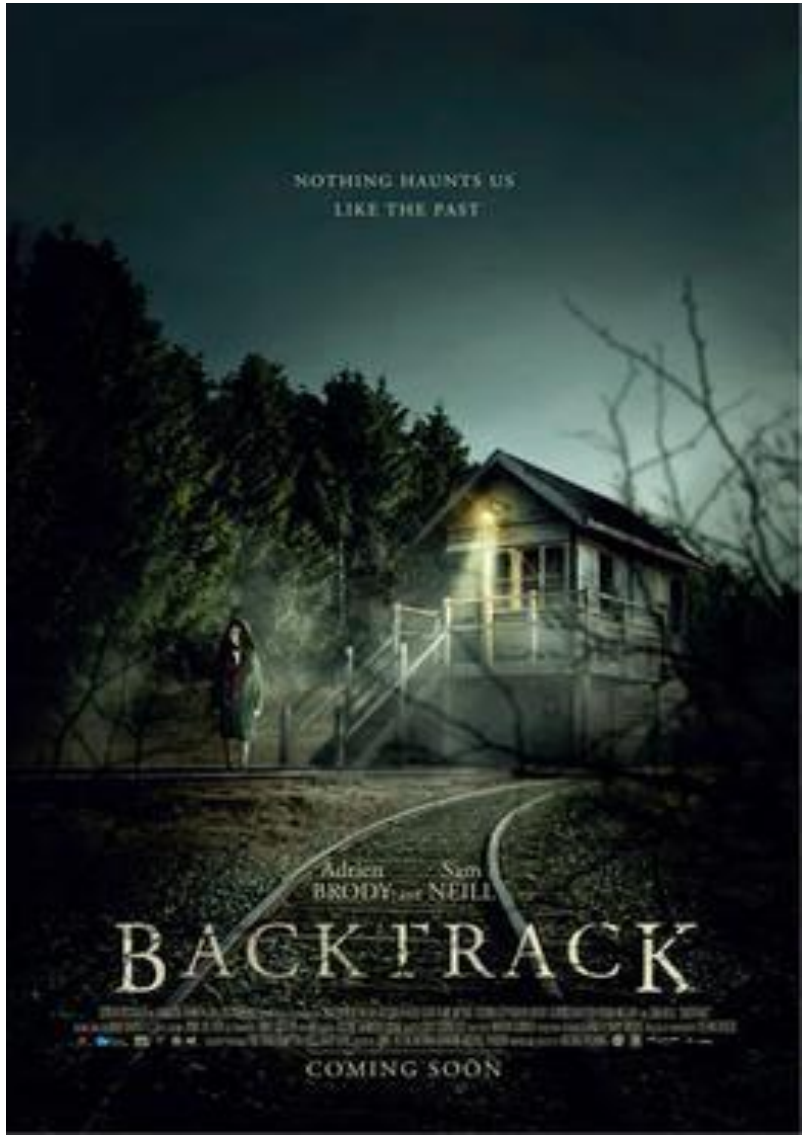


The poster for 2015 mystery-thriller "Backtrack." Critics gave it a 30% on Rotten Tomatoes, citing "not enough recursion."

YEAH A4

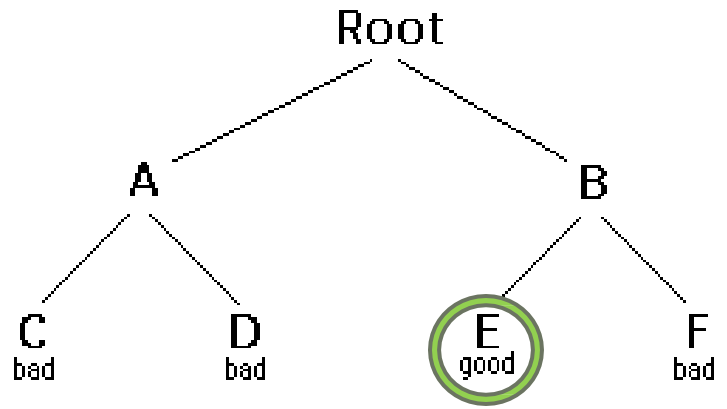
BACKTRACKING

What is Recursive Backtracking?

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

Difficult problems are problems where, at any point in time, 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 back to a place where you could go elsewhere.

