

YEAH Hours A3

Slides by Trip Master



Welcome to the
world of **Recursion!**

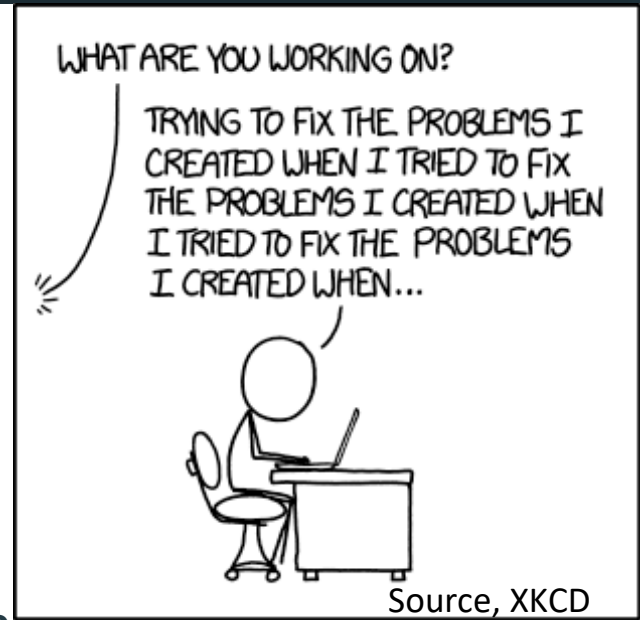
-Professor Oak, CS106B alum
and recursion pro

YEAH...what?

- YEAH (**Y**our **E**arly **A**ssignment **H**elp) hours were conceived many years ago to give students a boost when starting assignments early.
- During YEAH hours we'll review Assignment 3, and students will be able to ask clarifying questions along the way! (Please ask questions!)
- **There is a limit** to what I can answer! While I'll happily clarify and parts of the assignment, I cannot answer pointed implementation questions about the assignment -> we'll leave that part to all of you ;)
- I highly recommend having the handout open while you read these slides!

This week.... Recursion!

- 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!

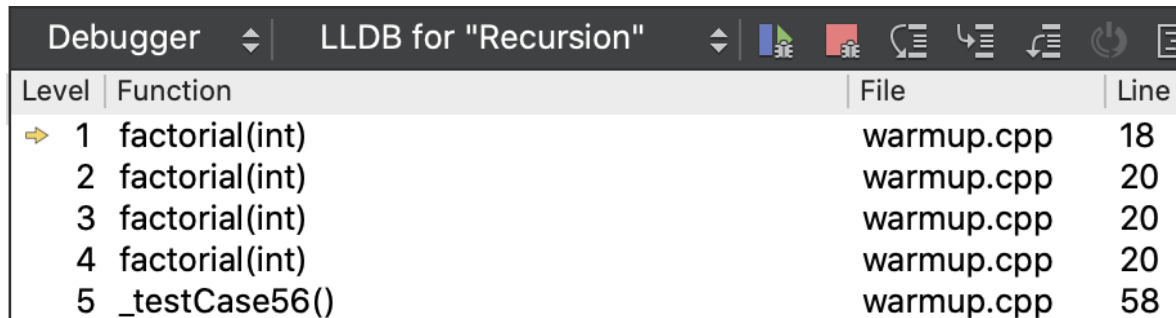


Let's take a look at Assignment 3

- 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. **Combinations** - Given a set of size **N** and a number of things to choose **K**, count the number of possible combinations
 - 4. **Sierpinski** - The ol' faithful triangle drawing program that has brought thousands of 106B students into the recursive world
 - 5. **Text Predict** - Implement a recursive T9-style predictive algorithm to guess what words a user will type on a keypad!

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 $n * \text{factorial}(n-1)$ 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?
 - **Hint #2:** If you want to print the output of `power()`, which is a `double`, `#include` the "strlib.h" library and look for `doubleToString()`!

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



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...

Part 2: Balanced

- You will need to implement two functions:
 - 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.
 - 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!

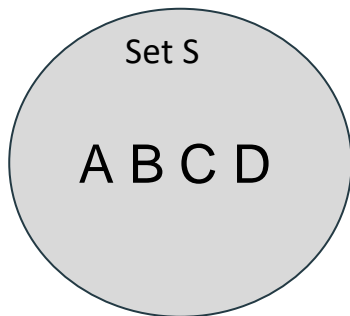
Questions about part 2?

