x-y coordinate system, where rows are called “streets” and columns are called “avenues”
  - Where is Karel on the grid right now?



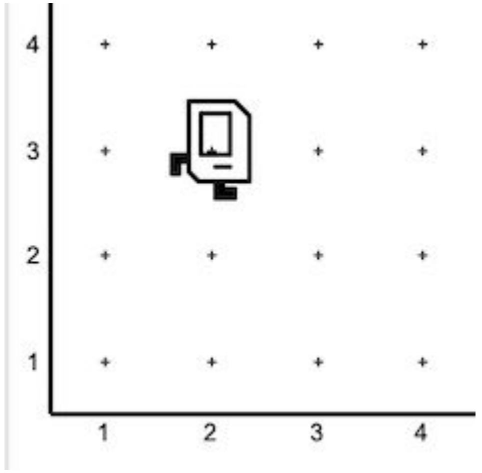
Part 3: KarelGPS

- For those of you who don’t know Karel -- this is Karel!
- Karel lives in an x-y coordinate system, where rows are called “streets” and columns are called “avenues”
  - Where is Karel on the grid right now?
- Karel dreams of getting to the intersection of 1st and 1st, and they want to do it in the least number of moves possible: can you help Karel do that using recursion?



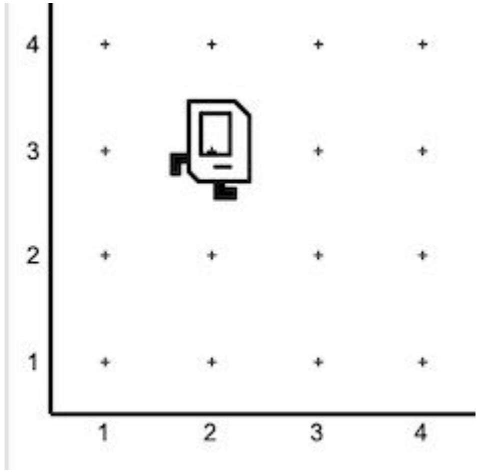
Part 3: KarelGPS

- Karel moves by turning in a direction and moving *one* unit of distance. Therefore, to get to 1,1, Karel should only be able to move *one* unit west or *one* unit south.
    - Karel could also go North / East, but Karel wants to get to 1,1 quickly, so we're not going to consider those possibilities!



Part 3: KarelGPS

- Karel is pretty good at data analysis, so all they need is for you to *count* the number of routes that they could possible take from their starting point to get to 1,1.

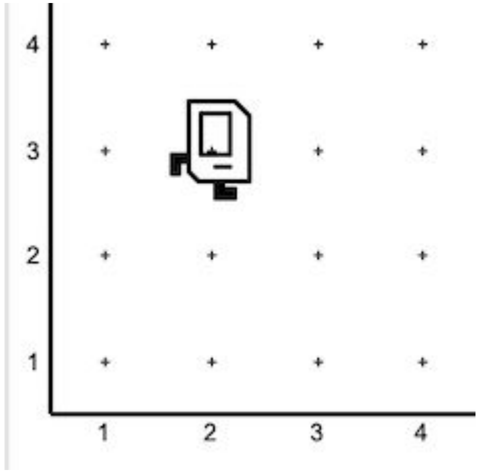


Part 3: KarelGPS

- Karel is pretty good at data analysis, so all they need is for you to *count* the number of routes that they could possible take from their starting point to get to 1,1.
- Therefore, you'll need to write the function:

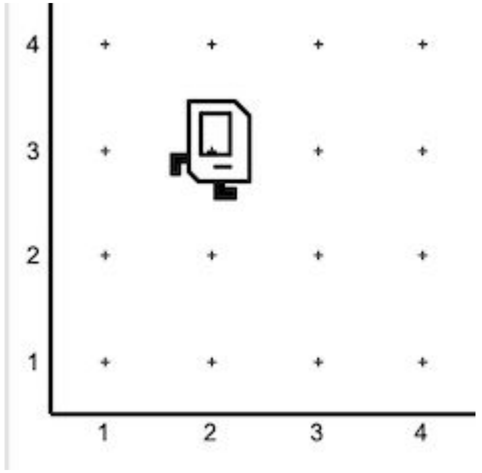
```
int countRoutes(int street, int avenue)
```

which, given the street-avenue combo, counts the number of possible routes Karel can take to get to 1,1



