YEAH A3: Recursion!!!

CS106B Summer '21: Assignment 3

Jin-Hee Lee && Grant Bishko

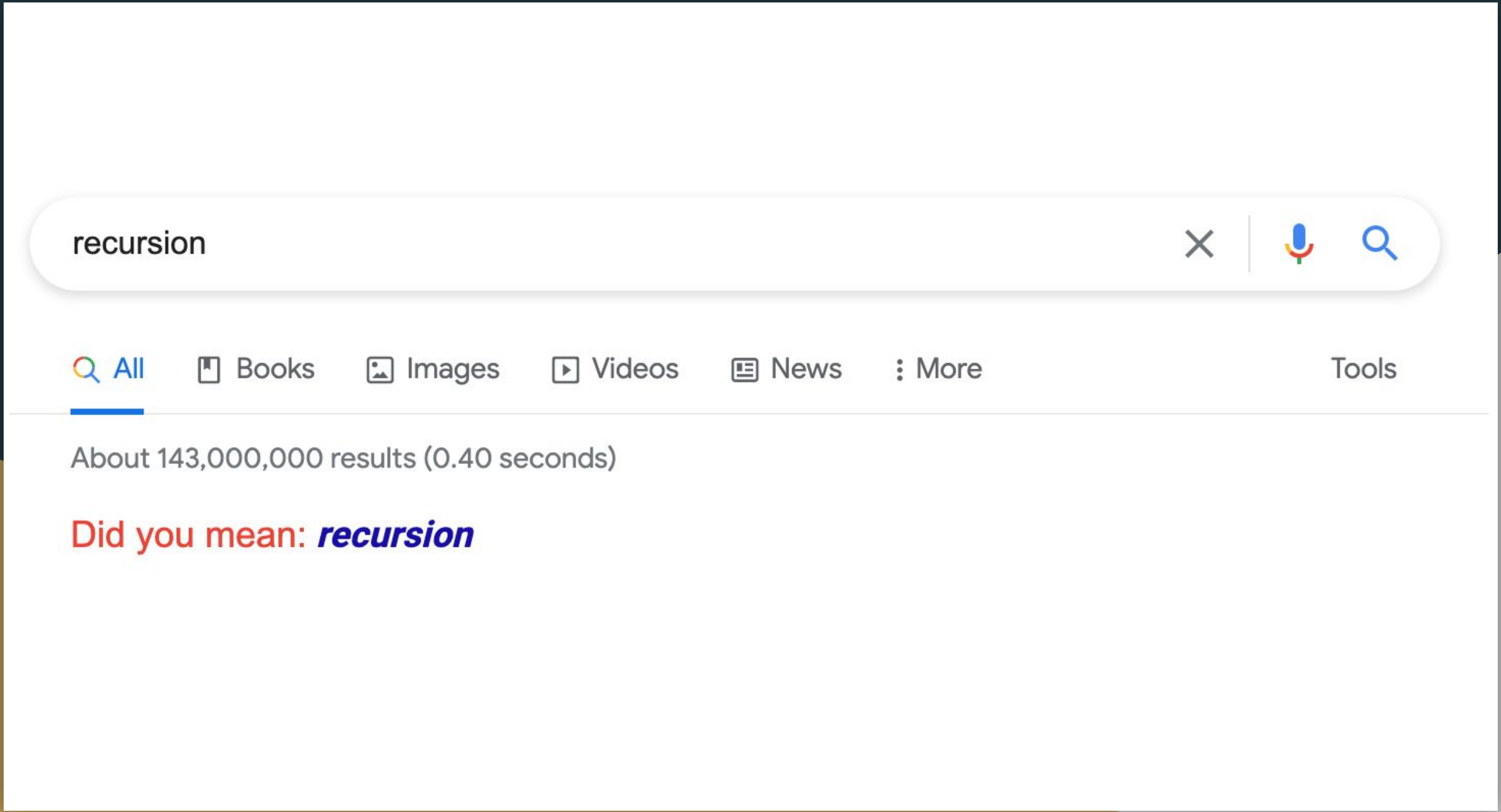
Assignment 3 ~Logistics~

A3 is due on **Thursday, July 15** at 11:59pm PT

Grace period until **Saturday, July 17** at 11:59pm PT

Note: this assignment has a lot of parts, so <u>start early!!</u>

(Seriously)


recursion
All
Books
Images
Videos
News
More
Tools
About 143,000,000 results (0.40 seconds)
Did you mean: recursion

Before anything, some wise words from **`char`** the charmander:

Recursion is all about breaking down a problem into smaller and smaller pieces until you reach the simplest case that you can't break down any further.