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

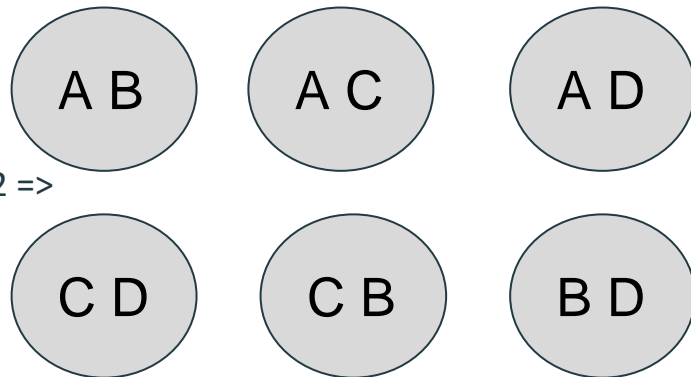
Source, XKCD

Part 3: Combinations

- In this part, you will write a function `int count_combos(int n, int k)` that counts the number of possible combinations you could make by choosing k distinct items from a container of size n .
 - Think: *if I have k Cardi 106B tickets and n friends, how many **unique combinations of friends can I bring?***



=> Combinations of 2 =>



Part 3: Combinations

- Don't worry, we give you an algorithm for this :)
 - Consider counting combinations to be a series of choices:
 - For each element x in n , you have a choice: you can either include x in your 'chosen' group (meaning you have $k - 1$ choices and $n - 1$ options to choose from for your next choice), or you can choose to omit x from your 'chosen' group (meaning that you have k choices remaining and $n - 1$ options to choose from in your next choice).

Part 3: Combinations

- (Once again): For each element x in n , you have a choice: you can either include x in your 'chosen' group (meaning you have $k - 1$ choices and $n - 1$ options to choose from for your next choice), or you can choose to omit x from your 'chosen' group (meaning that you have k choices remaining and $n - 1$ options to choose from in your next choice).
- The number of combinations you can make is the sum of *both* options (i.e. what would happen if you choose to omit choice x PLUS what would happen if you choose to include choice x).

$$\text{numCombos} = \text{tryIncludingX} + \text{tryExcludingX}$$

Part 3: Combinations

- A few tricky things:
 - Ensure that your formula returns an **error()** for certain inputs.
 - Can you choose 10 things from a set of 9 elements?
 - How about negative inputs?
 - **Hint:** If you're confused about how combinations work, try out the provided **combo_formula()**, which uses the factorial() function from the warmups to calculate combinations!
 - Make sure you've fixed the bug in it though :)
 - Base cases can be tricky here -- for what values of **n** and **k** do you know that you're done recursing?

Part 3: Combinations

- Conceptually: how can I think about distinct elements when I'm just given some integers? There's no actual set... right?
 - By subtracting 1 from n every time you make a choice, and 1 from k every time you choose to include, you are effectively making each "choice" one that concerns a distinct element.

Questions about part 3?

- Highly recommend giving the handout an extra-good parse on this one -- it's not that bad if you follow the approach given, but it could be tricky to conceptualize!



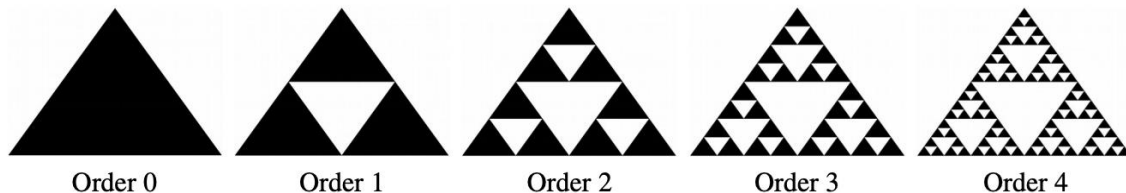
okurrrrrrr!

