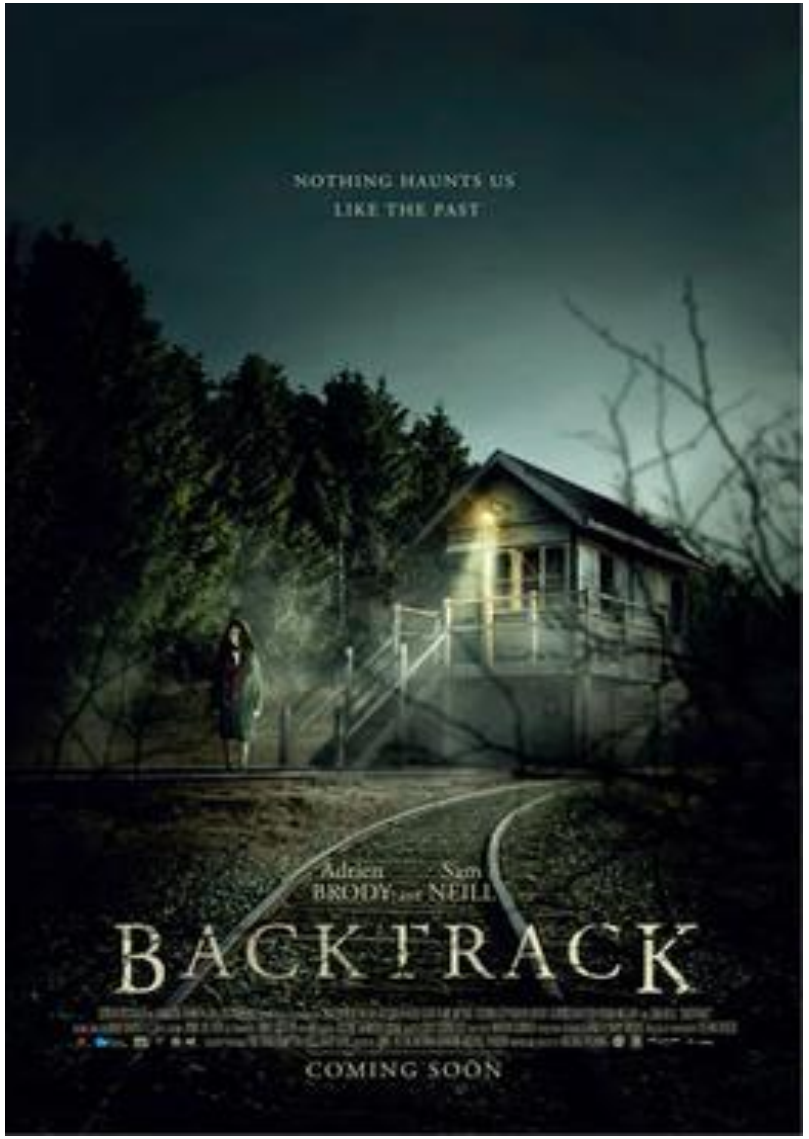


The poster for 2015 mystery-thriller “Backtrack.” Critics gave it a 30% on Rotten Tomatoes, citing “not enough recursion.”

YEAH (Your Early Assigment Help) A4

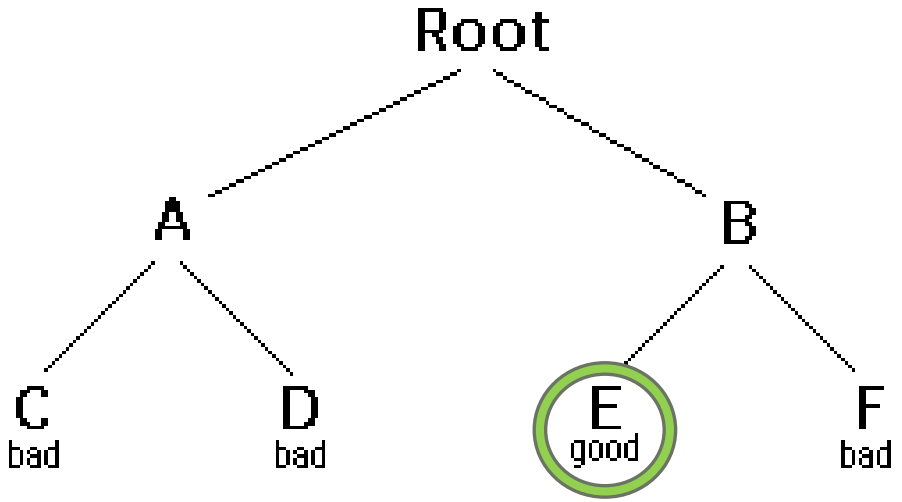
BACKTRACKING

What is Recursive Backtracking?