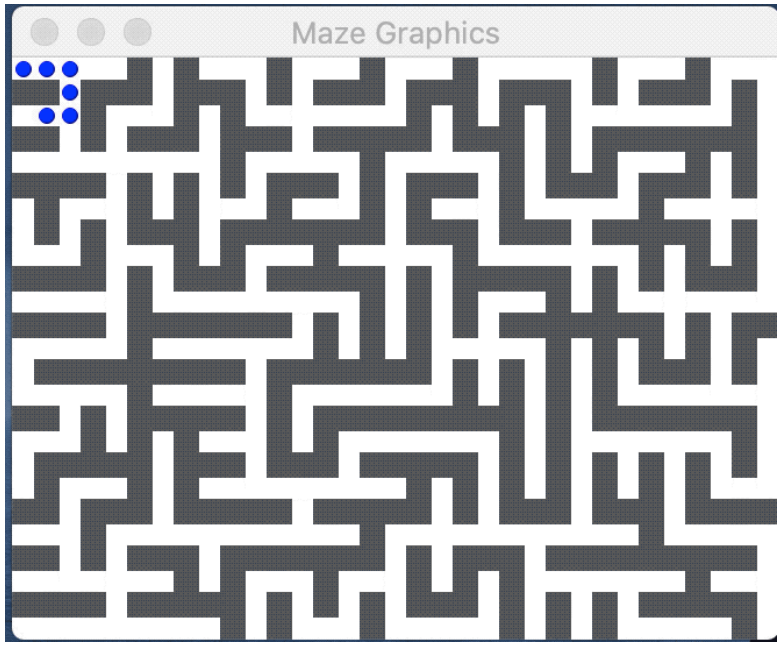
Backtracking frequently answers the question: *can this be done?* That's why many backtracking functions return bool



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

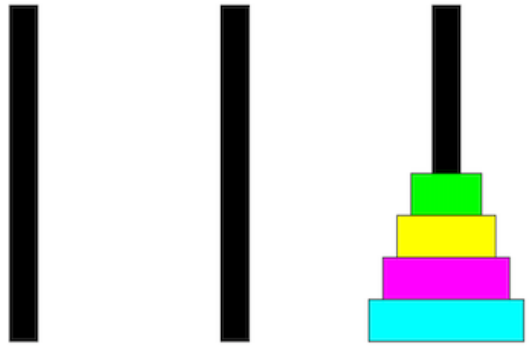
1. Assignment Logistics
2. Warmup
3. Text Predict (and why the smartphone changed everything)
4. Boggle Score (and why computers are cheaters)
5. Banzhaf Power Index (and why democracy is broken ☹)

But first! Some logistics

- This assignment is due **this Friday 10/16, at 11:59PDT.**
- As always, the deadline to submit is two days later, on **Sunday 10/18, at 11:59PDT**

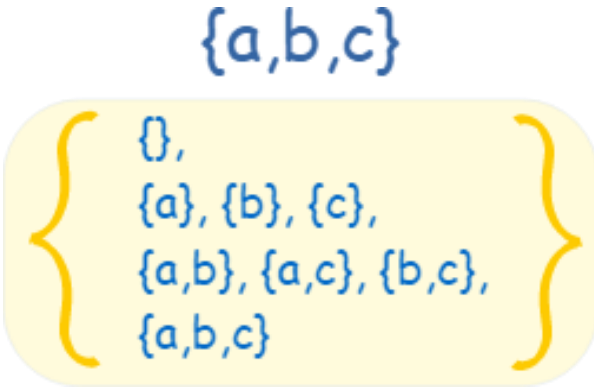
Part 1: Warmups

- You will be debugging **2** recursive functions!
- The first exercise will give you practice using the debugger's "step in/out/over" functionality on the **The Towers of Hanoi** problem.
- Step in/out/over are the three steps that you'll use to navigate programs in the debugger. It's really important that you know how to use them when debugging recursive backtracking functions!

