YEAH HOURS A3

-Professor Oak, CS106B alum and recursion pro

Let’s take a look at Assignment 3

Your assignment consists of **2 parts,** each with their own subparts!

- **Part 1: Normal Recursion**
  - 1. **Recursion Debugging Warmup**
  - 2. **Balanced** - A program that verifies that an expression has a balanced amount of parentheses / brackets
  - 3. **Sierpinski** - The ol’ faithful triangle drawing program that has brought thousands of 106B students into the recursive world
  - 4. **Merge** - A program that merges collections of sequences recursively to achieve lightning-fast sorting abilities!

Let’s take a look at Assignment 3

Your assignment consists of **2 parts,** each with their own subparts!

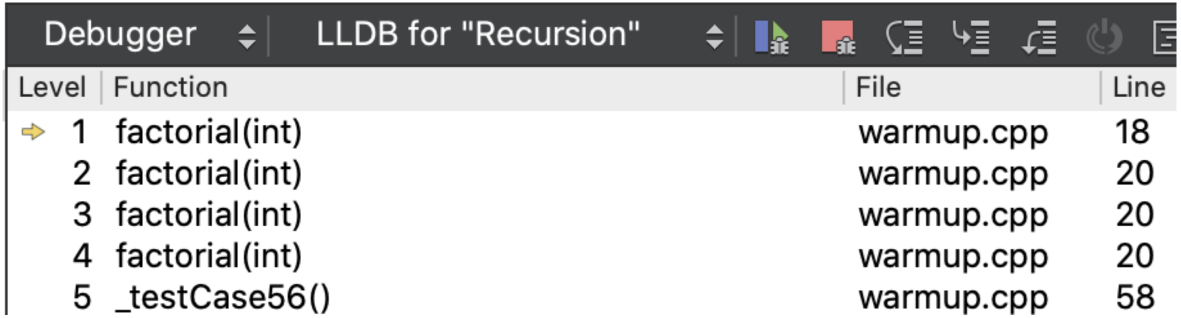
- **Part 2: Recursive Backtracking**
  - 1. **Recursive Backtracking Debugging Warmup**
  - 2. **Boggle** – An iconic 106B problem where you’ll become the best boggle player of all time!
  - 3. **Optional Extra Credit** – Spooky!

Let’s take a look at Assignment 3

- The assignment is due Thursday 7/16 at 11:59PDT; you’ll have a 1-day grace period.
- This assignment has a lot of parts! Start early and ask questions ☺

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

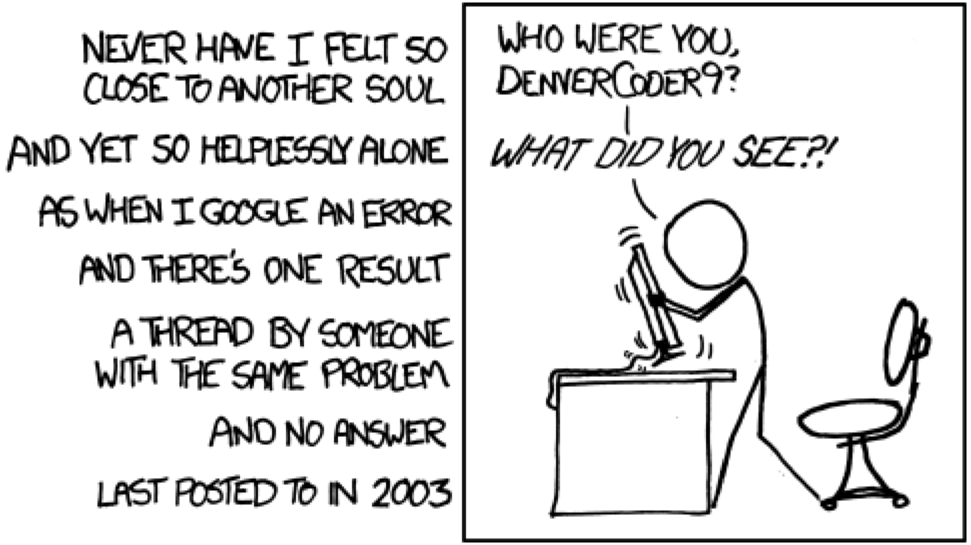
Part 1: Recursion Warmup!

- 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 it to support negative numbers?

