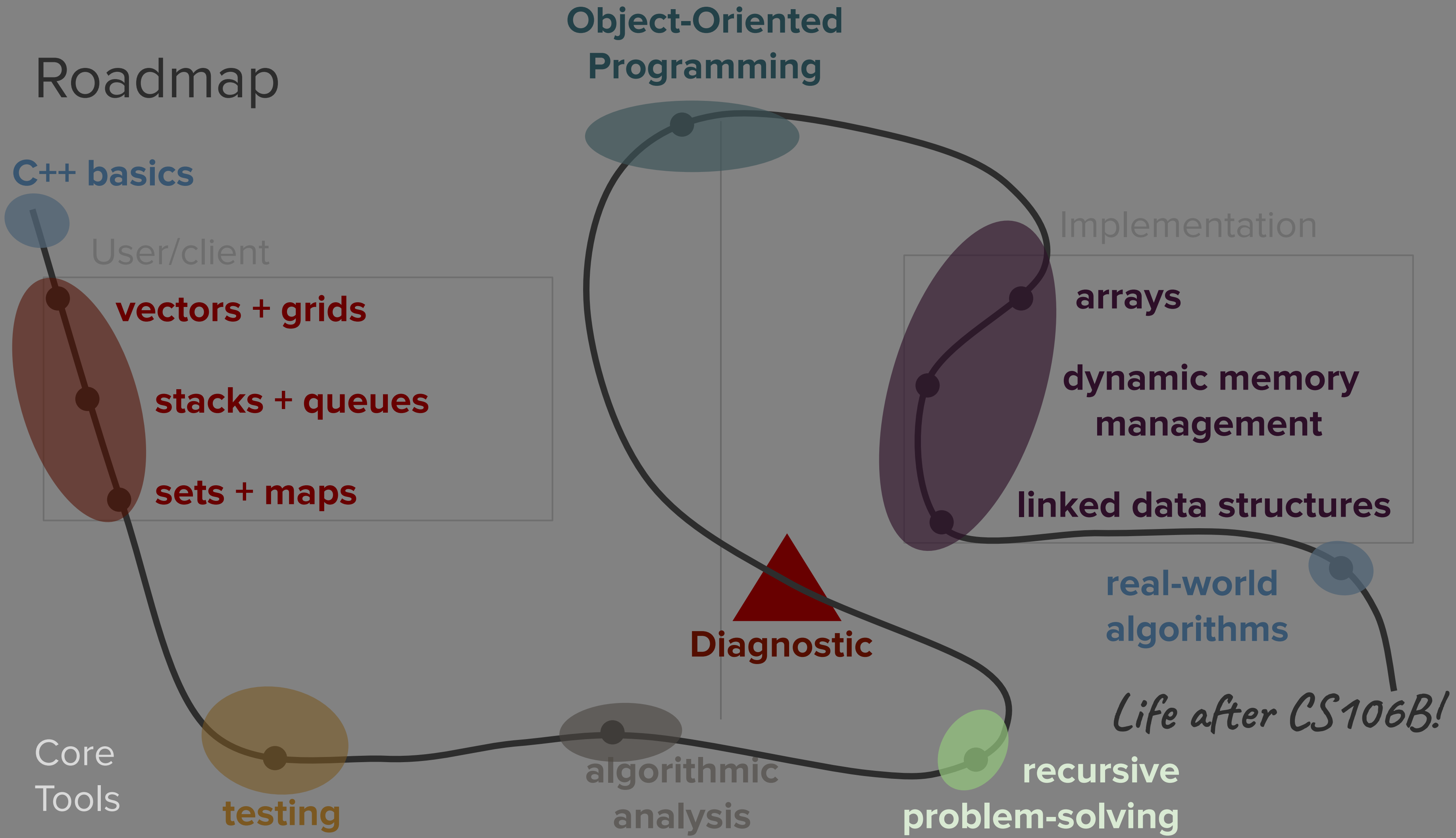
Why We Use Recursion

Which do you prefer: iteration or recursion?
Include a short phrase explaining why.
(put your answers the chat)

Roadmap


Object-Oriented Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

Today's question

Why is recursion such a powerful problem-solving tool?

Today's topics

1. Review
2. Elegance
3. Efficiency
   (the return of Big O)
4. Recursive Backtracking

Review

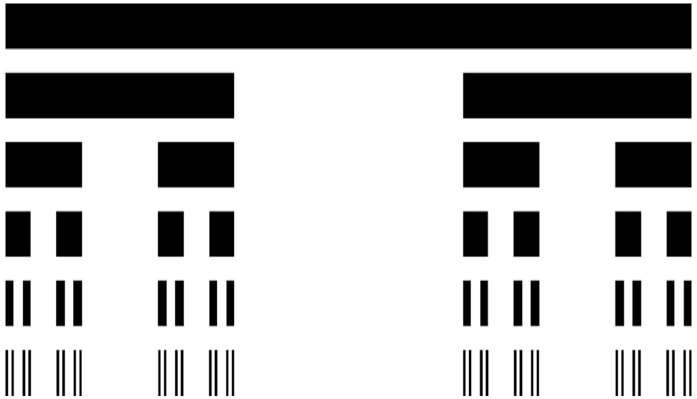
multiple base cases
and multiple recursive cases

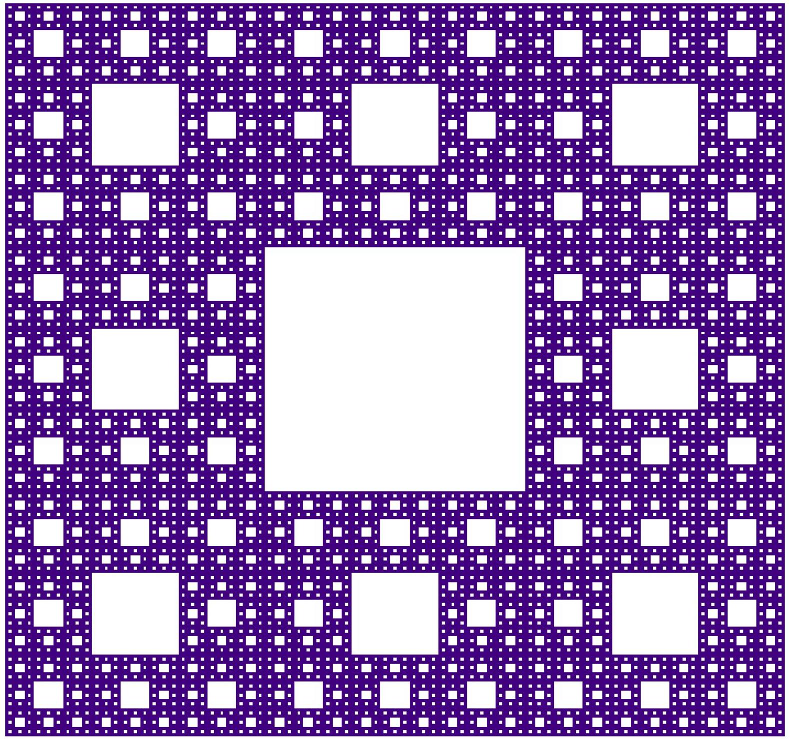
isPalindrome()

- **Base cases**:
  - isPalindrome(“”) ➜ **`true`**
  - isPalindrome(string of length 1) ➜ **`true`**
  - If the first and last letters are not equal ➜ **`false`**

There can be multiple base (or recursive) cases!