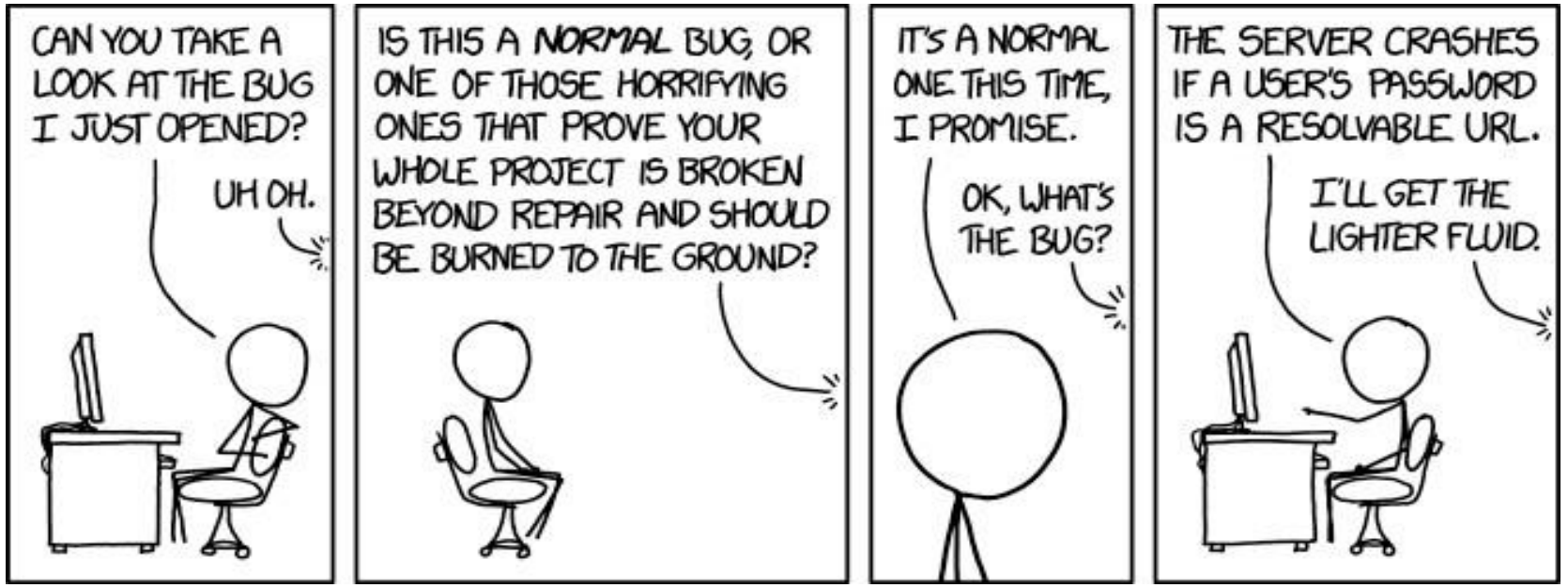


Part 1: Warmups

- Now that your debugging skills are up to par, it's time to put them into action!
- You'll be debugging a function that generates all possible subsets of a given set recursively!
  - Specifically, it returns a count of the number of subsets whose members sum to zero!
- Unfortunately, the function is buggy. It's a small bug, but it produces frustratingly wrong behavior. You'll need to find it and report about it in **shortanswer.txt**



Questions about the Warmups?



xkcd

Part 2: TextPredict

- Let's throw it back...

Part 2: TextPredict

- Let's throw it back...

Did any of you have one of these?

Part 2: TextPredict

- These were 10-digit keypads! Aaaaaand they sucked. They were really really bad because it was so hard to type on them.
- They were so bad that people made a bunch of **algorithms** to try and predict what you were typing before you were done. Auto-complete or mercy? You decide.
- You're going to implement one of these algorithms: the **Tegic T9 Algorithm!**

Part 2: TextPredict

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! (wait, why is this better than a HashSet<String> ?????? Ask an instructor!)
- 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.
- static const Map<int, Set<char>> keypad = {{2, {'a', 'b', 'c'}}, {3, {'d', 'e', 'f'}}, {4, {'g', 'h', 'i'}}, {5, {'j', 'k', 'l'}}, {6, {'m', 'n', 'o'}}, {7, {'p', 'q', 'r', 's'}},{8, {'t', 'u','v'}}, {9, {'w', 'x', 'y', 'z'}}};
- You can't modify this map!

Part 2: TextPredict

- Let's say I send a 1 word text to Nick 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!

Part 2: TextPredict

- Let's start with the first number, **9**.

Part 2: TextPredict

- Let's start with the first number, **9**.
- 9 maps to the chars {'w', 'x', 'y', 'z'}. These are the possible first letters of my message!

Part 2: TextPredict

- Let's start with the first number, **9**.
- 9 maps to the chars {'w', 'x', 'y', 'z'}. These are the possible first letters of my message!
- From there, we can consider 4 recursive sub-possibilities: a scenario in which each w, x, y, and z are the first letter of our string.