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 **operatorsOnly**(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...
    - Ex. “thisisSomeCode().hello{!(]” → “(){(]”

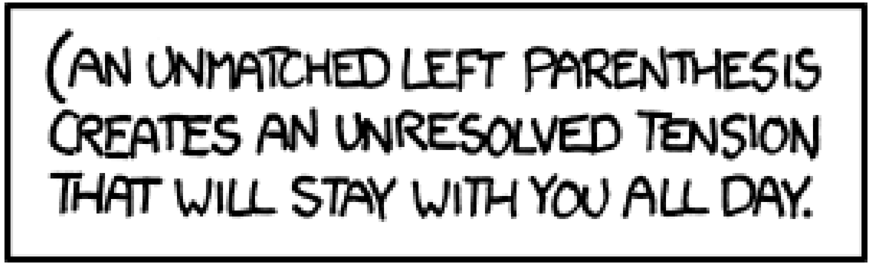
Part 2: Balanced

- Once you've gotten that working, you'll move to:
  - 2 - bool **checkOperators**(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 OR
      - 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

- Recursive process:
    - If your string is non-empty, here’s what you should 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!

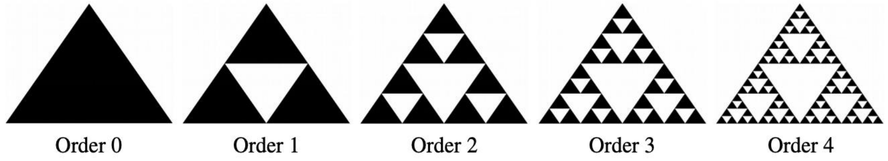
Questions about part 2?


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

Source, XKCD

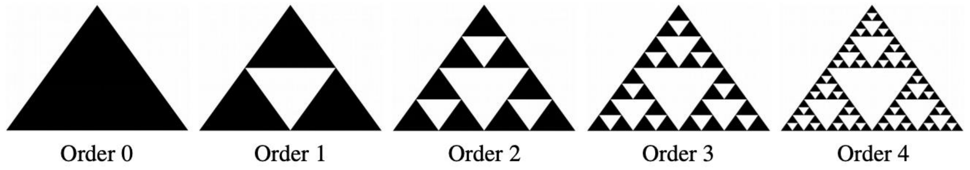
Part 3: Sierpinski

- 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.



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

Part 3: Sierpinski

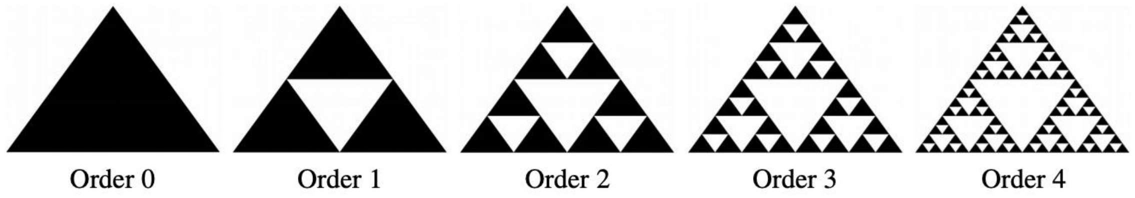


- Here are the two functions you'll be using:

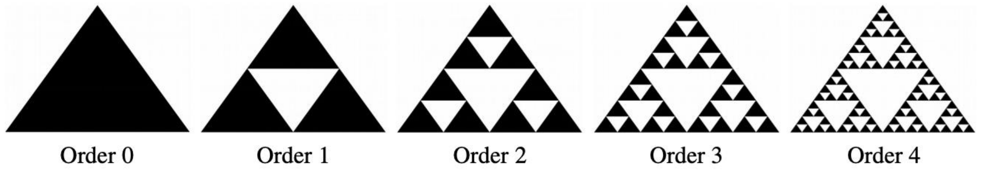
- void **drawSierpinskiTriangle**(GWindow& window, GPoint one, GPoint two, GPoint three, int order)
  - You'll need to implement this one!
- void **fillBlackTriangle**(GWindow& window, GPoint one, GPoint two, GPoint three)
  - A GWindow is just the console object that you'll be drawing on: you can ignore it :p

Part 3: 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.