- **Recursive case:** If the first and last letters are equal, isPalindrome(string) = isPalindrome(string minus first and last letters)





Sierpinski Carpet

```
void drawSierpinskiCarpet(GWindow& window, double x, double y, double size, int order) {
   // Base case: A carpet of order 0 is a filled square.
   if (order == 0) {
       drawSquare(window, x, y, size);
   }  else {
       for (int row = 0; row < 3; row++) {
           for (int col = 0; col < 3; col++) {
               // The only square to skip is the very center one.
               if (row != 1 || col != 1) {
                   double newX = x + col * size / 3;
                   double newY = y + row * size / 3;
                   drawSierpinskiCarpet(window, newX, newY, size / 3, order - 1);
               }
           }
       }
   }
}
```

Iteration + Recursion

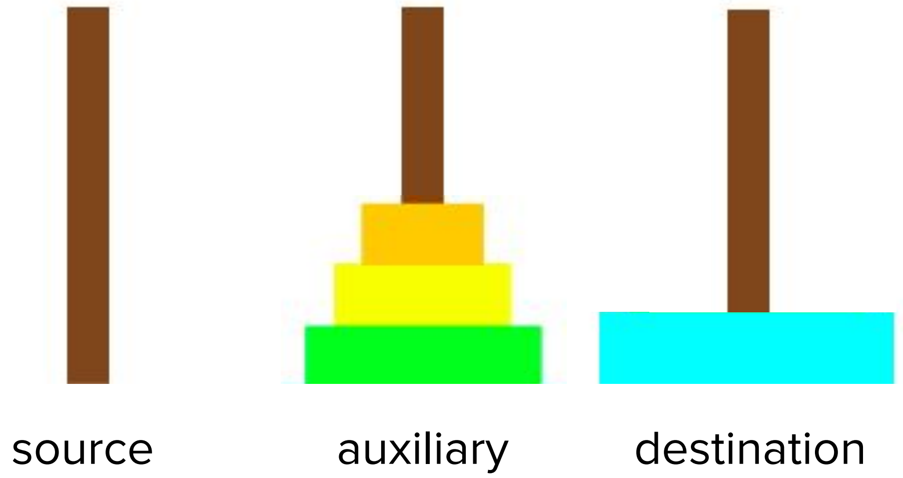
- It’s completely reasonable to mix iteration and recursion in the same function.
- Here, we’re firing off eight recursive calls, and the easiest way to do that is with a double for loop.
- Recursion doesn’t mean “the absence of iteration.” It just means “solving a problem by solving smaller copies of that same problem.”
- Iteration and recursion can be very powerful in combination!

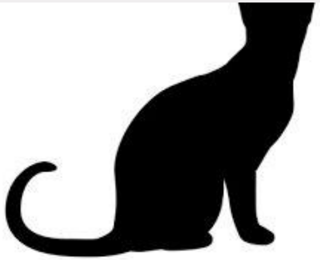
Why do we use recursion?

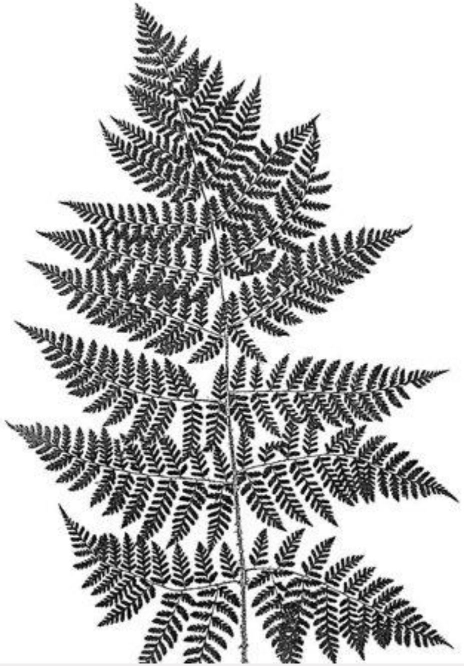
Towers of Hanoi with 4 disks

- We want to first move the biggest disk over to the destination peg.
- Now we need to move the stack of three from auxiliary to destination.

Use our existing 3-disk algorithm!







Why do we use recursion?

- Elegance
  - Allows us to solve problems with very clean and concise code

- Efficiency
  - Allows us to accomplish better runtimes when solving problems

- Dynamic
  - Allows us to solve problems that are hard to solve iteratively

An **efficient** example: Binary Search

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

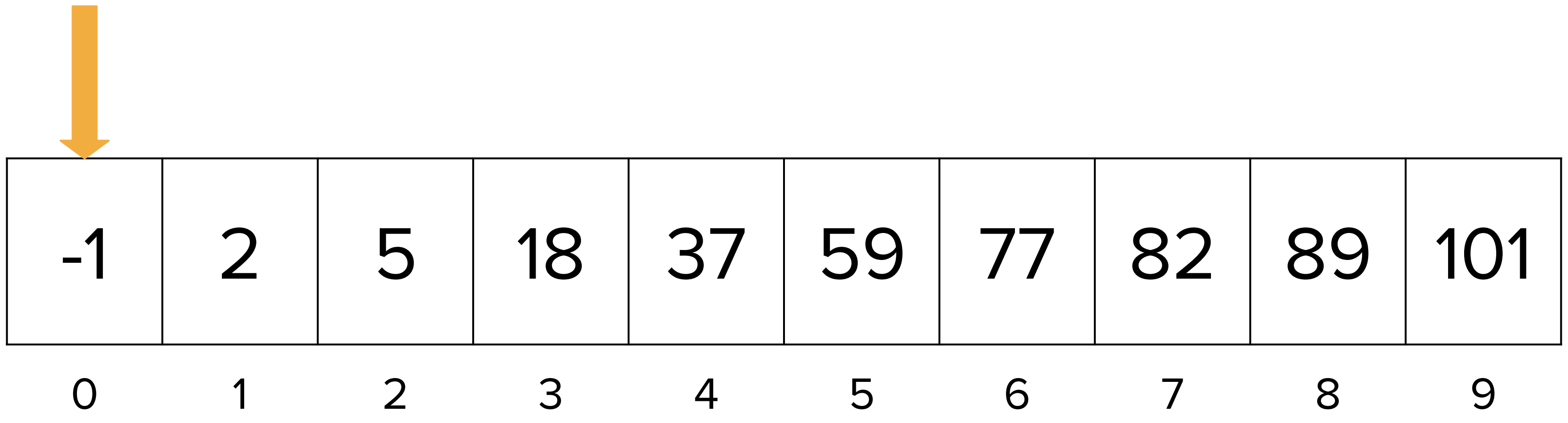
Where is 89?