Recursive Backtracking is a recursive technique that attempts to find a solution to what I like to call a **difficult** problem.

Difficult problems are problems where you need to try many different options in order to find the correct answer. If you make a recursive step in the wrong direction, you must '**undo'** that step and go in a different direction.

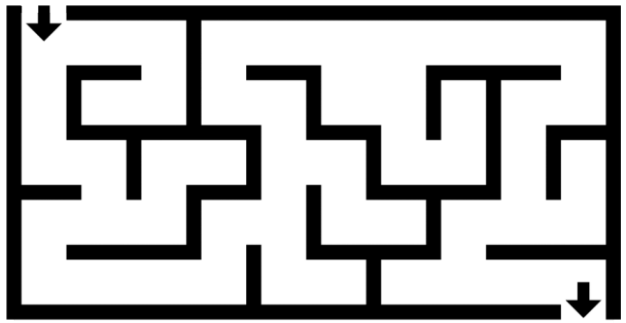
Backtracking frequently answers the question: *can this be done?* That's why many backtracking functions return bool



What are **Difficult Problems?** (recursively)

Backtracking problems are sometimes described as “trying to find a needle in a haystack.” In order to construct a backtracking solution, you typically have to recurse on **all possible options**, and if any recursive path yields success, return that particular option.

I like to think of **a maze** as a classic Backtracking example. If you were dropped in a maze, one way to escape would be to start going in a random direction until you hit a dead end, and then backtracking until you can go in an unexplored direction, repeating until you reach an exit or have exhausted every option. This is actually called a **Depth First Search** (DFS), and it’s often taught with BFS like you did on A2!



Today's Plan

0. Assignment Logistics (and why this week is special!)
1. Warmup (and why the past informs the present :o)
2. Multi-Merge (and why sorting will never be the same)
3. Boggle Score (and why computers are cheaters)
4. Banzhaf Power Index (and why democracy is broken ☹)

But first! Some logistics

For assignment 4, **you are only required to submit 2 of the 3 parts along with the warmups**.

I'll say that again: **you must submit the warmup. After that, you need only submit 2 of the following 3 parts.**

If you do all 3 parts, we'll give you some extra credit ☺

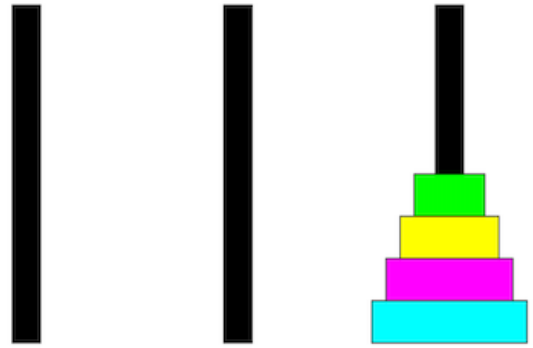
Regardless, we recommend looking at **all parts of the assignment anyway!**

And I'll go over them all right now ☺

Part 1: Warmups

We highly recommend you do the warmups first. I understand that they're not meant to be as intense as the coding assignments, but we give you the warmup **because it will equip you with tools and knowledge you'll need to complete the coding assignment. Please do it first!**

You will be debugging **2** recursive functions! The first exercise will give you practice using the debugger's "step in/out/over" functionality on a classic recursive problem called **The Towers of Hanoi.**