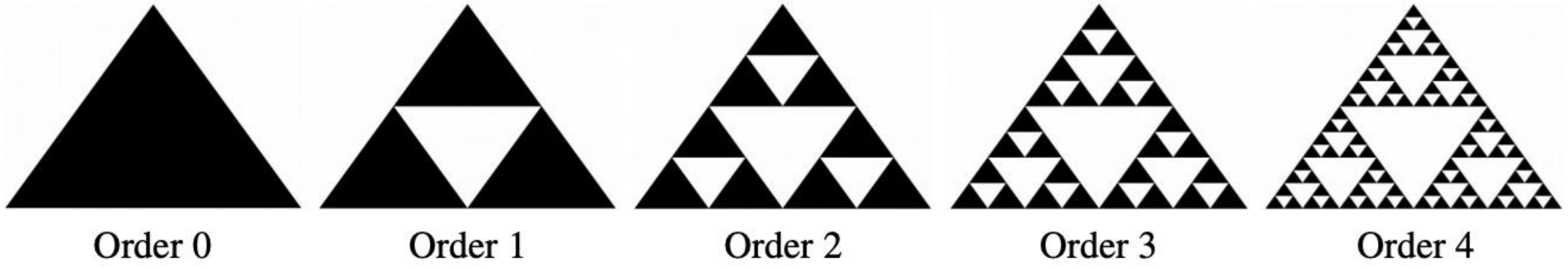


Part 3: Sierpinski

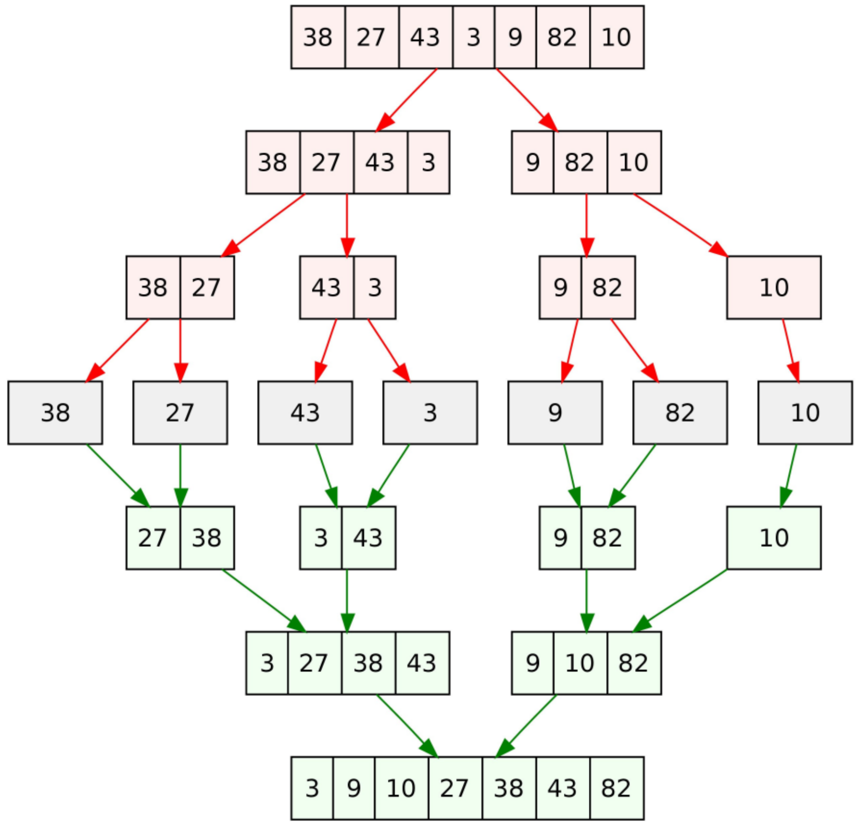


- Tips:
  - The GPoint object contains two **doubles**: x and y. To access them, use the **point.getX()** and **point.getY()** methods.
  - To declare a new GPoint, an easy way of doing so is using the {} brackets.
    - E.x. GPoint p = { 1.0, 2.0 };
  - To get the midpoint between two points, you can write
    - GPoint midpt = { (p1.getX() + p2.getX()) / 2, (p1.getY() + p2.getY()) / 2 };

Questions about part 3?



Any other questions?



Note: this is a diagram for **mergesort** on integers – you will be implementing something a little different!

Part 4: Multi-Merge

- 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.

Step 1: Implement the **merge** routine!