We should have a strong idea of how to break down the task at hand before even touching the keyboard.

To help, DRAW LOTS OF PICTURES AS YOU TACKLE THIS ASSIGNMENT!

Assignment 3

1. Sierpinski
2. Balanced
3. Mergesort
4. Boggle

But first! Warmups!

1) Examine Recursive Stack Frames
2) Recognize Stack Overflow (kind of like the recursive version of entering an infinite loop)
3) Test a Recursive Function
4) Debugging a Recursive Function

But first! Warmups!

Assignment 3

1. **Sierpinski**
2. Balanced
3. Mergesort
4. Boggle

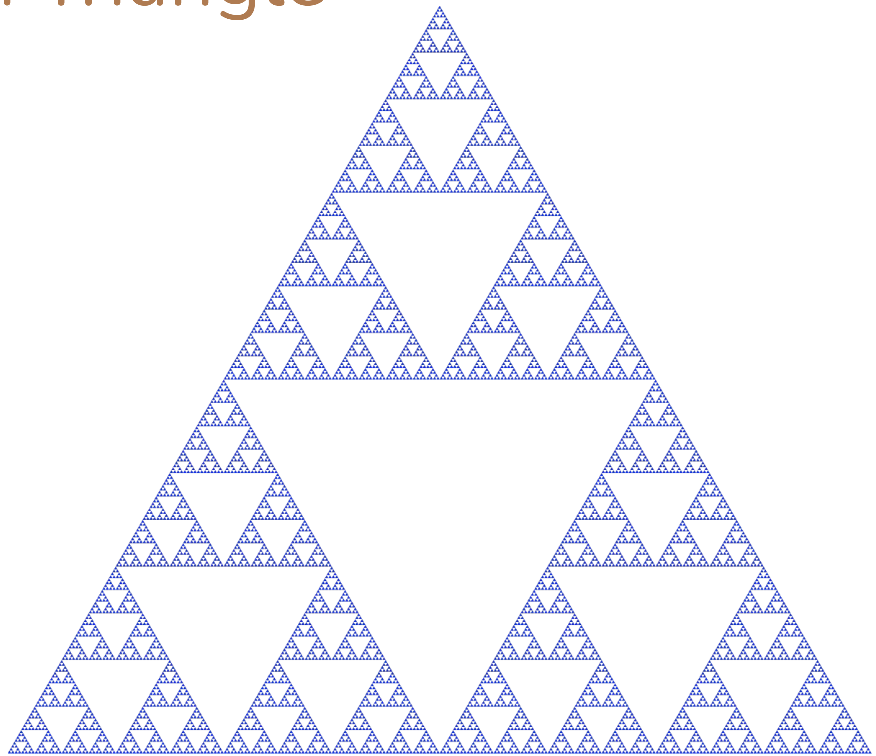
Sierpinski

A Polish Mathematician (1882-1969)

Famous for triangles!

Huh?

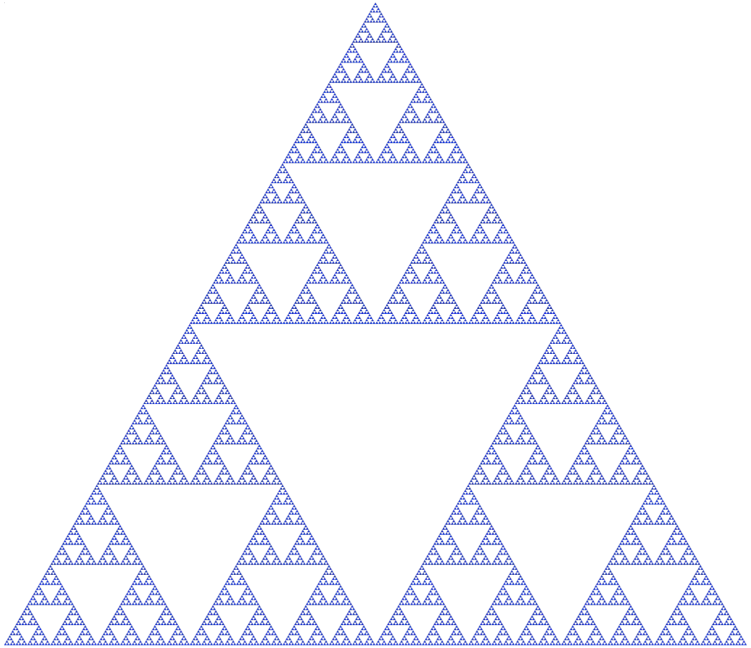
Sierpinski Triangle



Sierpinski Triangle

We can use *recursion* to create incredible graphic designs of **fractals** (images that have self-similar subparts)

Sierpinski Triangle is a famous example of fractals!