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

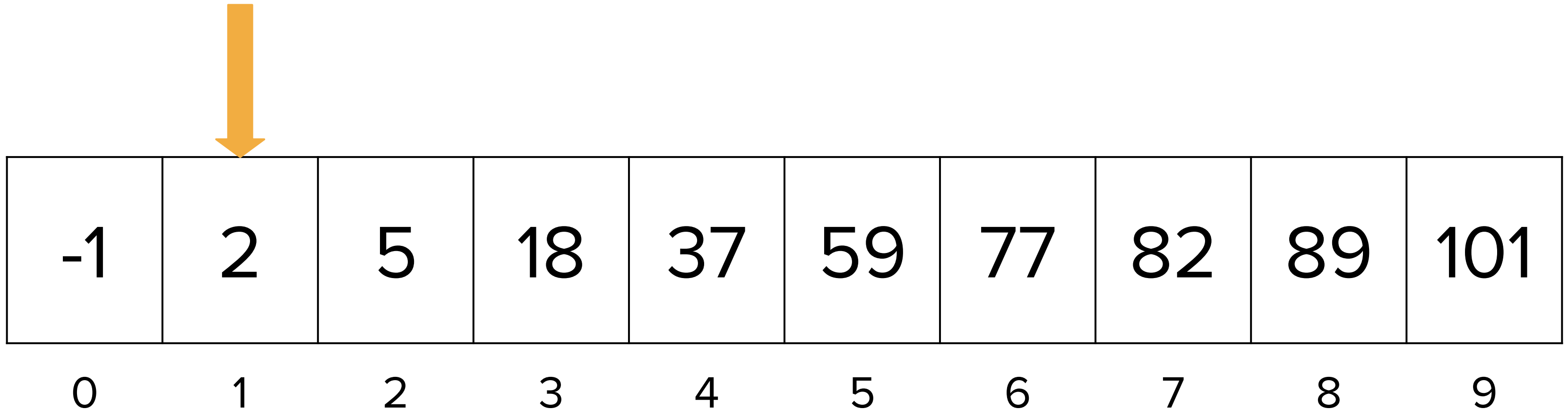
Idea #1: We could just go through each element in order and do a linear search.

Finding a number in a sorted list



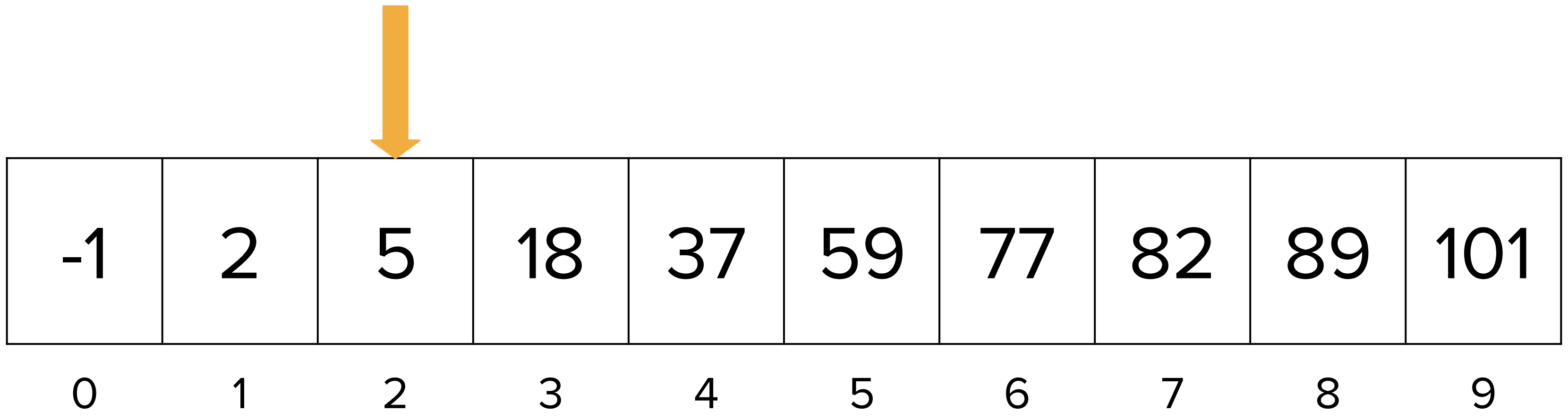
We could just go through each element in order and do a linear search.

Finding a number in a sorted list



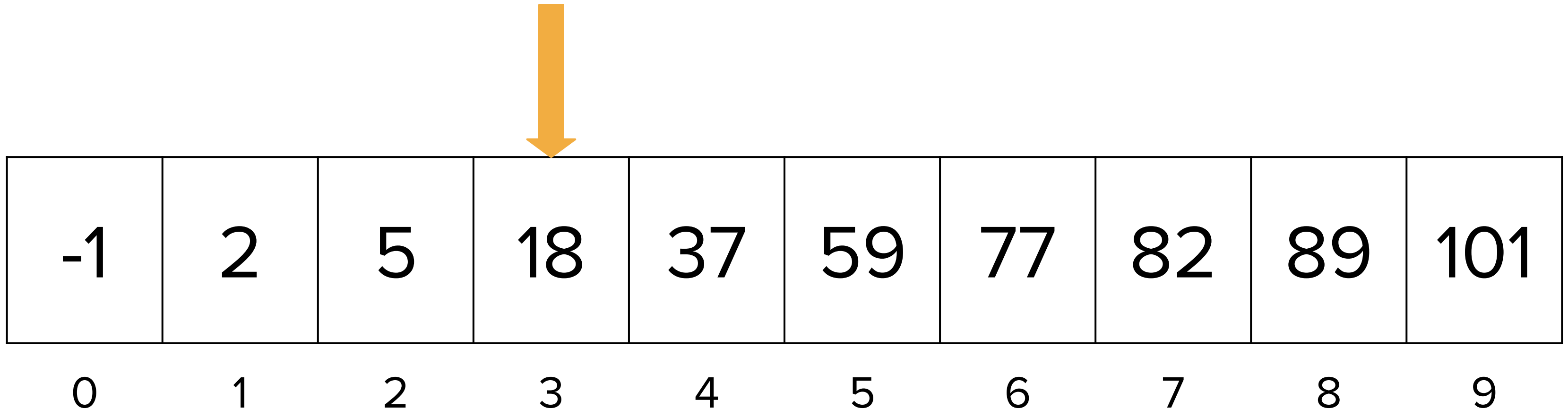
We could just go through each element in order and do a linear search.

Finding a number in a sorted list



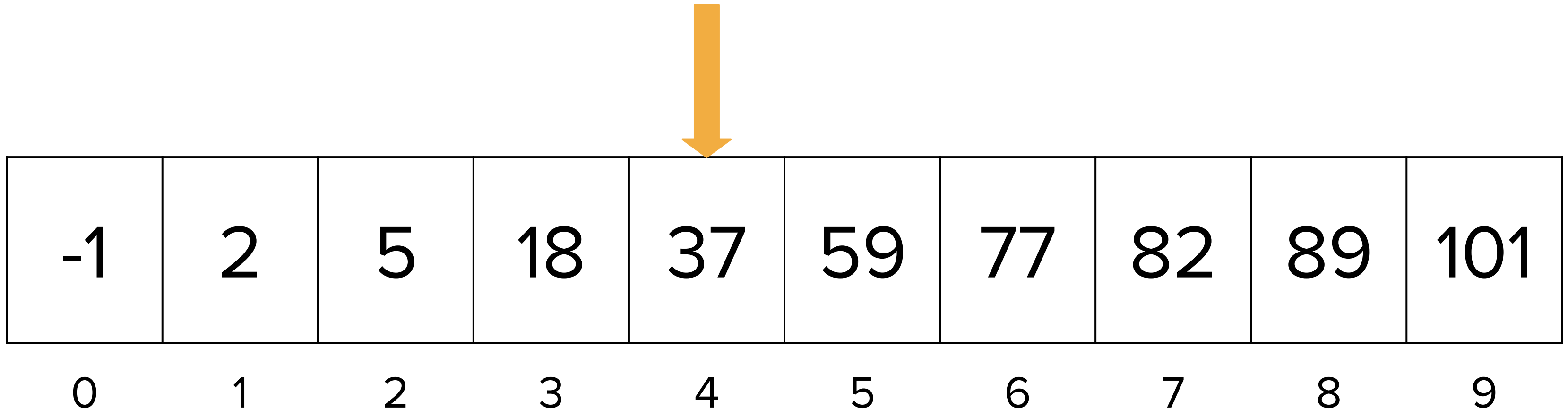
We could just go through each element in order and do a linear search.