- In the previous example, you saw we were merging together individual integers to make pairs of integers, which would be merged etc. etc…
- What if we reimplemented this idea with **Queue**<int> instead of integers?
- Here's the idea: given a **Vector< Queue**<int> recursively split the **vector**, so you'll be merging **Queue**<int> instead of individual integers. That way, there's no difference between merging single integers and merging groups of integers – in your case, it'll just be **Queue**<int>'s of different sizes!
- The first thing we want you to do is to **implement the merge routine**, where, given two **Queue**<int>'s, you combine them into a single **Queue**<int> , sorted from smallest (front) to largest (back).

Step 1: Implement the **merge** routine!

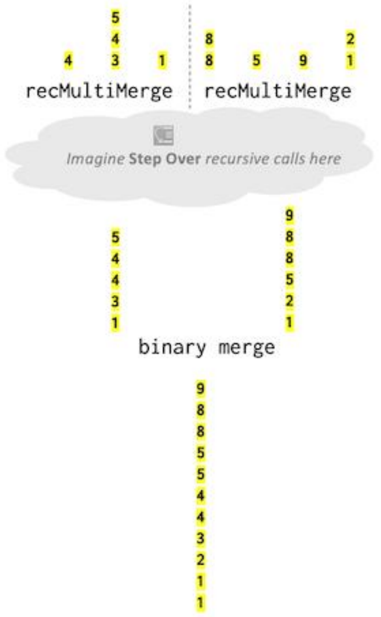
- Specifically, implement the **Queue**<int> **merge**(**Queue** <int> a, **Queue** <int> b) function, which accepts two **sorted** queues of integers and returns a single sorted queue.
- *Hint:* Think about dequeuing the first elements in each queue and comparing them – what can you assert about the smaller element with respect to every other element in either queue?
- **A few notes:**

1. We'd like you implement this iteratively, **not** recursively.
2. It is **not guaranteed that your queue's will be in sorted order**. Although most queue's will be, you are responsible for throwing an error if you encounter an element that is out of order
   - To do this, you could either attempt to verify both queues are sorted before you merge, or you could attempt to verify this *during* the merge. The latter is more efficient, but we think the former is more straightforward!
3. Be sure to account for the fact that you will often be merging queue's A and B that have different lengths!

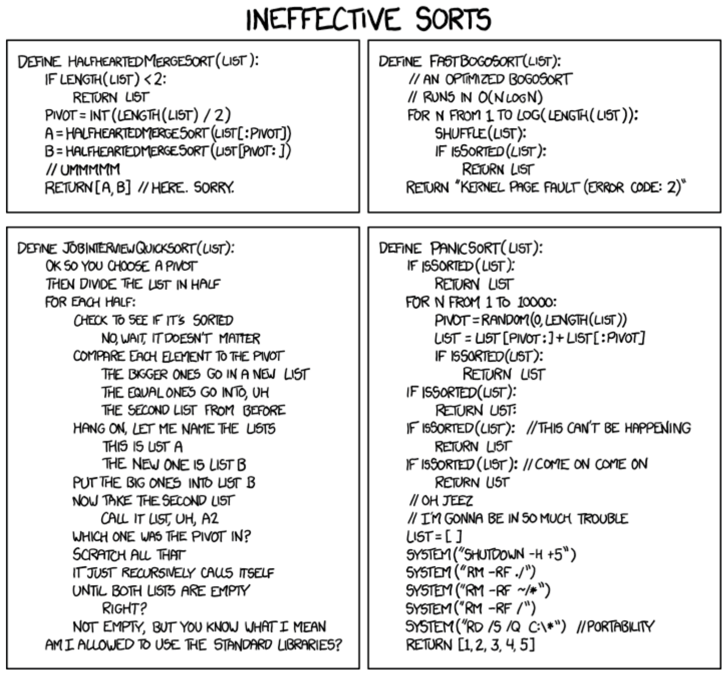
Step 2: Implement Multiway Merge

- Now it's time to harness the power of recursion to sort like a pro!
- You will implement the function **Queue**<int> **multiMerge**(**Vector**<**Queue**<int>>& all), which recursively splits **all** in half and merges the split halves. The handout tells you to proceed as so:

  1. Divide the input collection of k sequences into two halves. The "left" half is the first K/2 sequences in the vector, and the "right" half is the rest of the sequences.
     - The Vector subList operation can be used to subdivide a Vector, which you may find helpful.
  2. Recursively apply the multiway merge to the "left" half of the sequences to generate one combined sequence. Repeat the same process with the "right" half of the sequences, generating a second combined, sorted sequence.
  3. Use your binary merge function to join the two combined sequences into the final result, which is then returned.