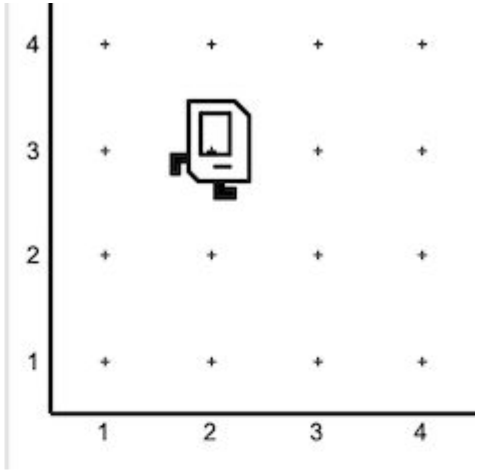
Part 3: KarelGPS

- A few details:
  - At any point in time (any location), think about the 'choices' you can make. Which directions is Karel allowed to go, and what does that look like in a function call?
  - Think about base and edge cases here. Are there times in which Karel *cannot* go in a direction, but still needs to move?
  - All you need to do is return a count of the unique paths. There's no need to print anything



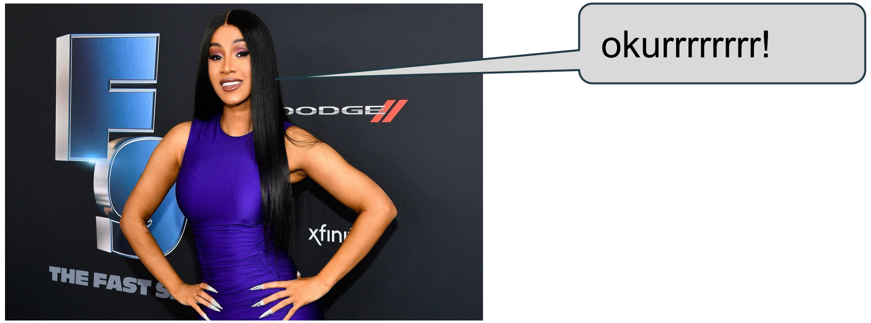
Part 3: KarelGPS

- A few more deets:
  - Although you can expect that all street-avenue pairs given to you will be >= 1, we can't promise it in our tests >:). You should throw an error if such a thing happens, and test that behavior with **EXPECT_ERROR**
  - Here's a hint from me: you don't actually *need* to declare any integers to solve this problem



Questions about part 3?

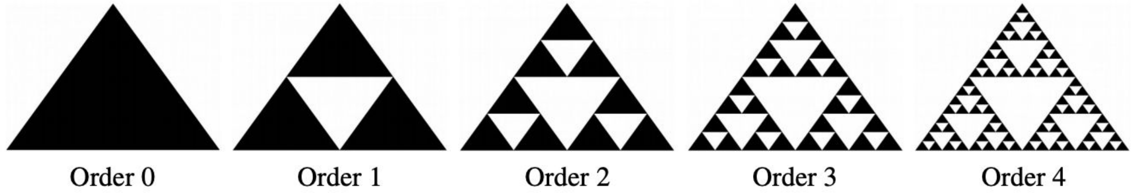
- Don’t overthink this problem! I’d highly recommend drawing pictures, and working through small examples you can trace your code on :)



Artist Cardi 106B, upon hearing about your heroic efforts to bring Karel home

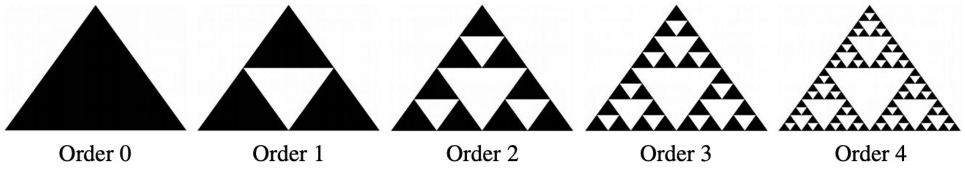
Part 4: Sierpinski

- In this part, you'll be asked to draw *n-order* Sierpinski Triangles, named after Polish Mathematician Wacław Sierpiński.
- 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.