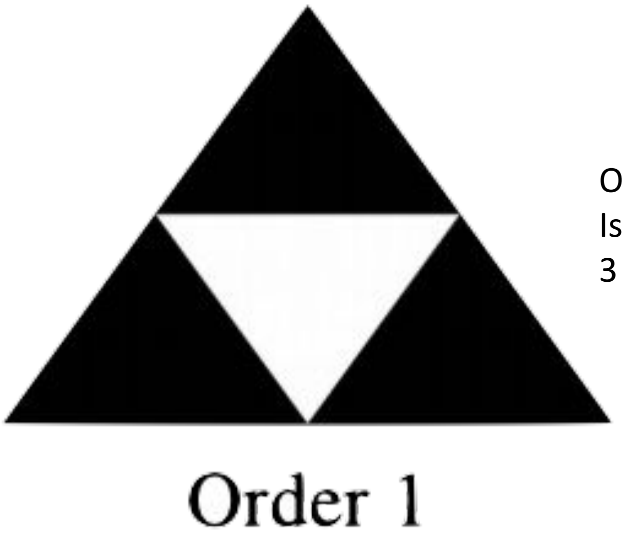


Sierpinski Triangle

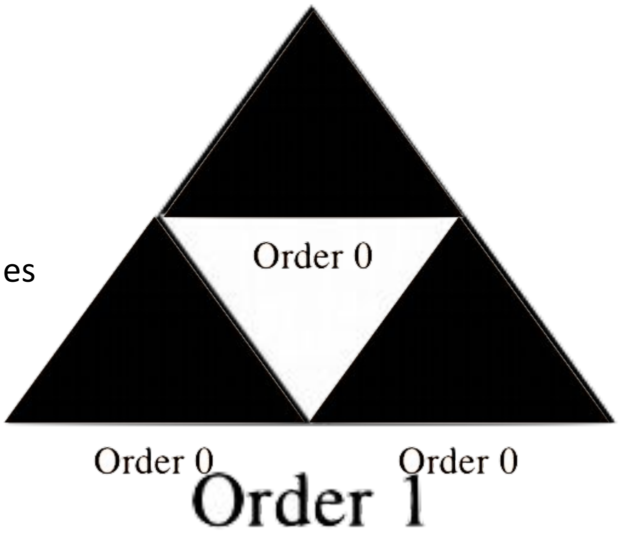
Sierpinski Triangles are defined recursively:

- 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 $n - 1$, whose side lengths are half the size of the original side lengths, arranged so that they meet corner-to-corner.

Sierpinski Triangle



Order-1
Is the same as:
3 order-0 triangles

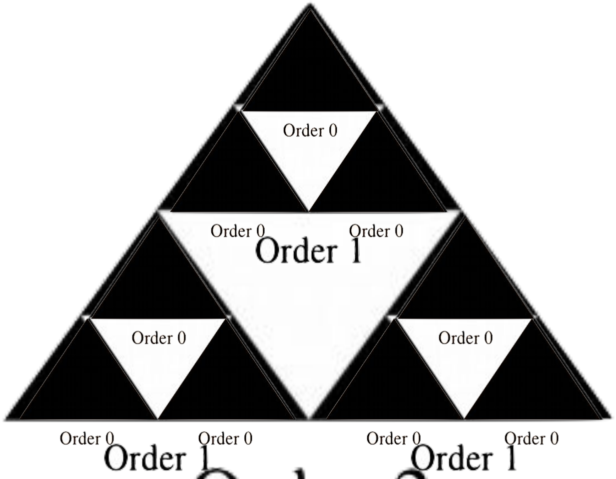


Sierpinski Triangle

Order 2

Order-2
Is the same as:
3 order-1 triangles
Which is the same as:
9 order-0 triangles

And so on…



Order 2

Sierpinski Triangle

You could say:

- **Drawing a sierpinski triangle of Order N is just like saying:**
    - Drawing a sierpinski triangle of Order N-1
    - Drawing a sierpinski triangle of Order N-1
    - Drawing a sierpinski triangle of Order N-1

(With different side lengths and positions of course)