Questions about merge?



```
DEFINE HALFHEARTEDMERGESORT(LIST):
    IF LENGTH(LIST) < 2:
        RETURN LIST
    PIVOT = INT(LENGTH(LIST) / 2)
    A = HALFHEARTEDMERGESORT(LIST[:PIVOT])
    B = HALFHEARTEDMERGESORT(LIST[PIVOT:])
    // UMMMMM
    RETURN [A, B] // HERE. SORRY.
```

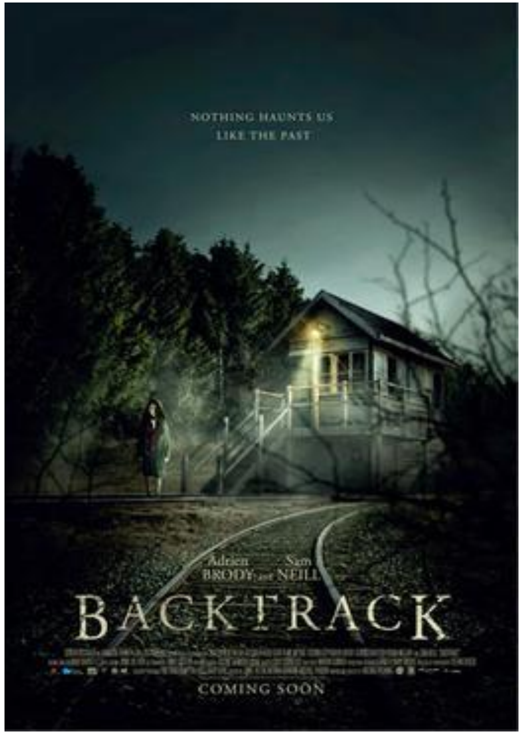
```
DEFINE FASTBOGOSORT(LIST):
    // AN OPTIMIZED BOGOSORT
    // RUNS IN O(N LOG N)
    FOR N FROM 1 TO LOG(LENGTH(LIST)):
        SHUFFLE(LIST):
        IF ISSORTED(LIST):
            RETURN LIST
    RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"
```

```
DEFINE JOBINTERVIEWQUICKSORT(LIST):
    OK SO YOU CHOOSE A PIVOT
    THEN DIVIDE THE LIST IN HALF
    FOR EACH HALF:
        CHECK TO SEE IF IT'S SORTED
            NO, WAIT, IT DOESN'T MATTER
        COMPARE EACH ELEMENT TO THE PIVOT
            THE BIGGER ONES GO IN A NEW LIST
            THE EQUAL ONES GO INTO, UH
            THE SECOND LIST FROM BEFORE
        HANG ON, LET ME NAME THE LISTS
            THIS IS LIST A
            THE NEW ONE IS LIST B
        PUT THE BIG ONES INTO LIST B
        NOW TAKE THE SECOND LIST
            CALL IT LIST, UH, A2
        WHICH ONE WAS THE PIVOT IN?
        SCRATCH ALL THAT
        IT JUST RECURSIVELY CALLS ITSELF
        UNTIL BOTH LISTS ARE EMPTY
            RIGHT?
        NOT EMPTY, BUT YOU KNOW WHAT I MEAN
    AM I ALLOWED TO USE THE STANDARD LIBRARIES?
```

```
DEFINE PANICSORT(LIST):
    IF ISSORTED(LIST):
        RETURN LIST
    FOR N FROM 1 TO 10000:
        PIVOT = RANDOM(0, LENGTH(LIST))
        LIST = LIST[PIVOT:] + LIST[:PIVOT]
        IF ISSORTED(LIST):
            RETURN LIST
    IF ISSORTED(LIST):
        RETURN LIST:
    IF ISSORTED(LIST):  //THIS CAN'T BE HAPPENING
        RETURN LIST
    IF ISSORTED(LIST): //COME ON COME ON
        RETURN LIST
    // OH JEEZ
    // I'M GONNA BE IN SO MUCH TROUBLE
    LIST = [ ]
    SYSTEM("SHUTDOWN -H +5")
    SYSTEM("RM -RF ./")
    SYSTEM("RM -RF ~/*")
    SYSTEM("RM -RF /")
    SYSTEM("RD /S /Q C:\*") //PORTABILITY
    RETURN [1, 2, 3, 4, 5]
```

xkcd

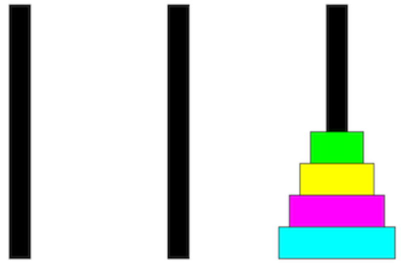
Now on to… Recusive backtracking!