You then 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}, {} }

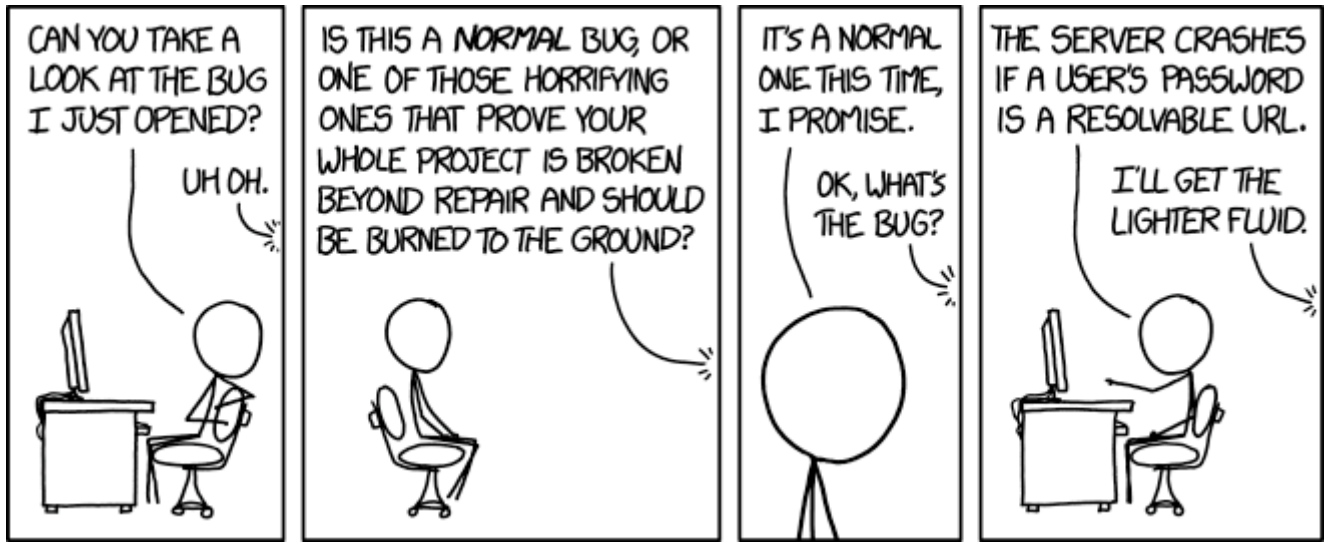
Of which only **{3, -3}** and **{}** have members who sum to zero.

These exercises will get you in great debugging shape for the rest of the assignment!

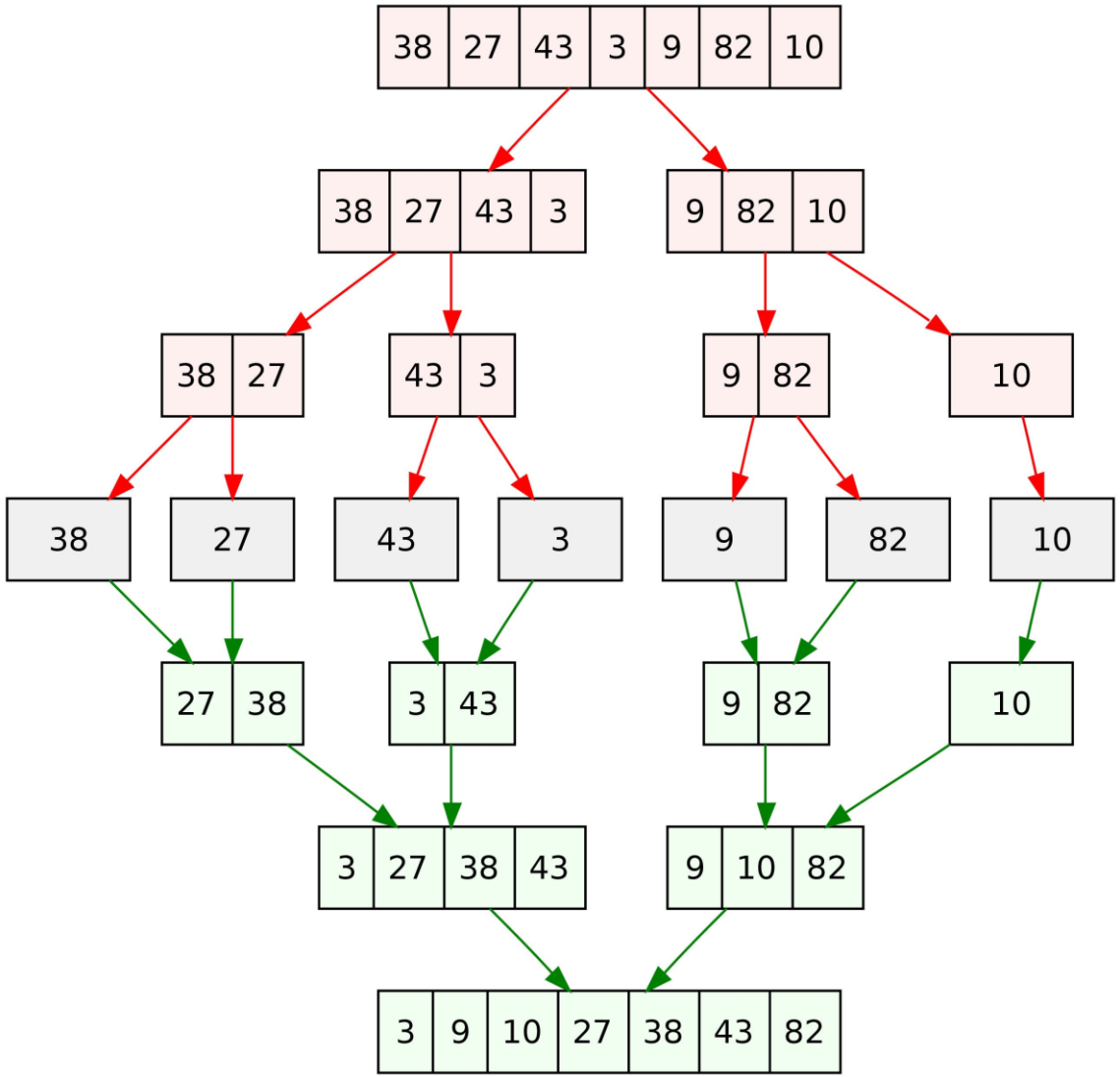
Questions about the Warmups?