Sierpinski Triangle

- **Drawing a sierpinski triangle of Order N is just like saying:**
  - **Drawing a sierpinski triangle of Order N-1**
    - Drawing a sierpinski triangle of Order N-1
    - Drawing a sierpinski triangle of Order N-1
    - Drawing a sierpinski triangle of Order N-1
  - **Drawing a sierpinski triangle of Order N-1**
    - Drawing a sierpinski triangle of Order N-1
    - Drawing a sierpinski triangle of Order N-1
    - Drawing a sierpinski triangle of Order N-1
  - **Drawing a sierpinski triangle of Order N-1**
    - Drawing a sierpinski triangle of Order N-1
    - Drawing a sierpinski triangle of Order N-1
    - Drawing a sierpinski triangle of Order N-1

Sierpinski Triangle

- **Drawing a sierpinski triangle of Order N is just like saying:**
  - **Drawing a sierpinski triangle of Order N-1**
    - **Drawing a sierpinski triangle of Order N-1**
      - Drawing a sierpinski triangle of Order N-1
      - Drawing a sierpinski triangle of Order N-1
      - Drawing a sierpinski triangle of Order N-1
    - **Drawing a sierpinski triangle of Order N-1**
      - Drawing a sierpinski triangle of Order N-1
      - Drawing a sierpinski triangle of Order N-1
      - Drawing a sierpinski triangle of Order N-1
    - **Drawing a sierpinski triangle of Order N-1**
      - Drawing a sierpinski triangle of Order N-1
      - Drawing a sierpinski triangle of Order N-1
      - Drawing a sierpinski triangle of Order N-1
  - **Drawing a sierpinski triangle of Order N-1**
    - **Drawing a sierpinski triangle of Order N-1**
      - Drawing a sierpinski triangle of Order N-1
      - Drawing a sierpinski triangle of Order N-1
      - Drawing a sierpinski triangle of Order N-1
    - **Drawing a sierpinski triangle of Order N-1**
      - Drawing a sierpinski triangle of Order N-1
      - Drawing a sierpinski triangle of Order N-1
      - Drawing a sierpinski triangle of Order N-1
    - **Drawing a sierpinski triangle of Order N-1**
      - Drawing a sierpinski triangle of Order N-1
      - Drawing a sierpinski triangle of Order N-1
      - Drawing a sierpinski triangle of Order N-1
  - **Drawing a sierpinski triangle of Order N-1**
    - **Drawing a sierpinski triangle of Order N-1**
      - Drawing a sierpinski triangle of Order N-1
      - Drawing a sierpinski triangle of Order N-1
      - Drawing a sierpinski triangle of Order N-1
    - **Drawing a sierpinski triangle of Order N-1**
      - Drawing a sierpinski triangle of Order N-1
      - Drawing a sierpinski triangle of Order N-1
      - Drawing a sierpinski triangle of Order N-1
    - **Drawing a sierpinski triangle of Order N-1**
      - Drawing a sierpinski triangle of Order N-1
      - Drawing a sierpinski triangle of Order N-1
      - Drawing a sierpinski triangle of Order N-1

See how this is recursive?? :)

Sierpinski Triangle

You will be implementing a function:

```
int drawSierpinskiTriangle(GWindow& window, GPoint one, GPoint two, GPoint three, int order)
```

Graphics window that will show the visuals

Three points of a triangle
GPoint is {x, y} coordinates

order number

You will be calculating the number of triangles you draw (recursively!)