- int **drawSierpinskiTriangle**(GWindow& window, GPoint one, GPoint two, GPoint three, int order)
- void **fillBlackTriangle**(GWindow& window, GPoint one, GPoint two, GPoint three)

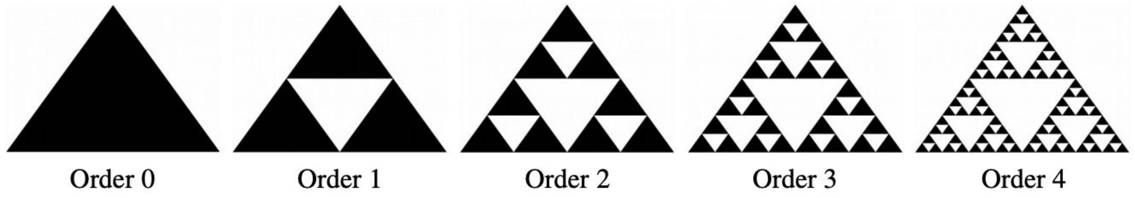
Part 4: Sierpinski



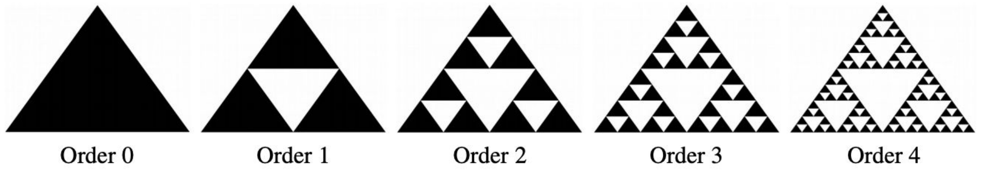
- int **drawSierpinskiTriangle**(GWindow& window, GPoint one, GPoint two, GPoint three, int order)
- void **fillBlackTriangle**(GWindow& window, GPoint one, GPoint two, GPoint three)
    - A GWindow is just the console object that you'll be drawing on: you can ignore it :p
    - drawSierpinskiTriangle() returns the number of triangles it drew.

Part 4: Sierpinski

- A few implementation thoughts:
  - If 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.



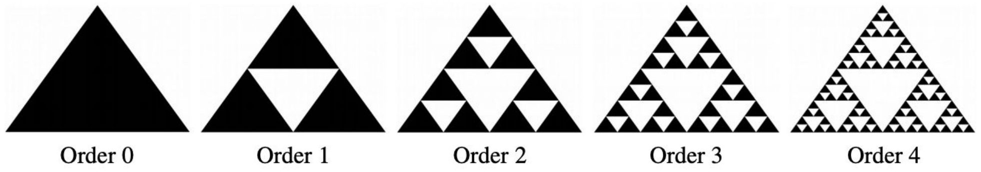
Part 4: Sierpinski



- Tips:
  - The GPoint object contains two **doubles**: x and y. To access them, use **point.x** and **point.y**. (sound familiar?)
  - To declare a new GPoint, an easy way of doing so is using the {} brackets.
    - E.x. GPoint p = { 1.0, 2.0 };
  - To get the midpoint between two points, you can write
    - GPoint midpt =

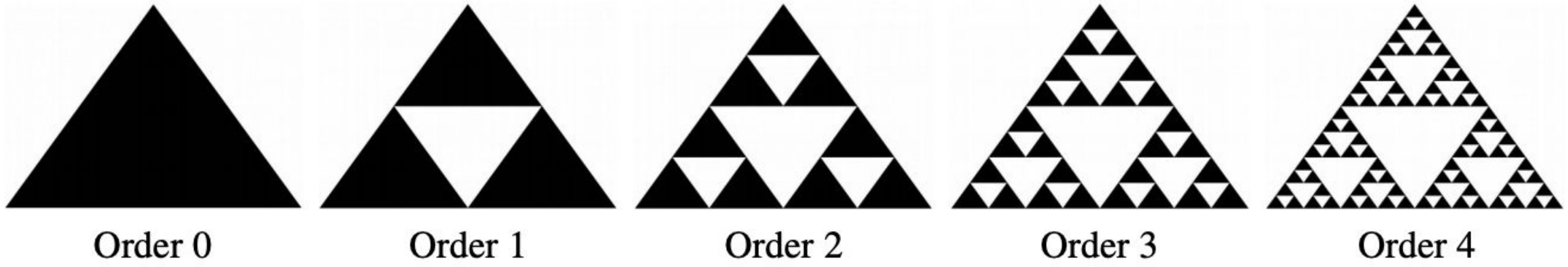
      { (p1.x + p2.x) / 2, (p1.y + p2.y) / 2 };