I specifically left these less detailed so that you'd spend lots of time reading up on all of that good debugging info ☺

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



xkcd



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

Part 2: 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!

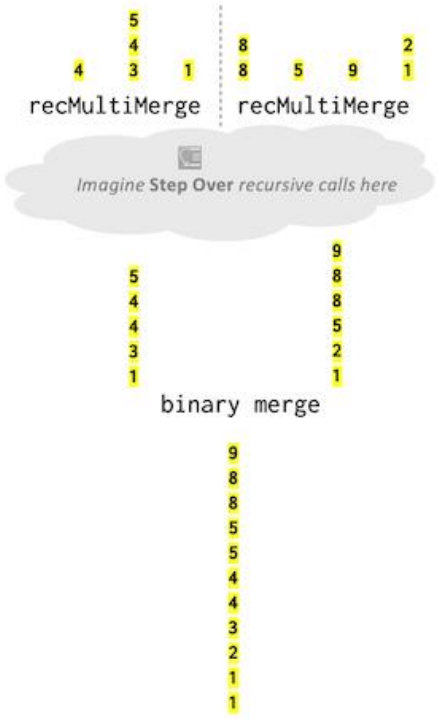
Step 2: Implement Multiway Merge

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.

That's it! A surprisingly simple algorithm with incredibly fast results!



Remember to test this function rigorously. What should you do with the empty vector, for example?

Questions about merge?

xkcd

Part 3: Boggle Computer Search

Who played Boggle?