Sierpinski Triangle

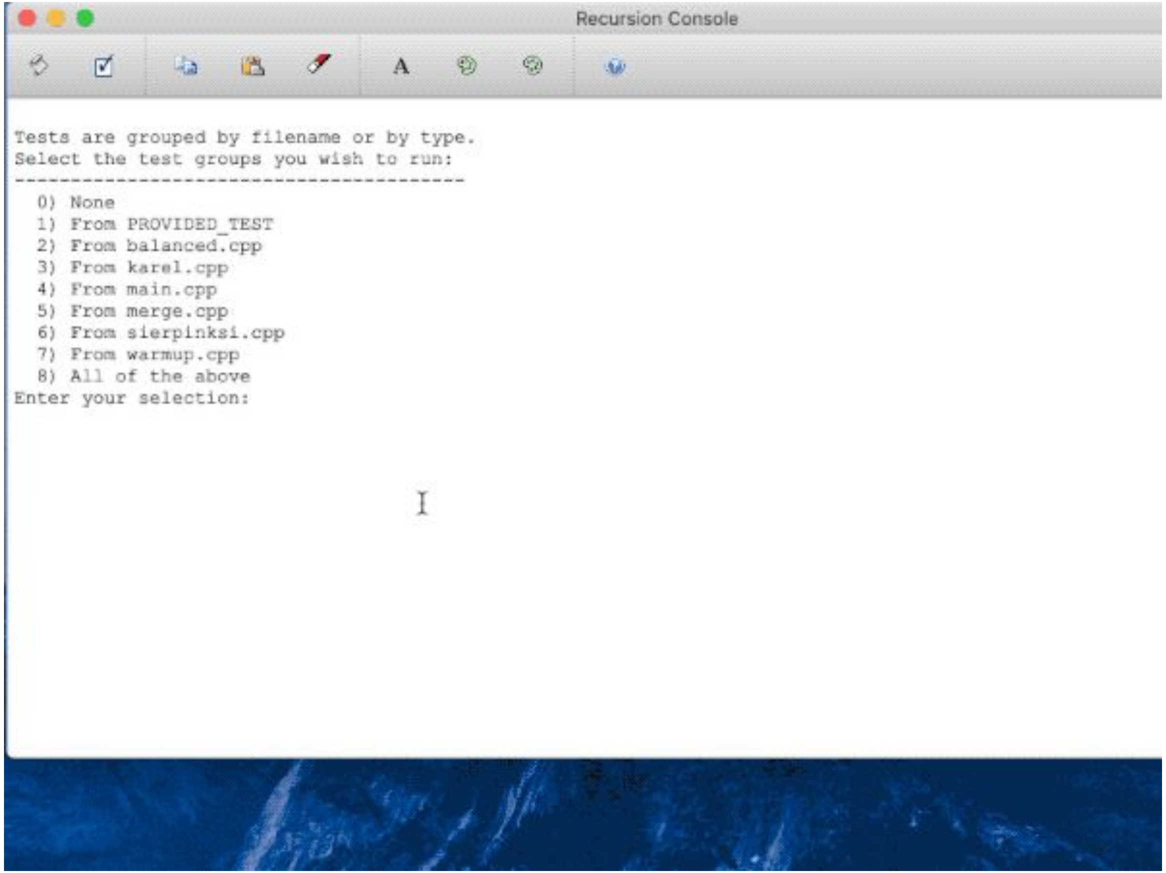
We provide you with the function:

```
void fillBlackTriangle(GWindow& gw, GPoint one, GPoint two, GPoint three)
```

Which draws a triangle onto the graphics window, given the window, and all three points.

(Hint: you only want to call this function **once** in drawSierpinskiTriangle (base case??))

Trust your recursion to draw each triangle in each correct position!


Recursion Console
A
Tests are grouped by filename or by type.
Select the test groups you wish to run:
----
0) None
1) From PROVIDED_TEST
2) From balanced.cpp
3) From karel.cpp
4) From main.cpp
5) From merge.cpp
6) From sierpinksi.cpp
7) From warmup.cpp
8) All of the above
Enter your selection:

Sierpinski Triangle Tips/Tricks

- Draw pictures!! This will be SUPER helpful in figuring out the location for each triangle corner!
- The Midpoint of each side of the outer larger triangle becomes a *corner* of one of the smaller inner triangles.
    - Given a line segment whose endpoints are {x1, y1} and {x2, y2}, its midpoint is located at { (x1+x2)/2, (y1+y2)/2 }
- Your triangles do not need to be equilateral!
- A **GPoint** struct is used to specify a x,y coordinate within the graphics window. Here is the documentation for GPoint.
- The only drawing function you need is our provided **fillBlackTriangle**. You do not need to dig into any other drawing functionality in the Stanford libraries.

Questions about Sierpinski?

Assignment 3

1. Sierpinski
2. **Balanced**
3. Mergesort
4. Boggle


PERFECTLY BALANCED
AS ALL THINGS SHOULD BE
memegenerator.net

Balanced

In most programming languages, we see *bracketing operators* such as:

`( . . . )` -- Parenthesis (?)

`[ . . . ]` -- Square Bracket (?)

`{ . . . }` -- Curly Bracket (?)

Balanced

Let's look at how they are used!

```
int main() {
    int x = 2 * (vec[2] + 3); x = (1 + random());
}
```

Balanced

Let’s look at how they are used!

```
int main() {
    int x = 2 * (vec[2] + 3); x = (1 + random());
}
```

Operators only → `(){([])(())}`

Yes, the operators are correctly balanced!