Part 2: TextPredict

- Let's start with the first number, **9**.
- 9 maps to the chars {'w', 'x', 'y', 'z'}. These are the possible first letters of my message!
- From there, we can consider 4 recursive sub-possibilities: a scenario in which each w, x, y, and z are the first letter of our string.
- We can then repeat the process on the next number, **3.** The possible letters we can get are {'d', 'e', 'f'}. For each of the 4 possibilities we have, each can have 3 recursive sub-possibilities, 'w' can have 'd', 'e', or 'f' as its second letter, 'x' can have 'd', 'e', or 'f' as its second letter, and so on with 'y' and 'z'.

Part 2: TextPredict

- Let's start with the first number, **9**.
- 9 maps to the chars {'w', 'x', 'y', 'z'}. These are the possible first letters of my message!
- From there, we can consider 4 recursive sub-possibilities: a scenario in which each w, x, y, and z are the first letter of our string.
- We can then repeat the process on the next number, **3.** The possible letters we can get are {'d', 'e', 'f'}. For each of the 4 possibilities we have, each can have 3 recursive sub-possibilities, 'w' can have 'd', 'e', or 'f' as its second letter, 'x' can have 'd', 'e', or 'f' as its second letter, and so on with 'y' and 'z'.
- You'll continue this recursive process until you have no more numbers left! The strings you have at the end are all possible words!

- What did I send to Nick? You'll have to use your algorithm to find out ;)

Part 2: TextPredict

- Some notes about TextPredict:

```
void predict(string digits, Set<string>& suggestions, Lexicon& lex)
{
    /* TODO: Implement this function. */
}
```

- In this function you'll be given a string of digits representing the original message. You'll also be given a lexicon to look up words, and you'll be given a Set<string> to store your suggestions when you find words.

Part 2: TextPredict

- Some notes about TextPredict:

```
void predict(string digits, Set<string>& suggestions, Lexicon& lex)
{
    /* TODO: Implement this function. */
}
```

- In this function you'll be given a string of digits representing the original message. You'll also be given a lexicon to look up words, and you'll be given a Set<string> to store your suggestions when you find words.
- Let's say you're a few steps into the recursion, and one of your words is "xjk". Should you keep recursing on this string, or is this prefix a dead end? You should use the **containsPrefix()** function in the lexicon to prune your decision tree.

Part 2: TextPredict

- Some notes about TextPredict:

```
void predict(string digits, Set<string>& suggestions, Lexicon& lex)
{
    /* TODO: Implement this function. */
}
```

- In this function you'll be given a string of digits representing the original message. You'll also be given a lexicon to look up words, and you'll be given a Set<string> to store your suggestions when you find words.
- Let's say you're a few steps into the recursion, and one of your words is "xjk". Should you keep recursing on this string, or is this prefix a dead end? You should use the **containsPrefix()** function in the lexicon to prune your decision tree.
- This should go without saying, but **don't add things to SUGGESTIONS that aren't valid words**. Use the lexicon to verify if your results are valid words

Part 2: TextPredict

- Some more notes about TextPredict:

```
void predict(string digits, Set<string>& suggestions, Lexicon& lex)
{
    /* TODO: Implement this function. */
}
```

- Feel free to write a helper function to do the recursion here – do you need more parameters that what you're given **(please don't change the original function header)**

Part 2: TextPredict

- Some more notes about TextPredict:

```
void predict(string digits, Set<string>& suggestions, Lexicon& lex)
{
    /* TODO: Implement this function. */
}
```

- Feel free to write a helper function to do the recursion here – do you need more parameters that what you're given **(please don't change the original function header)**
- Before you code this problem, you should have a *very* clear idea of what your choose / explore / and unchoose steps are.

Part 2: TextPredict

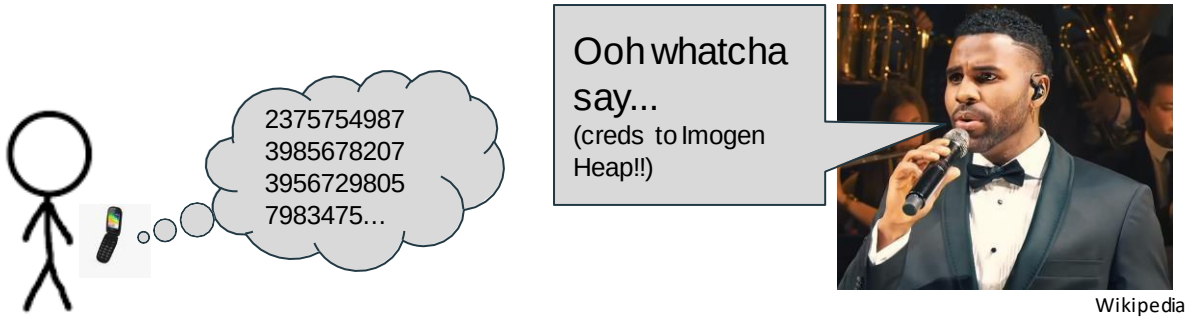
- Some more notes about TextPredict:

```
void predict(string digits, Set<string>& suggestions, Lexicon& lex)
{
    /* TODO: Implement this function. */
}
```