Part 3: 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)
```

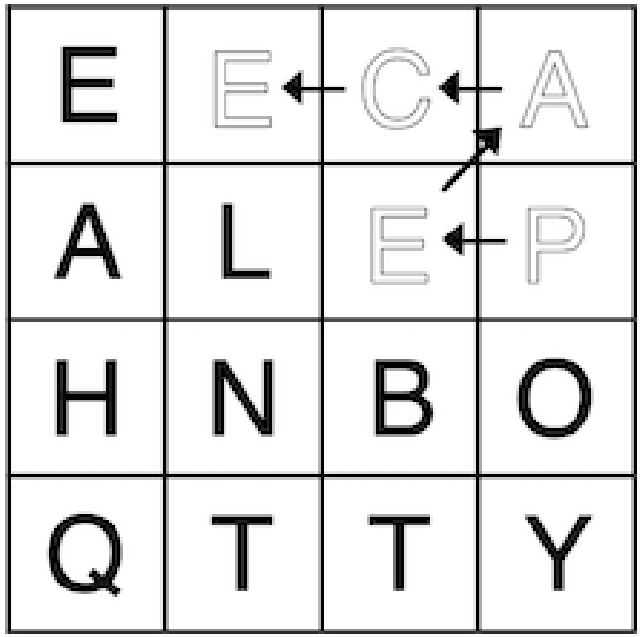
Which returns the maximum possible Boggle score (int) given the board.

Part 3: 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.

One such example 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 3: 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 3: 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 **HashSet** 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?

Part 4: Banzhaf Power Index

In this final part, you will answer the age-old question: *do all votes count equally?*

In this final part, instead of looking at individual voters, we're going to look at voting **blocks**, which are groups that have associated **block counts**, indicating their voting power. In order to pass any given vote, blocks can form **coalitions**, namely, any possible subset of the blocks, to vote in favor of something.

A vote count over the strict majority (TOTAL / 2 + 1) indicates a win!

Block	Block Count
Lions	50
Tigers	49
Bears	1

Vote power: 99

Vote power: 1

Part 4: Banzhaf Power Index

More specifically, given a **Vector** of constituent groups called “blocks”, each with their own share of total votes, compute the **Banzhaf Power Index**, which expresses a block's voting power as the percentage of situations in which this block is a critical voter (i.e. if they were not in a particular voting coalition, would said coalition fail to attain a strict majority of all votes).

All coalitions are: { **{ L, T, B }, {L, T}, {L , B}, {T, B}, {L} , {T}, {B}, {}** }

Winning coalitions: { **{ L, T, B }, {L, T}, {L , B}** }

Block	Block Count
Lions	50
Tigers	49
Bears	1

Block	Critical Votes
Lions	3
Tigers	1
Bears	1

Block	Banzhaf Power Index
Lions	60% (= 3/5)
Tigers	20% (= 1/5)
Bears	20% (= 1/5)

Part 4: Banzhaf Power Index

Let's take a closer look at these pictures and our approach:

1. We're given 3 blocks, codified as a **Vector<int>.** For each block, determine how many **critical votes** it has (we'll discuss this algorithm in depth on the next slide). A **critical vote** is one that would sway an election: the participation of the block in question would determine the outcome of the race in either direction.
2. Once we have a collection of each block's **critical vote** count, we want to divide each block's **critical vote** count by the total number of **critical votes** and multiply that by 100 to get a percentage share of critical votes. That's the **Banzhaf Power Index!**

Block	Block Count
Lions	50
Tigers	49
Bears	1

Block	Critical Votes
Lions	3
Tigers	1
Bears	1

Block	Banzhaf Power Index
Lions	60% (= 3/5)
Tigers	20% (= 1/5)
Bears	20% (= 1/5)

Part 4: Banzhaf Power Index

Let's talk a little more about finding a block's **critical vote** count. In a nutshell, what you want to do is, for each block, count the number of times that its vote would swing an election.

In other words, can you use recursion to

1. Generate all possible voting coalitions **without** the specified block, and then

2. For each generated coalition whose total vote count is less than the strict majority, see if adding the specified block's vote count would push the coalition's vote count above the strict majority. (Don't consider coalitions whose vote counts are higher – adding our vote wouldn't sway anything!)

This is not an easy ask, so we're going to explore it in more depth in the next slide!

Part 4: Banzhaf Power Index

In order to generate the aforementioned hypothetical coalitions, you should consider a recursive algorithm to generate **subsets**. This function will return the number of **critical votes** a particular block has.

1. Take your block out of the mix **before** you do anything (you could leave yourself in, but remember you want to generate these coalitions **without** your votes).
2. For each remaining block in the set (no loops, just recursion ☺), you want to return the number of **critical votes** you would receive if you **included it and excluded it in the coalition**. These are two separate recursive calls, and making them both will ensure that you account for **all** possible coalition subsets.
3. In order to do this ^ you'll need to keep track of the number of votes generated by a particular coalition (probably as a parameter). If this number goes over **the strict majority**, you should probably return 0 – your vote will not be a critical vote in this scenario.
4. Whenever you've run out of remaining blocks to consider including or excluding, it means that you have constructed enough votes to represent one of the hypothetical coalitions (sort of like your "done" base case). At this point, check if adding your block's votes to the coalition's votes would push you over the strict majority. If so, return 1!! You've found a scenario in which you provided the **critical vote!**

Part 4: Banzhaf Power Index

Some implementation notes:

- For the purposes of this problem, only a **strict majority** (>= TotalVotes / 2 + 1) will win an election. Ties are losses in this cruel, cruel world.
- When you're trying to actually find the index, recall that you're going to be dividing integers (critical votes for a block / total critical votes). Casting one of these two as a (double) should avoid any truncation issues.
- It's very important that you follow the aforementioned subset algorithm on this problem. I tried to implement a solution using a literal subset generation algorithm (one that creates a Vector<Vector<int>>, literally containing all subsets), instead of using the cheeky and friendly integer-oriented algorithm described before. I literally allocated so much memory that I blew out my heap – if your program crashes on the EU test with the error bad_alloc(), it means that you're using data structures where you shouldn't be!
- The recursive function you have to write here is tricky – be sure to write tests for it individually. If you can verify that it is correctly returning the number of **critical votes** for a given block, the problem is basically complete!
- Read the handout carefully on this one. It's trickier than the others, so be sure to be extra careful!

Any questions?