Balanced

What is an “unbalanced” pair?

```
( ( [ a ] )   The line is missing a close parenthesis.
3 ) (         The close parenthesis comes before the open parenthesis.
{ ( x } y )   The parentheses and braces are improperly nested.
```

Balanced

```
bool isBalanced(string str) {
    string ops = operatorsOnly(str);
    return checkOperators(ops);
}
```

- Takes in a string str
- Strips down that string to be *just the operators*
- Returns true/false depending on the output of checkOperators -- which takes in the bracket-only string

Balanced -- Your Task

You will be writing two functions:

1) `string operatorsFrom(string str)`
   a) Takes in a string, returns the string with only the brackets
2) `bool operatorsAreMatched(string ops)`
   a) Takes in a string, returns true/false if it's balanced

The catch? You will be implementing both of these recursively!

Remove everything but the Brackets

You’ve done this iteratively… let’s do it recursively:

Similar to the reverse recursive function you’ve seen in section/lecture!

- Process the first character of the string and determine (if it exists, and) if it should be kept as part of the outputted string
- Recursively process the **rest** of the string (hint: use substrings!)

Be sure to thoroughly test your function before moving on. You should add at least 2-3 tests of your own to validate the behavior of your operatorsFrom function.

operatorsAreMatched

As you think about implementing this recursive function:

A string consisting of only bracketing characters is balanced if and only if one of the following conditions holds:

- The string is empty.
- The string contains **"()"**, **"[]"**, or **"{}"** as a substring and the rest of the string is balanced after removing that substring.

Let’s go over that intuition!

Let’s look at the string:

“[(){}]”

Let’s go over that intuition!

Let’s look at the string:

“[(){}]”

We can find the substring “()” inside the above string, so let’s remove it

Let’s go over that intuition!

Let’s look at the string:

“[{}]”

Let’s go over that intuition!

Let’s look at the string:

“[{}]”

We can find the substring “{}” inside the above string, so let’s remove it

Let’s go over that intuition!

Let’s look at the string:

“[]”

Let’s go over that intuition!

Let’s look at the string:

“[]”

We can find the substring “[]” inside the above string, so let’s remove it

Let’s go over that intuition!

Let’s look at the string:

“”

Let’s go over that intuition!

Let’s look at the string:

“”

We’ve reached the empty string!! Hmmm… what could this mean??

What happens if we didn’t reach the empty string, and couldn’t find “()” or “{}” or “[]” as substrings?? Hmmm…

Questions about Balanced?

Assignment 3

1. Sierpinski
2. Balanced
3. **Mergesort**
4. Boggle

MergeSort

So far, we've been given ways to sort our data structures:

- v.sort() for a Vector
- Maps sort their keys automatically in increasing order

Now, it's time for us to take matters into our own hands and use the power of recursion to write a sorting algorithm called **merge sort!**

Binary Merge

Implementing a *binary merge* operation is very common in CS

- Merging two individually sorted sequences into one sorted sequence

You will be implementing:

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

Binary Merge

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

- Takes in 2 queues (both in sorted increasing order), and combines them!
  - If they aren’t in sorted order throw an error!
- Both queues inputted by value (copy), so feel free to modify them, doesn’t matter what they look like after the function is run
- Not necessarily the same lengths
- No special case with duplicates
- To be done **iteratively**, not with recursion.

Binary Merge Tips/Tricks

- Go through each queue (iterating through queues is funky remember!)
- Examine the element at the front of each one, enqueue the smaller one into a new Queue (the one you are returning)
- Don’t forget about q.peek() vs. q.dequeue()! Use peek to compare!
- You can either check that queues are in sorted order by:
  - Making a helper that throws an error if it finds an element out of order
  - Do it in your merge function, keeping track of a curr and prev elements!