Cardi 106B, upon hearing your robust implementation strategy for recursive combination counting)

Part 4: Sierpinski

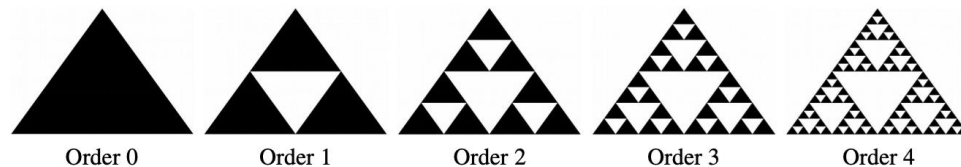


- In this part, you'll be asked to draw *n-order* Sierpinski Triangles, named after Polish Mathematician Wacław Sierpiński.
- The Sierpinski triangle is defined *recursively*, meaning:
 - An order-0 Sierpinski triangle is a plain filled triangle.
 - An order- n Sierpinski triangle, where $n > 0$, consists of three Sierpinski triangles of order $n - 1$, each half as large as the main triangle, arranged so that they meet corner-to-corner.

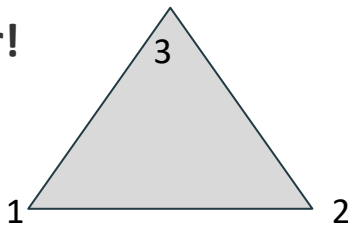


- `void drawSierpinskiTriangle(GWindow& window, GPoint one, GPoint two, GPoint three, int order)`
- `void fillBlackTriangle(GWindow& window, GPoint one, GPoint two, GPoint three)`

Part 4: Sierpinski

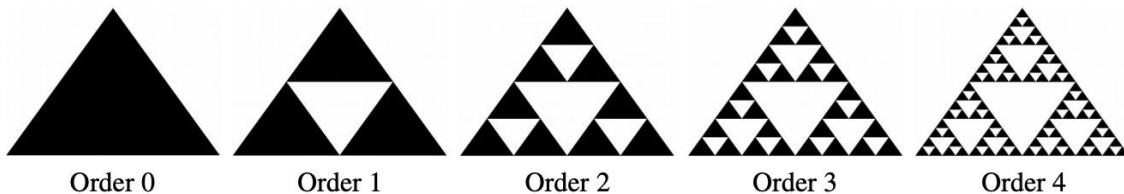


- `void drawSierpinskiTriangle`(GWindow& window, GPoint one, GPoint two, GPoint three, `int` order)
- `void fillBlackTriangle`(GWindow& window, GPoint one, GPoint two, GPoint three)
 - A GWindow is just the console object that you'll be drawing on: you can ignore it :p
 - The GPoints are ordered **like such**. **If you do something else, you'll get strange behavior!**



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.

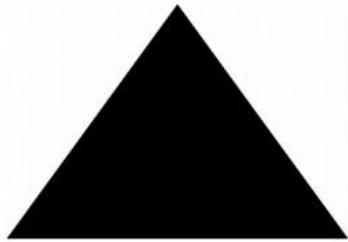


Part 4: Sierpinski

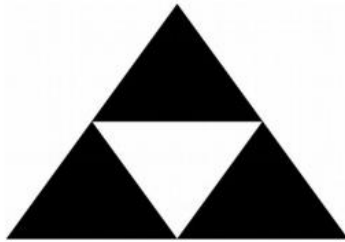


- Tips:
 - The GPoint object contains two **doubles**: x and y. To access them, use the **point.getX()** and **point.getY()** methods.
 - To declare a new GPoint, an easy way of doing so is using the {} brackets.
 - E.x. `GPoint p = { 1.0, 2.0 };`
 - To get the midpoint between two points, you can write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

Questions about part 4?



Order 0



Order 1



Order 2



Order 3



Order 4

Part 5: Text Predict

Let's throw it back...



Part 5: Text Predict

- These were 10 digit keypads! ... aaand they sucked. They were impossible to type words on. They were so unusable that **algorithms** were developed to try and predict what people were typing -> to put them out of their misery.
- Your task is to implement one of them: the Tegic T9 algorithm!