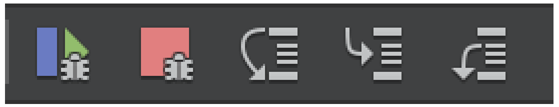


The poster for 2015 mystery-thriller "Backtrack." Critics gave it a 30% on Rotten Tomatoes, citing "not enough recursion."

Backtracking Warmups

- To ease you into the world of backtracking, we've given you **2** functions to examine in this warmup.
- The **first** exercise will explore **The Towers of Hanoi** example you saw in lecture.
  - In this part, you'll use the debugger features "**step over, step into, and step out**" to become a debugging pro!
  - You'll need to use these features to jump around the function's execution and report on the state of variables in different places!





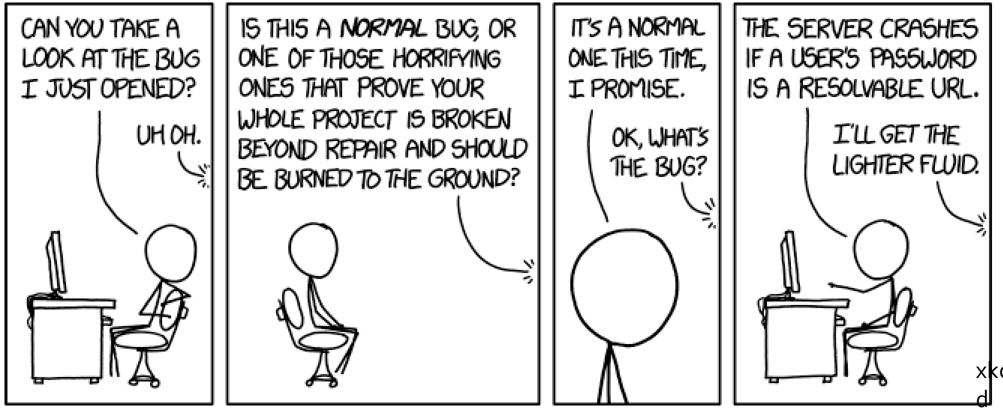
Backtracking Warmups

- For the second warmup, you will use student tests and the debugger to diagnose and fix a recursive function that counts the number of **subsets within a set whose elements sum to zero.**
- For example: The set **{3, 1, -3}** has subsets:
- **{ {3, 1, -3}, {3, 1}, {3, -3}, {1,-3}, {3}, {1}, {-3}, {} }**

  Of which only **{3, -3}** and **{}** have members who sum to zero.
- Right now, the function is buggy, and it's your job to figure out what's wrong. Examine the code to find out!

Questions about the Warmups?

- **Once again, we strongly recommend doing the warmups first. They are specifically made to help you in each assignment!**


CAN YOU TAKE A LOOK AT THE BUG I JUST OPENED?
UH OH.
IS THIS A NORMAL BUG, OR ONE OF THOSE HORRIFYING ONES THAT PROVE YOUR WHOLE PROJECT IS BROKEN BEYOND REPAIR AND SHOULD BE BURNED TO THE GROUND?
IT'S A NORMAL ONE THIS TIME, I PROMISE.
OK, WHAT'S THE BUG?
THE SERVER CRASHES IF A USER'S PASSWORD IS A RESOLVABLE URL.
I'LL GET THE LIGHTER FLUID.

xkcd

Part 5: Boggle Computer Search

- Who played Boggle?