Finding a number in a sorted list



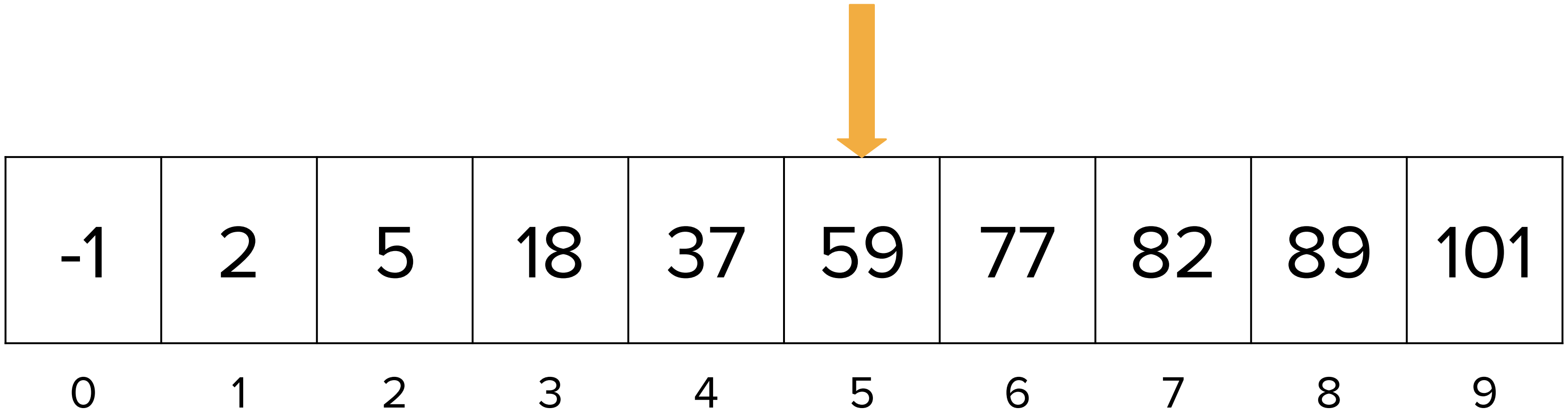
We could just go through each element in order and do a linear search.

Finding a number in a sorted list



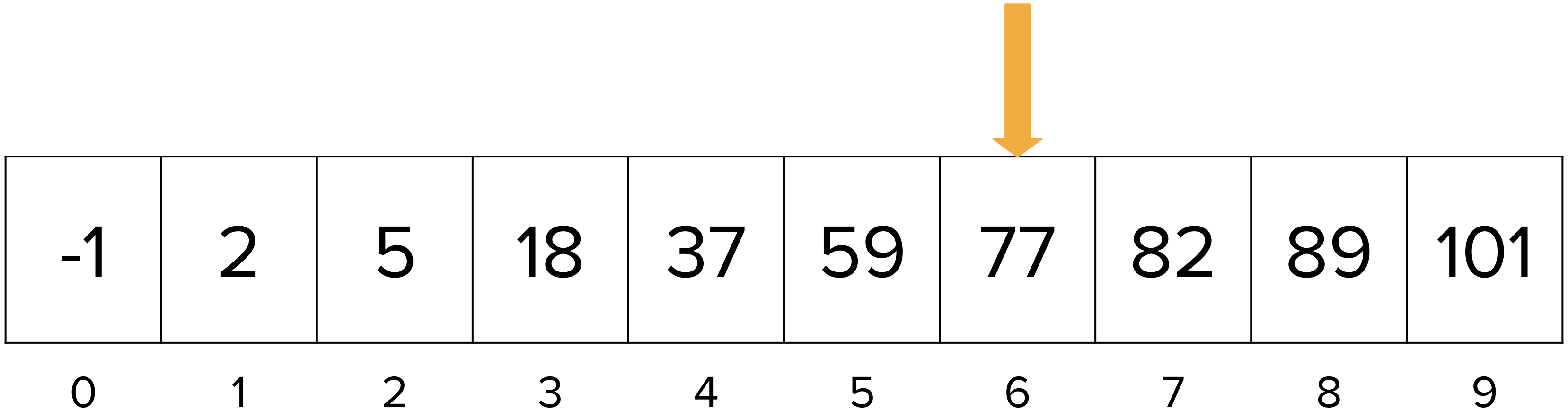
We could just go through each element in order and do a linear search.

Finding a number in a sorted list



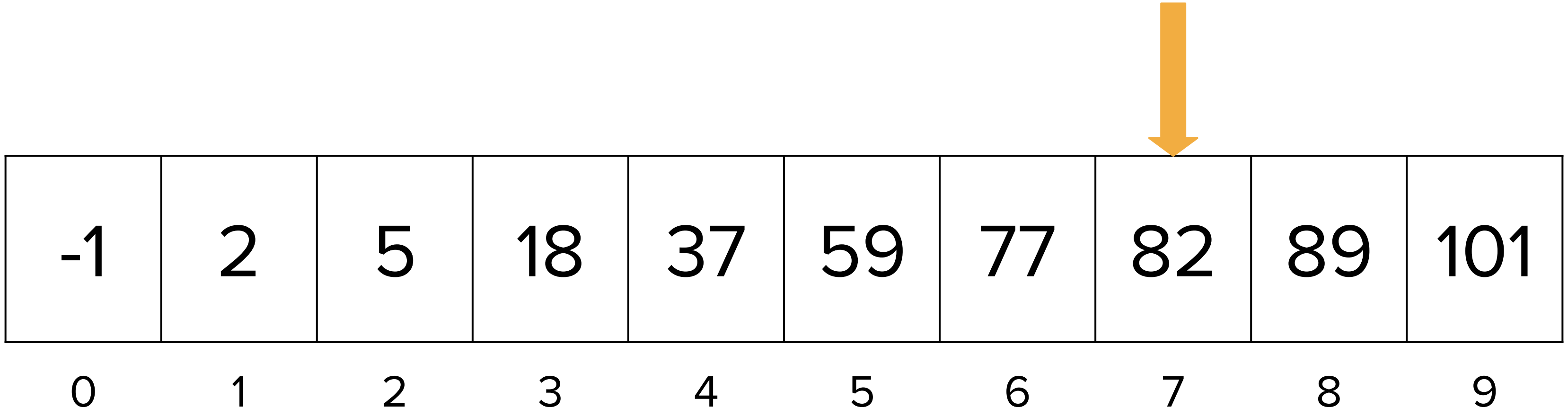
We could just go through each element in order and do a linear search.