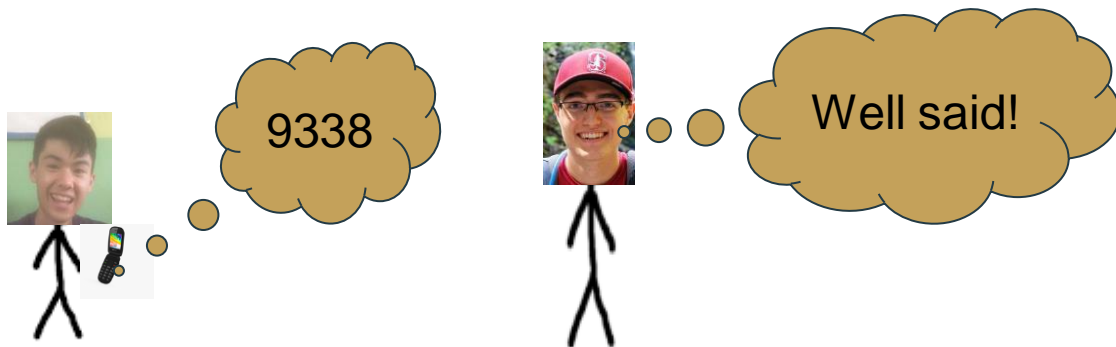


But before we do the algorithm...

- A few data structures to review...
- **Lexicon** -> A structure optimized for string storage. Like an actual dictionary, this data structure is excellent at determining whether a word exists within its context ($O(\log(n))!!!$), and it has some other cool features!
- **bool contains**(string s) -> A **Lexicon** method that returns true if string s exists within the **Lexicon**.
- **bool containsPrefix**(string s) -> A **Lexicon** method that returns true if string s is a valid prefix of a string within the **Lexicon**.
- **Map<int, string> keypad** = `{{2,"abc"}, {3,"def"}, {4,"ghi"}, {5,"jkl"}, {6,"mno"}, {7,"pqrs"}, {8,"tuv"}, {9,"wxyz"}}`;
- Treat this Map like a constant!

Part 5: Text Predict

- Let's say I send a 1 word text to Nick B on my dinosaur phone using the numbers "9338." There are finite number of 4-letter words that I could have sent Nick... let's step through the algorithm to figure out what I sent him!



Text Predict Algorithm - decode (9338)

1. Take the first character from the input string -> '9'. '9' corresponds to the string "wxyz". We know our word starts with one of these letters! Try them all, starting with 'w'.
2. If we consider 'w' the first letter, let's repeat the process and take the next digit -> '3', which corresponds to "def". Let's again try all characters, starting with "d". See the pattern?
3. We now have "wd" as our word. Unfortunately, we check our handy lexicon, and it tells us that "wd" isn't the prefix for any word, so we need to go back and try the next letter, "e".
4. "We" is in fact a valid prefix, so we can now process the next '3', which once again maps to "def". We try 'd' first.
5. We now have the string "wed". Sadly, we aren't done because I sent 4 characters to Nick. We'd have to keep exploring with '8' and "tuv".

Text Predict Algorithm

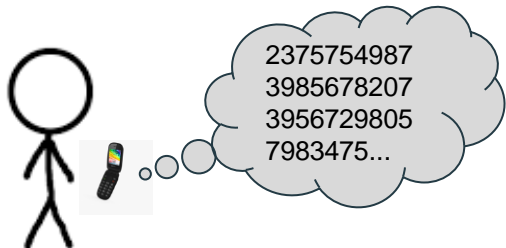
- At a certain point in time (unless I entered hopeless garbage), we'll hit a path in which we have a string of length 4 and no remaining digits. If so, we should look at our constructed string. If the lexicon tells us that our constructed string is a valid word, add it to a `Set<string>` of possible options
 - It's important that we add it to a container, because there could be many possible valid outcomes!
- Eventually, we will exhaust all of our options. At that time, our `Set<string>` will contain all possible options! What did I send to Nick? I'll leave that as an exercise up to the reader ;)

```
void predict(string digits, Set<string>& suggestions, Lexicon& lex, string sofar)
```

Part 5: Text Predict

- A few notes about this problem:
 - You're given significantly less direction on this problem in the handout. Be extra careful with testing!
 - You should skip non-digits. If I text “##9#338###”, you should be able to clean out the non-digits.
 - Apart from string cleaning, you shouldn't need a recursive helper function for this part.
 - Be sure to use containsPrefix()! If you don't, things could get slow...
 - Be mindful of the data you modify in this function. If you have the **sofar** string “w”, and you're considering adding one of each {d,e,f}, make sure that if “wd” fails to be a valid prefix, you remove ‘d’ before adding ‘e’ and trying again -- this is easy to miss!

Questions about Text Predict?



Ooh whatcha
say...
(creds to Imogen
Heap!!)



Wikipedia

.json Derulo, CS106B
alum and predictive
algorithm fiend

Any other questions?