- Feel free to write a helper function to do the recursion here – do you need more parameters that what you're given **(please don't change the original function header)**
- Before you code this problem, you should have a *very* clear idea of what your choose / explore / and unchoose steps are.
- You're given a string of digits, and we guarantee that it'll only be numbers 2-9 (no funny business here!) To access into the map, you'll probably have to convert characters to actual **integers.** The **charToInteger(char c)** function in strlib.h can do this for you :)

Charole Baskin is watching!

Questions about TextPredict?



Wikipedia

.json Derulo, CS106B alum and predictive algorithm fiend

Part 3: Boggle ComputerSearch

Who played Boggle?

Part 3: Boggle ComputerSearch

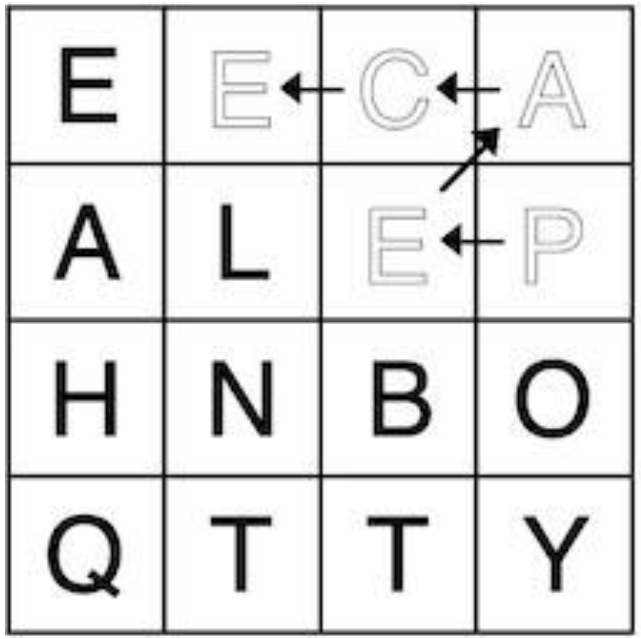
- In this part, you'll be asked to write the functionality for a **computerized Boggle player,** who uses a **Lexicon** and recursion to find every possible word on the Boggle board!
- More specifically, you'll be implementing the following function:

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

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

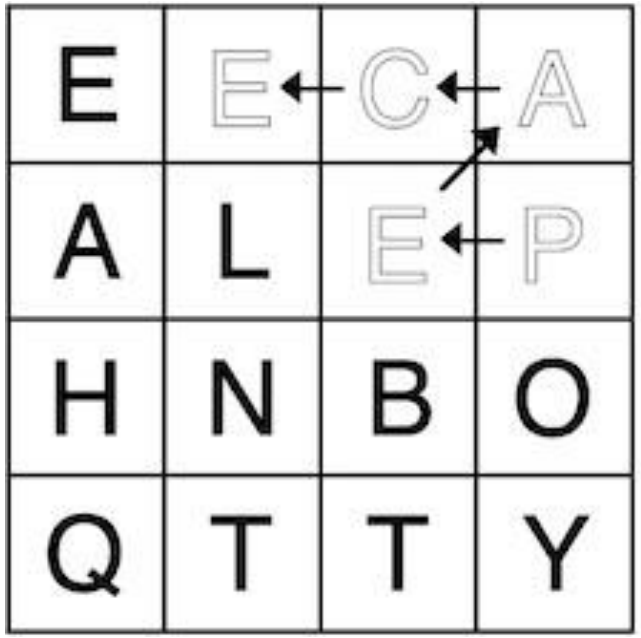
Part 3: Boggle ComputerSearch

- 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.