Part 4: Sierpinski



- More tips:
    - 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!)
    - When returning how many triangles were drawn by a call, think about that number recursively -- **don’t just compute it from the first call**. We want to see your recursive calls work together to compute this number. Think about the Karel assignment you just did :)

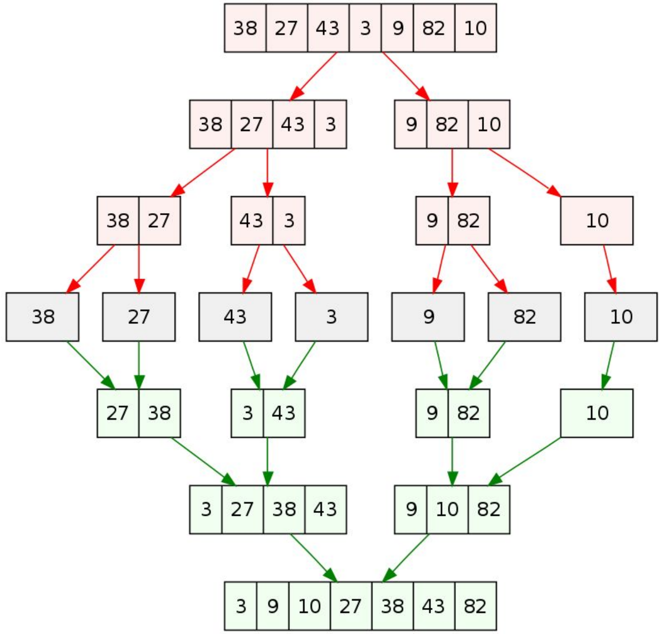
Questions about part 4?



Part 5: Merging Sorted Sequences

- In this part of the assignment, you’re going to be tasked with implementing a very famous sorting algorithm called MergeSort. MergeSort is a recursive divide-and conquer algorithm that achieves an impressive runtime by recursively splitting and recombining its data in sorted order.
- In this assignment, however, you’re actually going to be implementing a special ~flavor~ of mergeSort!
- Let’s go through it step by step.

Part 5: Merging Sorted Sequences

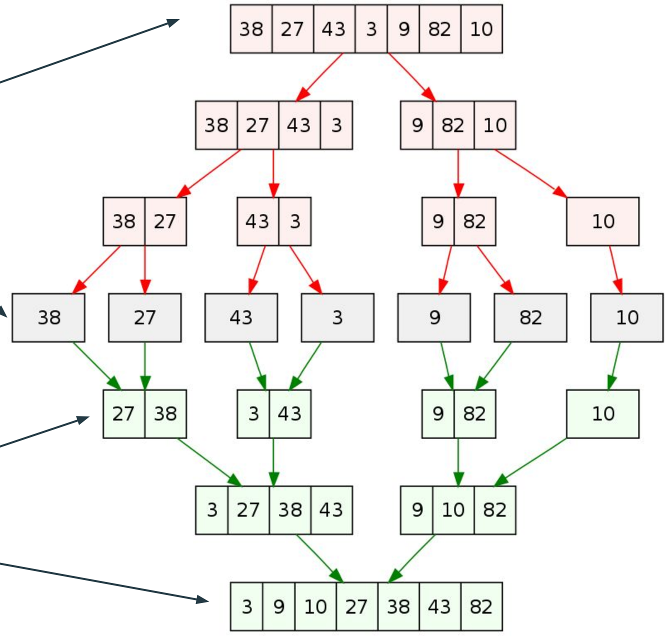


NOTE: Don’t worry about this diagram too much -- it’s a little thicc. I’m happy to talk about mergeSort more after YEAH, but you don’t need to know everything about it to get this part done!