Part 5: Boggle Computer Search

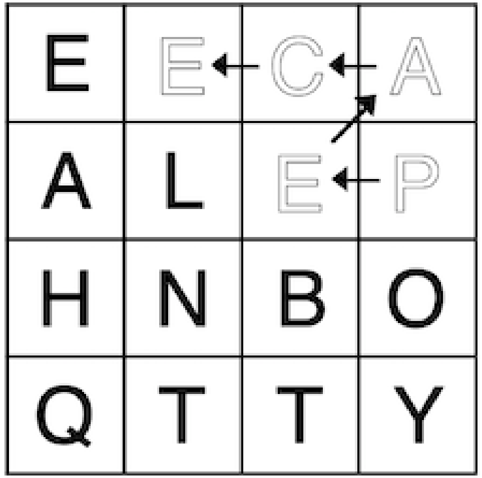
- In this part, you'll be asked to write the functionality for a **computerized Boggle player,** who uses a **Lexicon** and recursion to find every possible word on the Boggle board!
- More specifically, you'll be implementing 2 functions:
- int **points**(string str)
- Which returns the number of points awarded for string "str", and

int **scoreBoard**(Grid<char>& board, Lexicon& lex)

Which returns the maximum possible Boggle score (int) given the board.

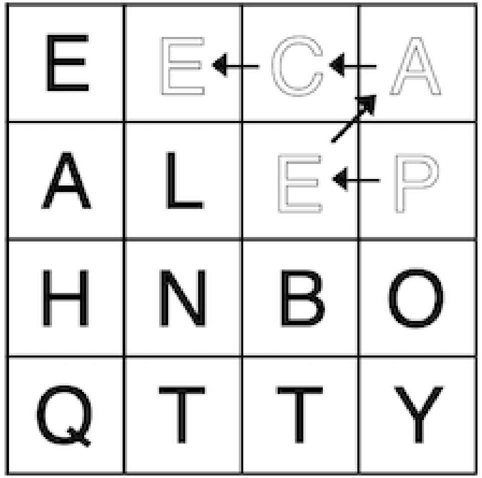
Part 5: Boggle Computer Search

- In order to compute the maximum score, you're going to use **recursive backtracking**. More specifically, you're going to need to examine every square in the Boggle board and find **all words** starting on that square.
  - This means that you should call your backtracking routine on every square on the board!



Part 5: Boggle Computer Search

- One such example of this routine is on the right: if we're starting at 'P', we can find some words by examining all of our neighbors: **if appending a neighbor's character leaves you with a valid prefix, repeat the exploration process, starting on that neighbor. If at any time your current string happens to be a valid word, be sure that you keep track of the points you earn!**
- **Be sure that you're not looking at places you've already been! If your string is 'pe' and you're looking for neighbors, don't consider 'p' again!**



Part 5: Boggle Computer Search

- A few more notes:
  - For scoring, a word must be at least 4 letters long. From there, the [length : score] relationship looks like this: [4:1], [5:2], [6:3], [7:4] and so on!
  - We've only given you these two functions. Do you have enough variables to solve this problem, or will you need a helper function?
  - You can only examine adjacent cubes. That's just Boggle, I guess.
  - For scoring, words are unique. This means that if a word exists multiple times on the board, its score will only count **once** in your point total.
  - When you find a word, do you want to end your search?

Part 5: Boggle Computer Search

- A few *hints:*
  - The **GridLocation** struct from A2 maze may very well come in handy here.
  - When needing to keep track of things like visited locations, I find the **Set** to be a great data structure.
  - Don't sleep on the **.inBounds()** function in the **Grid** class!
  - ^^ The same about the **.containsPrefix()** function for **Lexicon**'s! If you don't prune your decision tree, you're going to be taking too long (scoring a board should take less than a second!)
  - Get used to the double for loop syntax for the **Grid.** One way of accessing elements in a **Grid** while also knowing your coordinates (helpful if you're using **GridLocation**'s) is this:

```
for (int r = 0; r < board.numRows(); r++) {
    for (int c = 0; c < board.numCols(); c++) { // har.
        char boggleLetter = board[r][c];
    }
}
//Could you use something like this to look at your neighbors too? I wonder…
```

Questions about Boggle?

Last part: Option Challenge Question

- Have some time to spare this week? Looking for a tricky problem? We've got one for you!
- The last problem (which is **optional**) is a challenging recursive backtracking problem about voting power. When you have electoral blocks, do all votes matter? Your job is to find out!
- I'm not going to go into this one – if you want to do it, you're on your own ;)
  - I'll give you one hint: think about how to generate subsets recursively – you'll need to apply that logic to this program *without* actually creating the physical subsets!
- It might be a good problem to examine before your **midterm assessment** that is coming up!

Any last questions?

- Congrats! You’re now ready to be ~recursive~