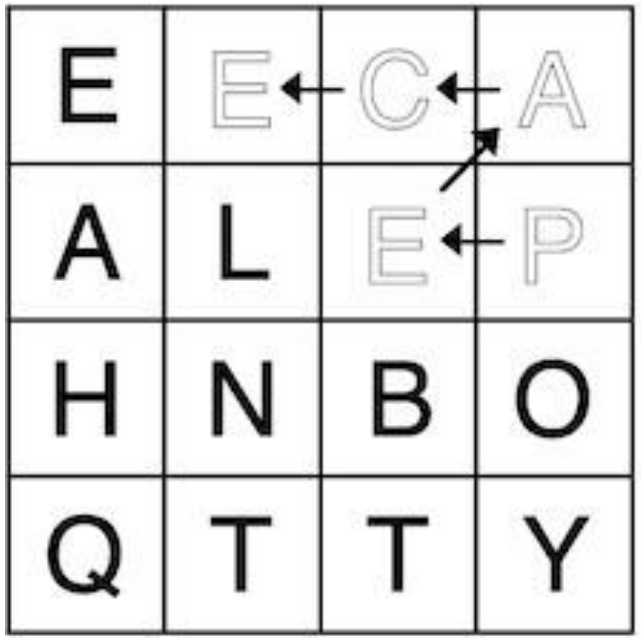
Part 3: Boggle ComputerSearch

- 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.
- Let's think about the routine you'd need to do for each letter on the board. Consider that we're trying to find all words from the starting letter **'P'**.



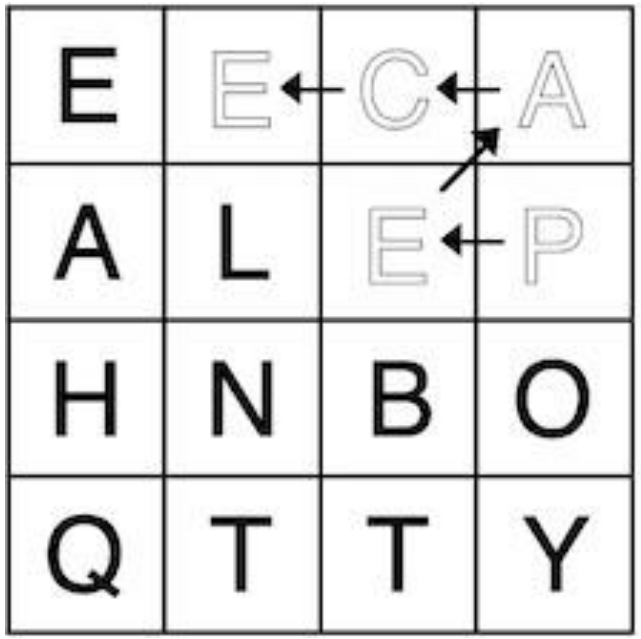
Part 3: Boggle ComputerSearch

- 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.
- Let's think about the routine you'd need to do for each letter on the board. Consider that we're trying to find all words from the starting letter **'P'**.
  - From there, my I should look at all of the letters around me **that I haven't already visited in my current word**. At the beginning, this will be false because my word is just **'P'**.



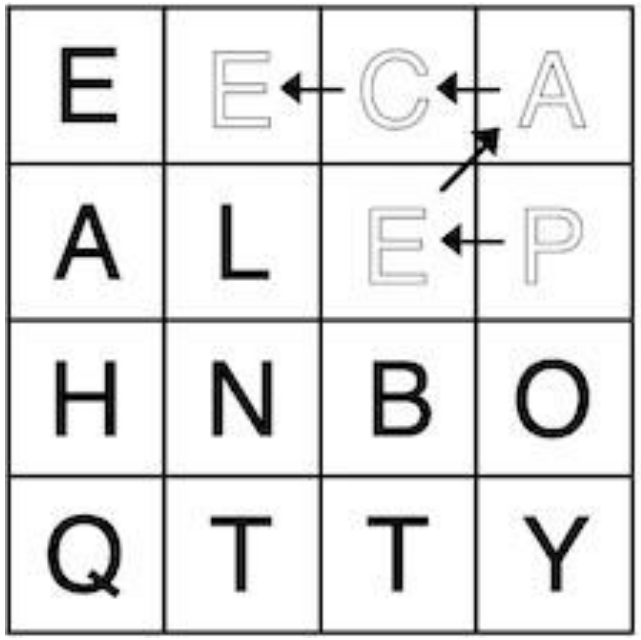
Part 3: Boggle ComputerSearch

- Let's think about the routine you'd need to do for each letter on the board. Consider that we're trying to find all words from the starting letter **'P'**.
  - From there, my I should look at all of the letters around me **that I haven't already visited in my current word**. At the beginning, this will be false because my word is just **'P'**.
  - Consider all of your neighboring letters. One by one, append them to your current string, and repeat the exploration process.
    - When you append one neighboring letter, you need to remove it before you append the next! **Verify to yourself that this must be true!**