Part 5: Merging Sorted Sequences

Part 1: Split sequences in half until you get singleton elements

Part 2: Recombine each pair in order



Part 5: Merging Sorted Sequences

- Although the diagram above made it look like you’ll be working with a Vector<int>, in this part, we’ll be using a **Vector<Queue<int>>**. Don’t worry! **It doesn’t add any complexity to the problem!**
  - Just think of every element in the collection as a Queue, instead of a single number.

Part 5: Merging Sorted Sequences

- The **first** thing you'll need to do in this part is write the function that **merges** two singleton elements.
  - In this case, that's merging **two sorted Queue<int>**'s to make one large sorter **Queue<int>**.
  - Why are they already sorted? In the beginning we'll just guarantee it, so in subsequent merges, they'll be sorted too. That's just how mergeSort works 😎

Part 5: Merging Sorted Sequences

```
Queue<int> merge(Queue<int> one, Queue<int> two)
```

- More specifically you’ll need to implement the above function. It should merge the queue’s **one** and **two**, and return a sorted version that contains all element from both.
    - Sanity Check -- why are the queue’s passed by value?

Part 5: Merging Sorted Sequences

```
Queue<int> merge(Queue<int> one, Queue<int> two)
```

- More specifically you'll need to implement the above function. It should merge the queue's **one** and **two**, and return a sorted version that contains all element from both.
  - Sanity Check -- why are the queue's passed by value?
- The queue's will be sorted from **smallest to largest**, so the first element you dequeue() will be the smallest element in the queue.
  - Hint -- this fact is **very useful** for putting elements together in a larger queue.

Part 5: Merging Sorted Sequences

```
Queue<int> merge(Queue<int> one, Queue<int> two)
```

- More deets about this function:
    - It’s very easy to make this function large and repetitive -- you can solve this in a compact way, we promise!
        - You should never feel like you’re copying any code on this part.

Part 5: Merging Sorted Sequences

```
Queue<int> merge(Queue<int> one, Queue<int> two)
```

- More deets about this function:
    - It’s very easy to make this function large and repetitive -- you can solve this in a compact way, we promise!
        - You should never feel like you’re copying any code on this part.
    - The two queue’s you’re given are not guaranteed to the be the same length. What should you do if one queue has ran out of elements but another has not?

Part 5: Merging Sorted Sequences

```
Queue<int> merge(Queue<int> one, Queue<int> two)
```

- Even more deets about this function:
    - Remember when we said that the queue's given to you were in sorted order? We lied… sort of (*sort*, get it hahahaha)

Part 5: Merging Sorted Sequences

```
Queue<int> merge(Queue<int> one, Queue<int> two)
```

- Even more deets about this function:
    - You’ll need to verify that the queue’s given are in sorted order, and **raise an error** if they’re not.

Part 5: Merging Sorted Sequences

```
Queue<int> merge(Queue<int> one, Queue<int> two)
```

- Even more deets about this function:
  - You’ll need to verify that the queue’s given are in sorted order, and **raise an error** if they’re not.
    - The easiest way to do this is to write a helper that just does a full pass through both queue’s before you merge, verifying that everything is sorted.
    - The more complicated yet faster way is to somehow verify that the queue’s are in order *during* the merge itself… this might require some extra bookkeeping on your end.

Part 5: Merging Sorted Sequences

```
Queue<int> merge(Queue<int> one, Queue<int> two)
```

- Even more deets about this function:
    - You’ll need to verify that the queue’s given are in sorted order, and **raise an error** if they’re not.
        - The easiest way to do this is to write a helper that just does a full pass through both queue’s before you merge, verifying that everything is sorted.
        - The more complicated yet faster way is to somehow verify that the queue’s are in order *during* the merge itself… this might require some extra bookkeeping on your end.

Part 5: Merging Sorted Sequences

```
Queue<int> merge(Queue<int> one, Queue<int> two)
```

- One last thing…
  - As tempting as it might be, **please implement this iteratively.**
  - Let me restate: **you may not use recursion to complete this function**. It produces too many stack frames, and could crash your program.

Questions about binary merge?

```
Queue<int> merge(Queue<int> one, Queue<int> two)
```

Trip telling 106B students to **not** use recursion to complete the recursion assignment

Part 5: Merging Sorted Sequences

- It’s time for the final code of the assignment!