Finding a number in a sorted list



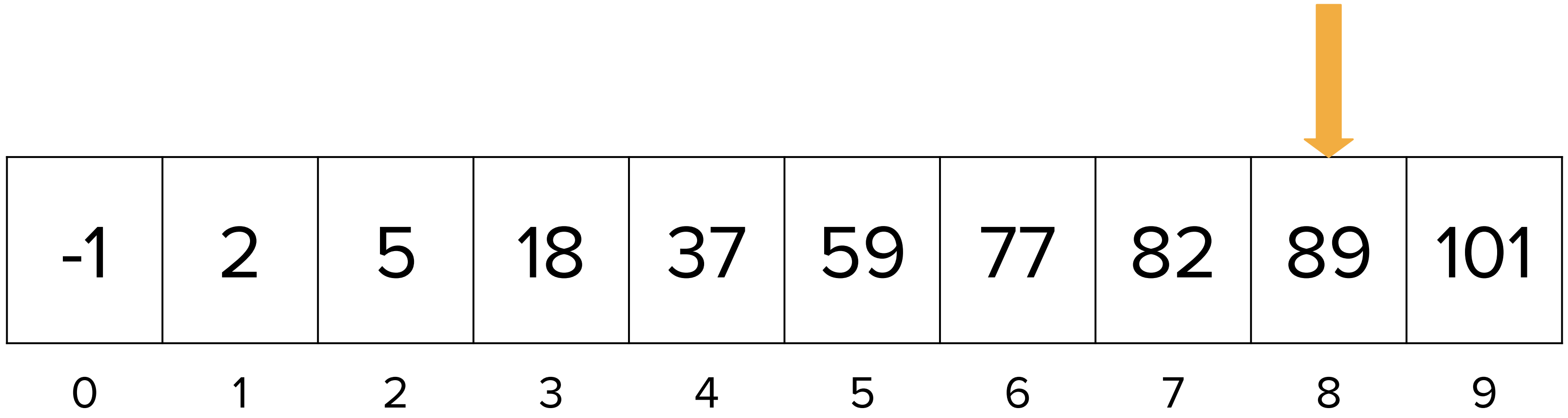
We could just go through each element in order and do a linear search.

Finding a number in a sorted list



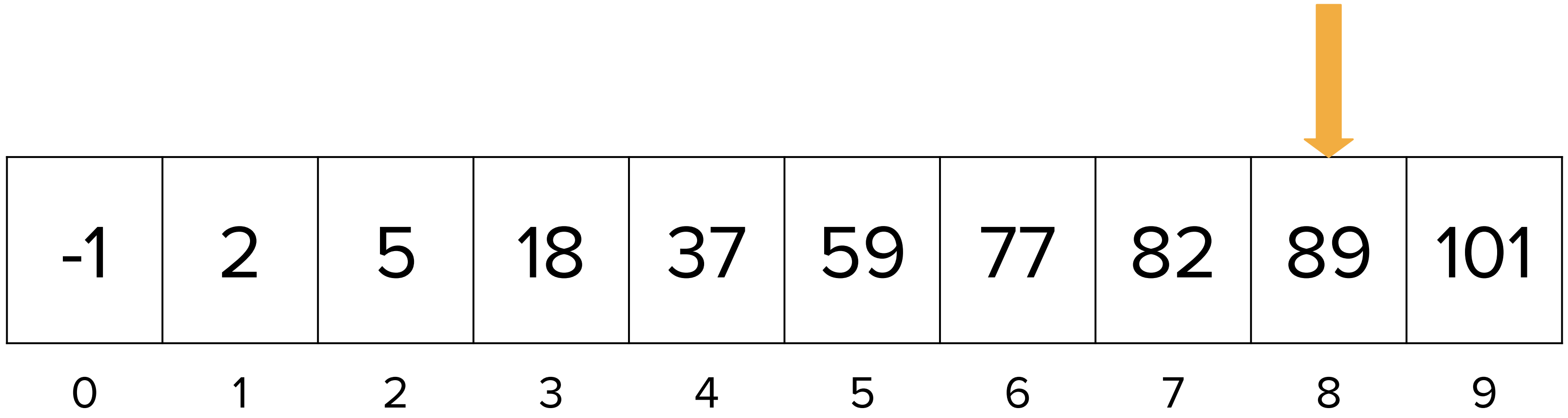
We could just go through each element in order and do a linear search.

Finding a number in a sorted list



We could just go through each element in order and do a linear search.

Finding a number in a sorted list



Linear search is **O(n)**

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Can we do better? Can we take advantage of the structure of the data?

ADT Big-O Matrix

- Vectors
  - `.size() - O(1)`
  - `.add() - O(1)`
  - `v[i] - O(1)`
  - `.insert() - O(n)`
  - `.remove() - O(n)`
  - `.clear() - O(n)`
  - `traversal - O(n)`
- Grids
  - `.numRows()/.numCols() - O(1)`
  - `g[i][j] - O(1)`
  - `.inBounds() - O(1)`
  - `traversal` - O(n^2)
- Queues
  - `.size() - O(1)`
  - `.peek() - O(1)`
  - `.enqueue() - O(1)`
  - `.dequeue() - O(1)`
  - `.isEmpty() - O(1)`
  - `traversal - O(n)`
- Stacks
  - `.size() - O(1)`
  - `.peek() - O(1)`
  - `.push() - O(1)`
  - `.pop() - O(1)`
  - `.isEmpty() - O(1)`
  - `traversal - O(n)`
- Sets
  - `.size() - O(1)`
  - `.isEmpty() - O(1)`
  - `.add() - ???`
  - `.remove() - ???`
  - `.contains() - ???`
  - `traversal - O(n)`
- Maps
  - `.size() - O(1)`
  - `.isEmpty() - O(1)`
  - `m[key] - ???`
  - `.contains() - ???`
  - `traversal - O(n)`

ADT Big-O Matrix

- Vectors
  - `.size() – O(1)`
  - `.add() – O(1)`
  - `v[i] – O(1)`
  - `.insert() – O(n)`
  - `.remove() – O(n)`
  - `.clear() – O(n)`
  - `traversal – O(n)`
- Grids
  - `.numRows()/.numCols() – O(1)`
  - `g[i][j] – O(1)`
  - `.inBounds() – O(1)`
  - traversal – $O(n^2)$
- Queues
  - `.size() – O(1)`
  - `.peek() – O(1)`
  - `.enqueue() – O(1)`
  - `.dequeue() – O(1)`
  - `.isEmpty() – O(1)`
  - `traversal – O(n)`
- Stacks
  - `.size() – O(1)`
  - `.peek() – O(1)`
  - `.push() – O(1)`
  - `.pop() – O(1)`
  - `.isEmpty() – O(1)`
  - `traversal – O(n)`
- Sets
  - `.size() – O(1)`
  - `.isEmpty() – O(1)`
  - `.add() – ???`
  - `.remove() – ???`
  - `.contains() – ???`
  - `traversal – O(n)`
- Maps
  - `.size() – O(1)`
  - `.isEmpty() – O(1)`
  - `m[key] – ???`
  - `.contains() – ???`
  - `traversal – O(n)`

Note: Sets and Maps don’t actually use a sorted list to store information, but the general idea of searching sorted data is similar.

Remember how their elements/keys always printed out in alphabetical order?

Note: Sets and Maps don't actually use a sorted list to store information, but the general idea of searching sorted data is similar.

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Where is 89?

Idea #2: Binary search

- Eliminate half of the data at each step.

- **Algorithm**: Check the middle element at **`(startIndex + endIndex) / 2`**
  - If the middle element is bigger than your desired value, eliminate the right half of the data and repeat.
  - If the middle element is smaller than your desired value, eliminate the left half of the data and repeat.
  - Otherwise, you’ve found your element!

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Where is 89?