Big-O Analysis of Iterative Merge

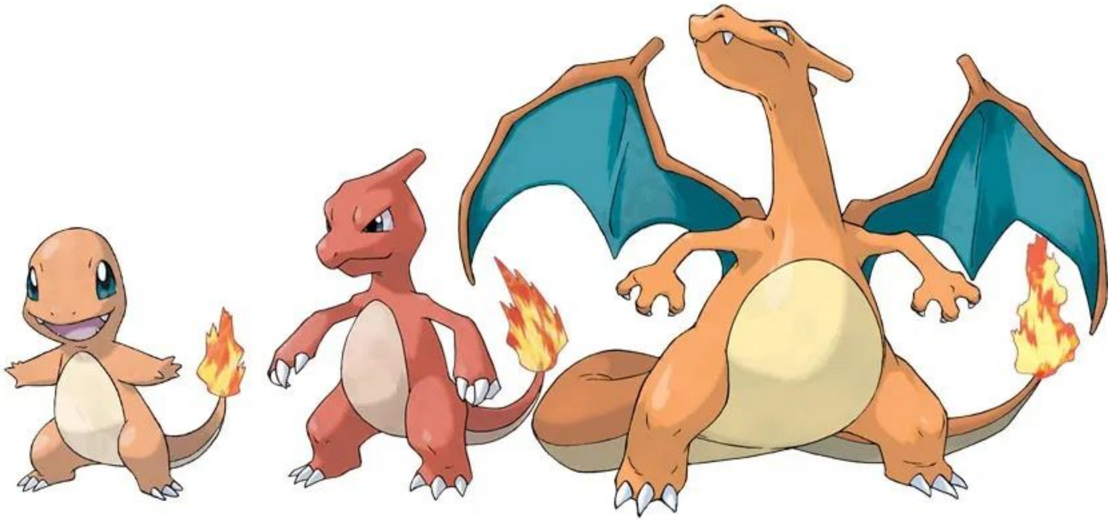
You will be asked to predict/calculate a Big-O analysis of the Iterative Merge method!

- Use Time Trials here! Find the relationship between size of queues and times!
- Experiment with varying sizes

Q11. Include the data from your execution timing and explain how is supports your Big O prediction for binary **`merge`**.

Questions about binary merge?

We can do better than that! Level up!



We can do better than that! Level up!

RECURSION

Recursive MultiMerge

A multi-way merge takes the "divide-and-conquer" strategy to recursively merge a sequence.

Especially for longer sequences, this approach is **much more efficient**.

Our job is to write

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

Recursive MultiMerge Algorithm

The handout says it best:

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. 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, generating 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.

Implementation Tips

- Consider your base cases:
- You take in a Vector<Queue<int>> as a parameter, right?
    - What are the smallest possible cases we might get for an input of this type?
    - How should we deal with them?

Implementation Tips

- Trust the recursion.
  - Recursively merge the left and the right
  - Merge the 2 final resulting queues into 1!
- Remember: your recursion should lead you to your base cases
- Recursion is all about breaking down our problem into smaller pieces
  - What are the pieces that are getting smaller with each recursive call?
- As long as you follow the handout, you should be in good shape!

Questions about multi-way merge?

Assignment 3

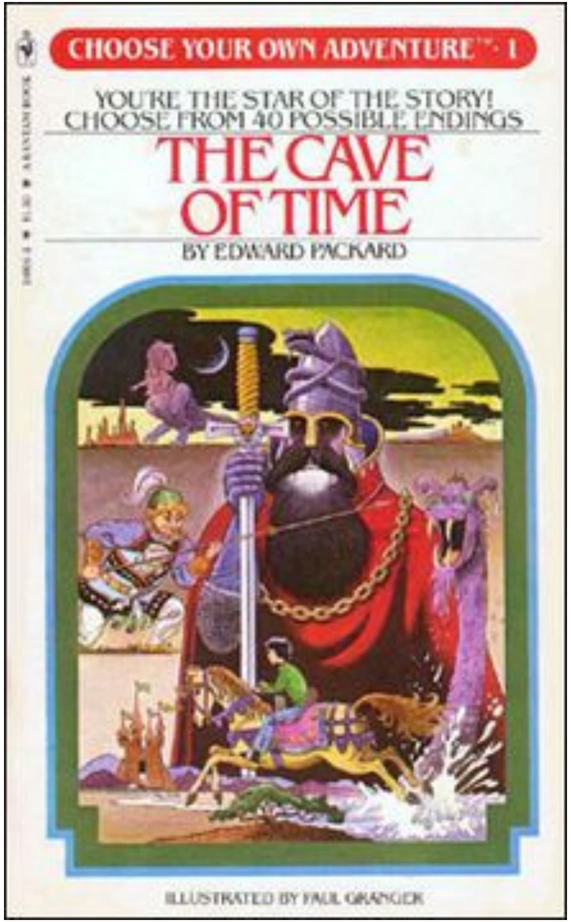
1. Sierpinski
2. Balanced
3. Mergesort
4. **Boggle**


One does ~~not~~
simply...
bactrack...
quickmeme.com

Remember these? →

Choose-Your-Own-Adventure can be thought of as recursive backtracking.

We **choose** one path, **explore** it, then **unchoose** it if we do not reach a solution/happy ending.



Recursive Backtracking

Recursive backtracking helps us solve problems for which we need to try many different options to find the desired solution.