Part 5: Merging Sorted Sequences

```
Queue<int> recMultiMerge(Vector<Queue<int>>& all)
```

- You'll be writing the above function, that takes in a *sequence*, or a Vector of Queue<int>'s, and returns a single Queue<int> that contains all elements from the Vector of Queue's but is completely sorted!
- This might sound tricky, but we've given you another **algorithm** to follow here!

Part 5: Merging Sorted Sequences

1. Divide the vector of **k** sequences (queues) into two halves. The "left" half is the first **k/2** sequences (queues) in the vector, and the "right" half is the rest of the sequences (queues).
    - The Vector class has a helpful **subList** operation to subdivide a Vector.
2. Make a recursive call to **recMultiMerge** on the "left" half of the sequences to generate one combined, sorted sequence. Then, do the same for the "right" half of the sequences to generate a second combined, sorted sequence.
3. Use your binary **merge** function to join the two combined sequences into the final result sequence, which is then returned.

Part 5: Merging Sorted Sequences

```
Queue<int> recMultiMerge(Vector<Queue<int>>& all)
```

- Let’s do a little sanity check of the algorithm to ensure it does what we’re expecting:

Part 5: Merging Sorted Sequences

```
Queue<int> recMultiMerge(Vector<Queue<int>>& all)
```

- Let’s do a little sanity check of the algorithm to ensure it does what we’re expecting:
  - Step 1 divides the Vector **ALL** into two halves, a left half and a right half.

Part 5: Merging Sorted Sequences

```
Queue<int> recMultiMerge(Vector<Queue<int>>& all)
```

- Let’s do a little sanity check of the algorithm to ensure it does what we’re expecting:
    - Step 1 divides the Vector **ALL** into two halves, a left half and a right half.
    - Step 2 calls recMultiMerge() on both halves. Now we have a Queue<int> representing the left half, and a Queue<int> representing the right half.

Part 5: Merging Sorted Sequences

```
Queue<int> recMultiMerge(Vector<Queue<int>>& all)
```

- Let’s do a little sanity check of the algorithm to ensure it does what we’re expecting:
  - Step 1 divides the Vector **ALL** into two halves, a left half and a right half.
  - Step 2 calls recMultiMerge() on both halves. Now we have a Queue<int> representing the left half, and a Queue<int> representing the right half.
  - Step 3 returns the result of calling your **merge** function on both halves -- it returns a single Queue<int> -- does that sound right?

Part 5: Merging Sorted Sequences

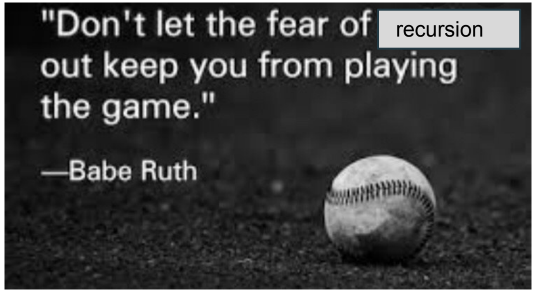
```
Queue<int> recMultiMerge(Vector<Queue<int>>& all)
```

- Some notes on this problem:
    - It might feel weird to jump into that algorithm without fully understanding *why* mergesort works -- it’s actually a little unintuitive, but I promise the algorithm works. I’ll stay after and explain if anyone wants to know more!
    - Be sure that you’ve got the right set of base cases here -- for example, what should you do if the Vector is empty?

Any question about part 5?

- You’ve just (more or less) implemented the function mergeSort, which, when run on a Vector<int> has a runtime of **Nlog(N)**, where N is the number of ints in the container. Funnily enough, **Nlog(N)** is actually the fastest you can do with comparative-based sorting. Want to know why? Take CS161!

That's it!


"Don't let the fear of recursion out keep you from playing the game."
—Babe Ruth

- Good luck! Recursion is really difficult to grasp at first, but over time (with practice), you'll start seeing problem decomposition a little differently.
    - Be sure to start this assignment early! It's not a lot of code, but the few lines you write need to be precise. **I always say that the more code you write in your recursive functions, the more likely it is that your program won't work.**
    - Feel free to ping your SL or go to office hours or LAIR. It's really important to understand recursion, because we're going to use it frequently for the rest of the quarter!