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Start by looking at index:

```
(startIndex + endIndex) / 2
```

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

Start by looking at index:

`(0 + 9) / 2`

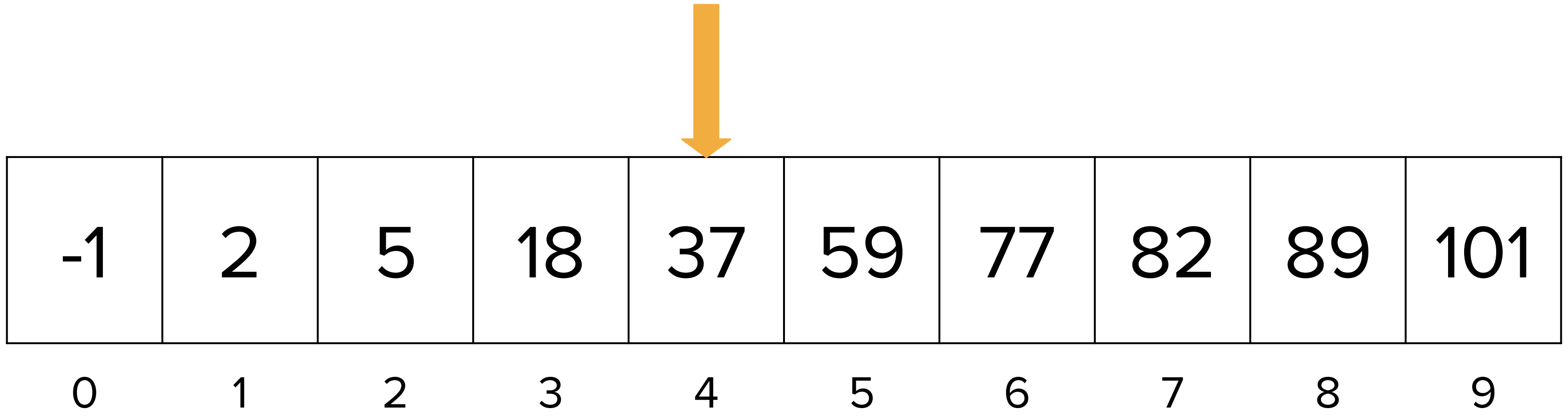
Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

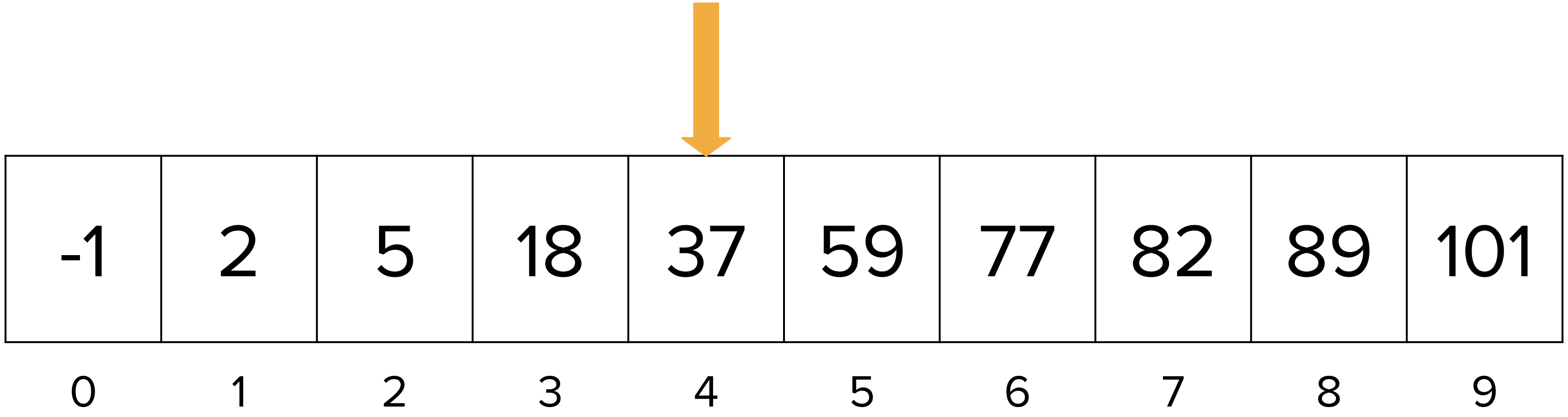
Start by looking at index:

4

Finding a number in a sorted list



Finding a number in a sorted list



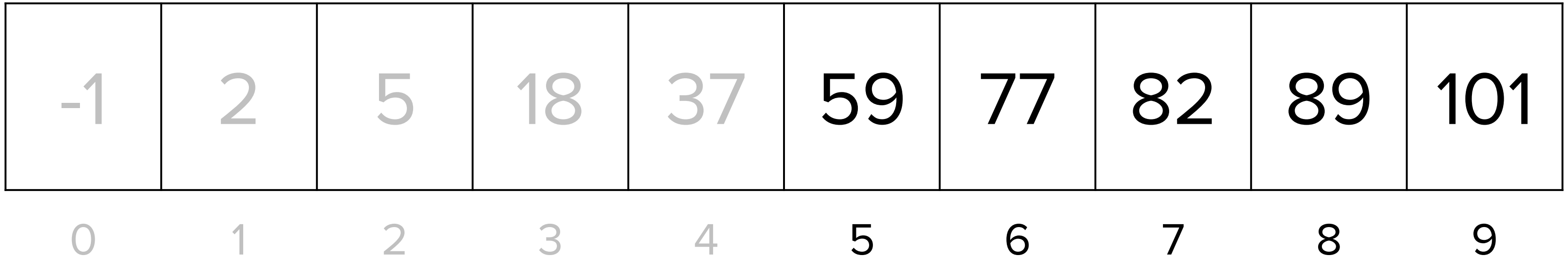
Too small

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

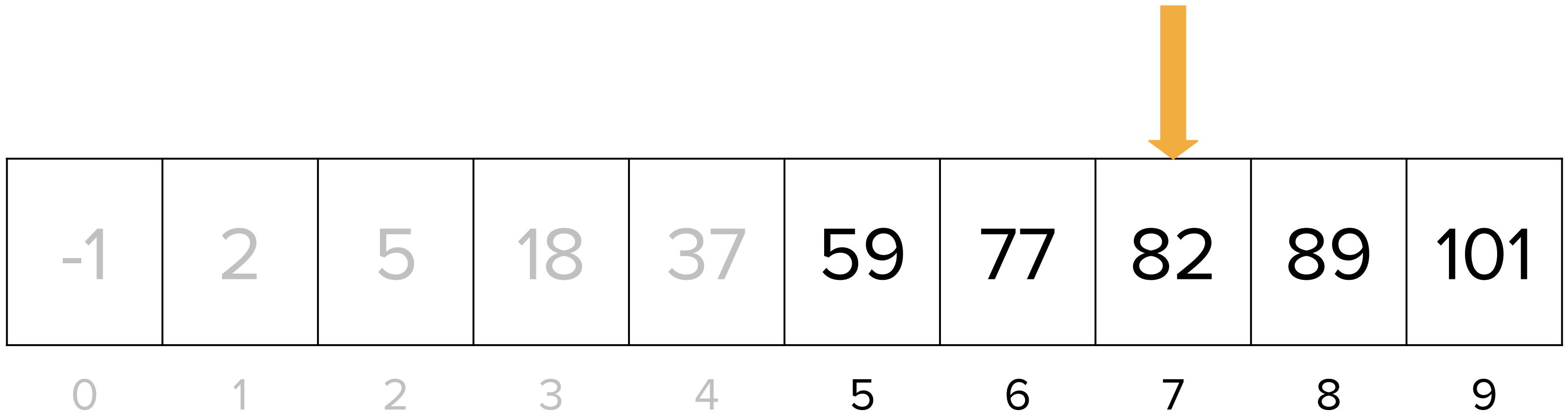
Eliminate left half

Finding a number in a sorted list



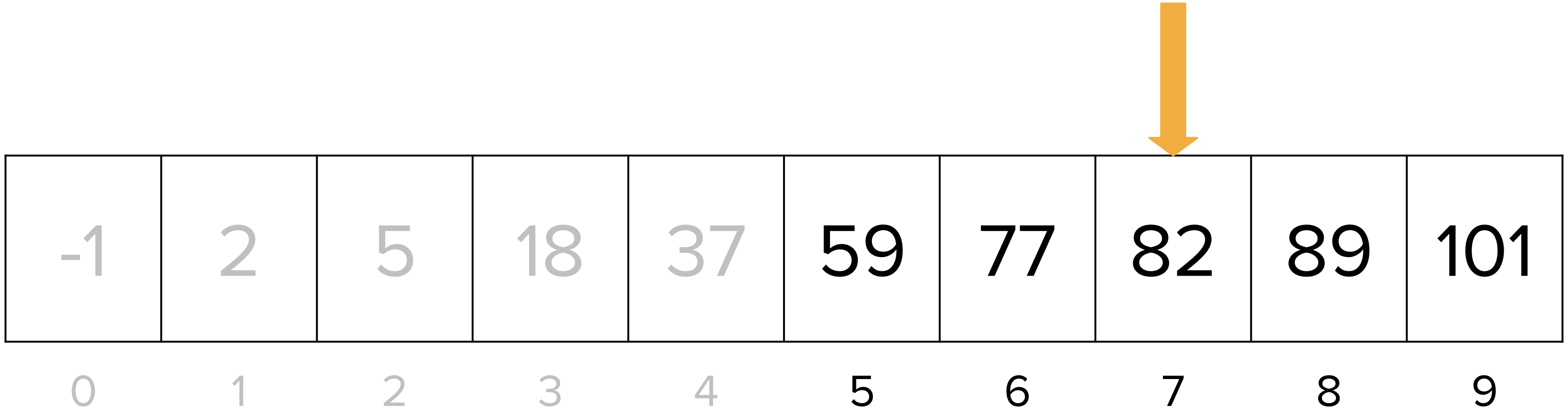
```
(startIndex + endIndex) / 2 =
(5 + 9) / 2 =
7
```

Finding a number in a sorted list



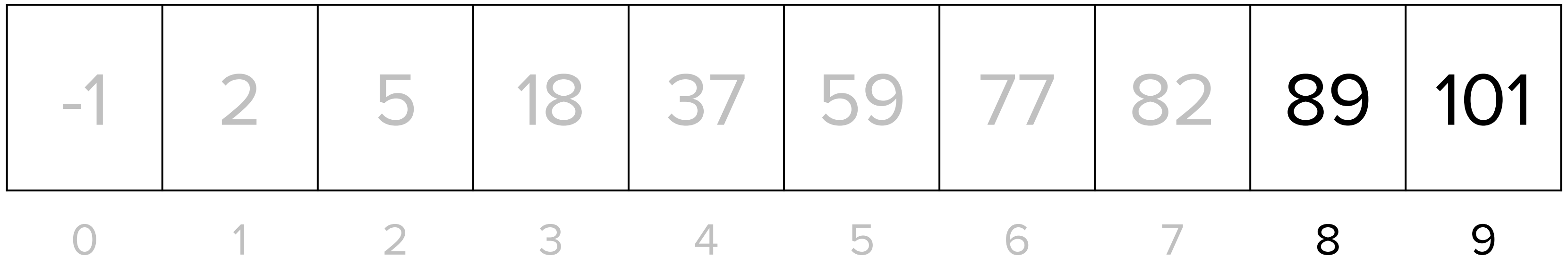
```
(startIndex + endIndex) / 2 =
(5 + 9) / 2 =
7
```