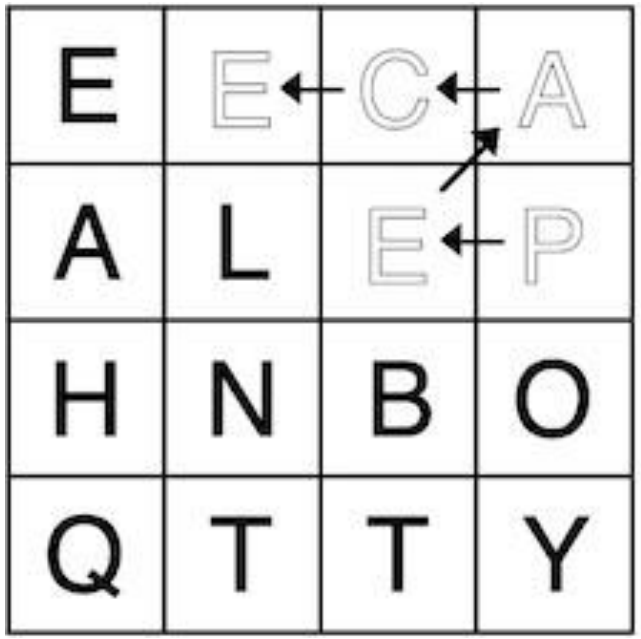
Part 3: Boggle ComputerSearch

- Let's think about the routine you'd need to do for each letter on the board. Consider that we're trying to find all words from the starting letter **'P'**.
  - Consider all of your neighboring letters. One by one, append them to your current string, and repeat the exploration process.
    - When you append one neighboring letter, you need to remove it before you append the next! **Verify to yourself that this must be true!**
  - Think about your base cases here. When do you know that you've succeeded in finding a word? What should you do then? Are there cases in which you should **stop** exploring a certain string?



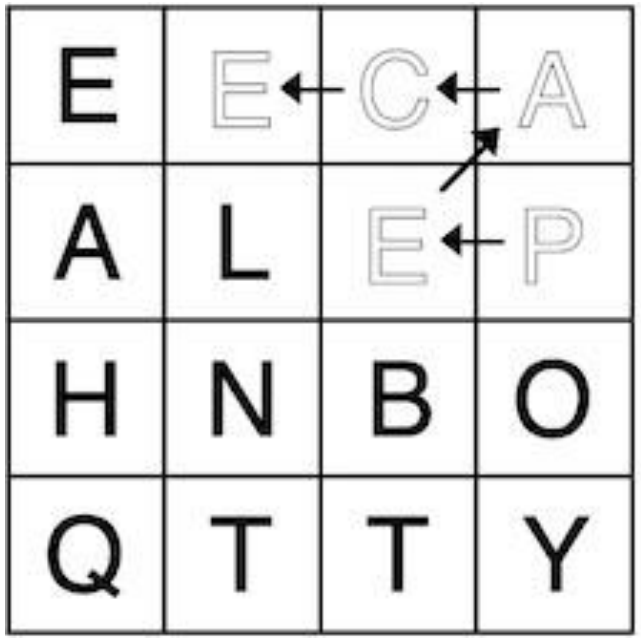
Part 3: Boggle ComputerSearch

- Once again, ensure 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!
  - Why would considering 'p' be a problem?



Part 3: Boggle ComputerSearch

- When you find a word, you'll need to compute the score for it. Words shorter than 4 characters long cannot be scored. From there, the [length : score] relationship looks like this: [4:1, 5:2, 6:3, 7:4....]



Part 3: Boggle ComputerSearch

A few more notes:

- We've only given you this one function. 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 ComputerSearch

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?

What should we call this new word game?

Dog: how about Doggle?

Bog: I have a better idea

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 votes, we're going to look at voting **bocks,** 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

An example list of voting blocks

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

There are 5 critical votes total!

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 N 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 a hypothetical 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 the steps 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.