If we make a step in the wrong direction, we need to backtrack -- undo thats step -- and choose a different step.

To construct a backtracking solution, you'll likely have to recurse on all possible options. If you reach the end of an option, you've found a solution!


Boggle

The Game of Boggle

In this game, each player must try to find as many words as possible on the Boggle Board. Each word has a corresponding score depending on its length.

We will be coding a computerized Boggle player, whose goal is to find the maximum possible score!

The Game of Boggle

Of course, we will harness the power of recursive backtracking!

- Look at each letter on the board
- Find all words that begin with that letter

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

The function operates by recursively exploring the board to find all words and tallying up the points.

Boggle Rules

- A word can only build up using a character that is adjacent (4 cardinal directions NSEW, **and** diagonals NE, SE, NW, SW)
- Each word has a score that contributes to the point total

Big Idea 1: Backtracking

Our backtracking solution should recurse on **all possible options**, since we need to examine every possible word on the board.

This means every single letter-cube on the board (every single location on the Grid) must be examined as the first letter of a word.

Then, we can proceed in the adjacent cubes around that letter-cube and build up words!

Scoring

- A word must be at least 4 letters long to be scored
  - length 4 $\longrightarrow$ 1 point
  - length 5 $\longrightarrow$ 2 points
  - length 6 $\longrightarrow$ 3 points … etc.
- Only *unique* words should contribute to the score
  - Even if the word "hello" happens to appear in 3 different variations on the board, the word only counts once in the score total

Can this word-so-far contribute to my score?

Yes, if…

- It is at least 4 letters long
- It is a valid word, according to a function from the Lexicon class
- The word can be traced through adjacent letter cubes
- The word has not yet been seen / contributed to the score total

Big Idea: Marking

By this point, hopefully it's clear that we should be keeping track of the words we've already scored.

Hint: we will also need to keep track of locations we've already visited on the board, to avoid going back to a character that has already been added to the word.

This means we'll be "marking" locations as seen as we reach them.

Once we've reached a word...

- Great!
- Food for thought: should we stop recursing here?
    - Example: if I've found "TREAT" and I stop here,
    - will I ever find "TREATY"?

Some tips to keep in mind...

- Using **`GridLocation`**s makes it much easier to look at any given point on the board (a Grid)
- Keep track of visited locations using a data structure that doesn't allow duplicates 🤔
- Careful not to go out of bounds… **`g.inBounds()`** sounds nice here 🤔
- Take advantage of the functions in the Lexicon class that give you information about a specified word!
    - Very helpful in determining full words and prefixes so we can **prune**

Prune, you say?

Prune, you say?

Pruning

- Say you are building up a word, and that word so far is "xbqmah".
- Even if there are still more letter blocks to visit on the Boggle board, *what's the point?*
  - We'd be doing ourselves (and our program) a disservice by continuing down a path that we know is a dead end from the start.
- The good news is, the Lexicon class has a function **`containsPrefix(wordSoFar)`** that tells us whether some wordSoFar is a prefix of any word. Be sure to make use of it!


WAIT A MINUTE
WHO ARE YOU?
FlukyFeed

We've talked about many different pieces of information.

We've talked about many different pieces of information.

But the only information we get is the 2 parameters:

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

That isn't enough to keep track of all the information we need!

The helper function - wrapper function relationship is **key** in recursive backtracking.

Often, we need to keep track of information as we go down a path, meaning we need to have all of the appropriate parameters.

Consider how you might use this approach in your solution!

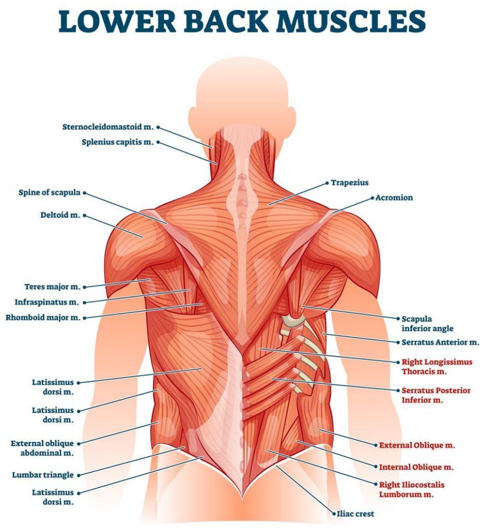
- What pieces of information do I need to have access to within each function call?

An infamous meme from Trip:

Questions about Boggle?

Final questions?

Good luck



-ing!