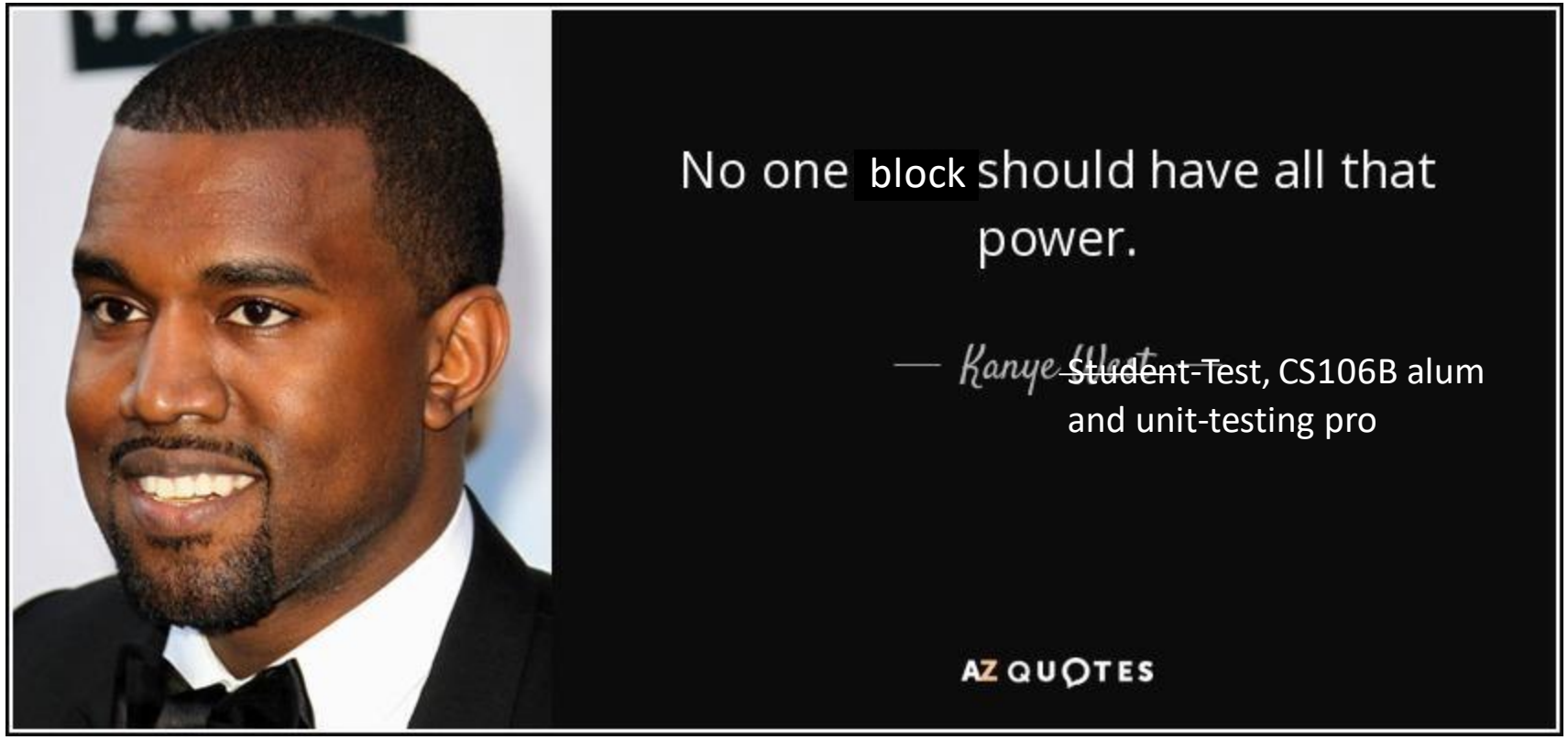
1. Take your block out of the mix **before** you do anything (you could leave yourself in, but remember you want to generate these coalitions **without** your votes).
2. From there, you can begin recursively generating subsets from the remaining blocks. You've seen recursive subsets code before – the warmup has an example too! The kicker here is that **instead of literally generating subsets of blocks, simply store the number of votes a block would contribute (that should be your recursive choice that you 'include' or 'exclude')**
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 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.
- Although it is possible to complete this assignment by literally generating the voting subsets (a Vector<Vector<int>>) and **then** start computing critical votes, this approach is **super slow** and uses a lot of computer memory. You should try and complete this part without generating literal subsets.
- 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?



~~Student~~-Test, CS106B alum and unit-testing pro