Finding a number in a sorted list



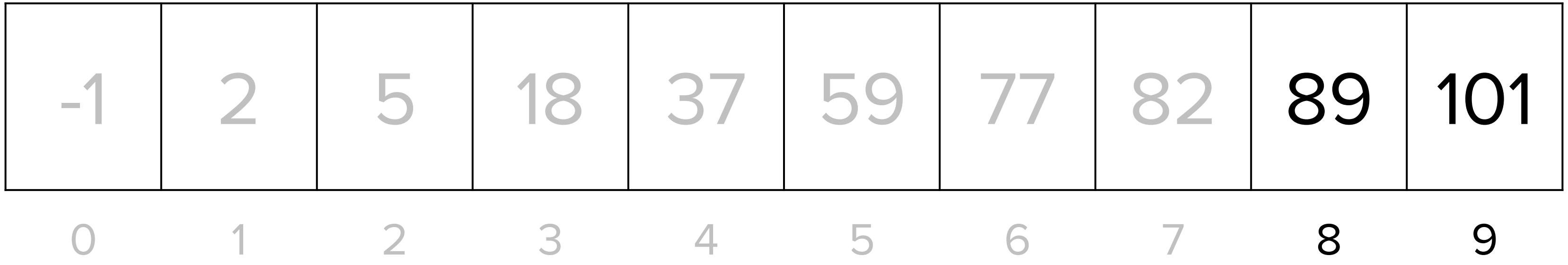
Too small

Finding a number in a sorted list



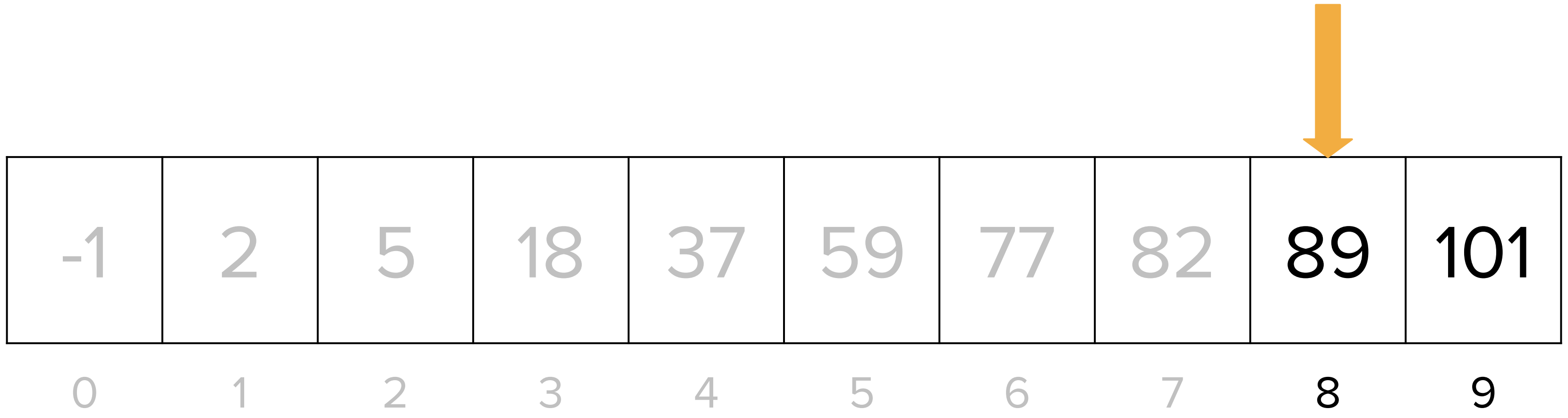
Eliminate left half

Finding a number in a sorted list



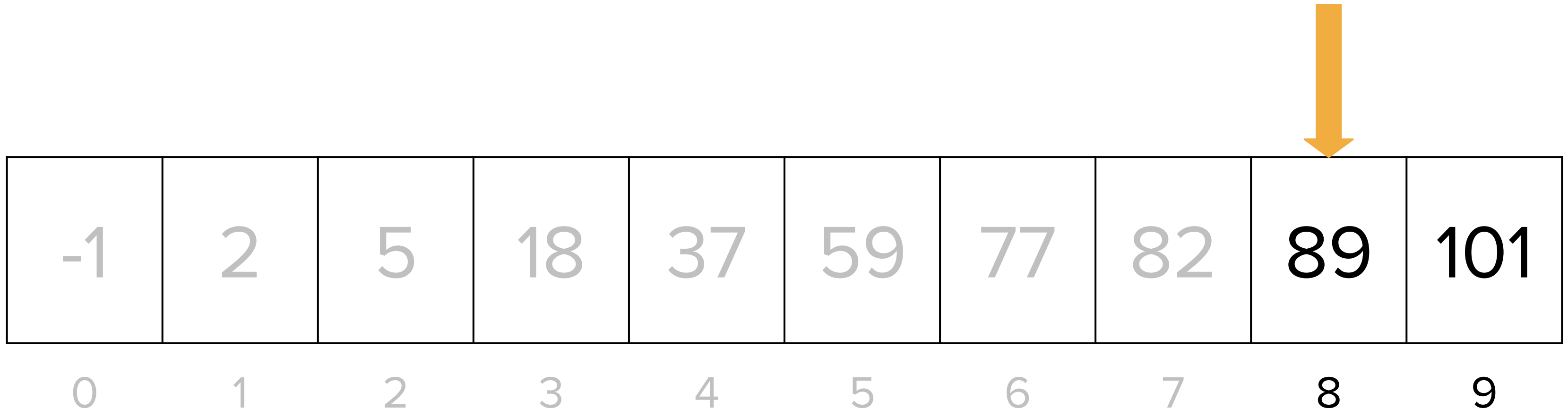
```
(startIndex + endIndex) / 2 =
(8 + 9) / 2 =
8
```

Finding a number in a sorted list



```
(startIndex + endIndex) / 2 =
(8 + 9) / 2 =
8
```

Finding a number in a sorted list



Success!

Defining binary search recursively

- **Algorithm**: Check the middle element at **`(startIndex + endIndex) / 2`**
    - If the middle element is bigger than your desired value, eliminate the right half of the data and repeat.
    - If the middle element is smaller than your desired value, eliminate the left half of the data and repeat.
    - Otherwise, you’ve found your element!

Defining binary search recursively

- **Algorithm**: Check the middle element at **`(startIndex + endIndex) / 2`**
    - If the middle element is bigger than your desired value, eliminate the right half of the data and repeat.
    - If the middle element is smaller than your desired value, eliminate the left half of the data and repeat.
    - Otherwise, you’ve found your element!

- Recursive cases
    - Element at middle is too small ➔ binarySearch(right half of data)
    - Element at middle is too large ➔ binarySearch(left half of data)

Defining binary search recursively

- **Algorithm**: Check the middle element at **`(startIndex + endIndex) / 2`**
  - If the middle element is bigger than your desired value, eliminate the right half of the data and repeat.
  - If the middle element is smaller than your desired value, eliminate the left half of the data and repeat.
  - Otherwise, you've found your element!

- Recursive cases
  - Element at middle is too small ➔ binarySearch(right half of data)
  - Element at middle is too large ➔ binarySearch(left half of data)
- Base cases
  - Element at middle == desired element
  - Desired element is not in your data

Binary search code

```
int binarySearch(Vector<int>& v, int targetVal,
        int startIndex, int endIndex) {
   if (startIndex > endIndex) {
       return -1;
   }

   int middleIndex = (startIndex + endIndex) / 2;
   int currentVal = v[middleIndex];
   if (targetVal == currentVal) {
       return middleIndex;
   } else if (targetVal < currentVal) {
       return binarySearch(v, targetVal, startIndex, middleIndex - 1);
   } else {
       return binarySearch(v, targetVal, middleIndex + 1, endIndex);
   }
}
```

Binary search code

```
int binarySearch(Vector<int>& v, int targetVal, int startIndex, int
endIndex) {
    if (startIndex > endIndex) {
        return -1;
    }

    int middleIndex = (startIndex + endIndex) / 2;
    int currentVal = v[middleIndex];
    if (targetVal == currentVal) {
        return middleIndex;
    } else if (targetVal < currentVal) {
        return binarySearch(v, targetVal, startIndex, middleIndex - 1);
    } else {
        return binarySearch(v, targetVal, middleIndex + 1, endIndex);
    }
}
```

Base cases

Binary search code

```
int binarySearch(Vector<int>& v, int targetVal, int startIndex, int endIndex) {
    if (startIndex > endIndex) {
        return -1;
    }

    int middleIndex = (startIndex + endIndex) / 2;
    int currentVal = v[middleIndex];
    if (targetVal == currentVal) {
        return middleIndex;
    } else if (targetVal < currentVal) {
        return binarySearch(v, targetVal, startIndex, middleIndex - 1);
    } else {
        return binarySearch(v, targetVal, middleIndex + 1, endIndex);
    }
}
```

Recursive cases

Binary search code

```
int binarySearch(Vector<int>& v, int targetVal, int startIndex, int endIndex) {
    if (startIndex > endIndex) {
        return -1;
    }

    int middleIndex = (startIndex + endIndex) / 2;
    int currentVal = v[middleIndex];
    if (targetVal == currentVal) {
        return middleIndex;
    } else if (targetVal < currentVal) {
        return binarySearch(v, targetVal, startIndex, middleIndex - 1);
    } else {
        return binarySearch(v, targetVal, middleIndex + 1, endIndex);
    }
}
```

$\underbrace{\texttt{int startIndex, int endIndex}}_{}$

We don't want the user to have to pass these in, but we need them to update our search range

Binary search code

```
int binarySearch(Vector<int>& v, int targetVal) {
    return binarySearchHelper(v, targetVal, 0, v.size() - 1);
}
```

```
int binarySearchHelper(Vector<int>& v, int targetVal, int startIndex, int endIndex) {
    ...
}
```

*Use a **recursive helper function** for the extra parameters!*

*(**`binarySearchHelper`** would have the same code as the previous slide)*

Finding a number in a sorted list

-1	2	5	18	37	59	77	82	89	101
0	1	2	3	4	5	6	7	8	9

What's the runtime?

Binary search runtime

- For data of size **N**, it eliminates half until 1 element remains:

```
N, N/2, N/4, N/8, ..., 4, 2, 1
```

  - How many divisions does it take?

Binary search runtime

- For data of size **N**, it eliminates half until 1 element remains.

- Think of it from the other direction:
  - How many times do I have to multiply by 2 to reach **N**?

```
1, 2, 4, 8, ..., N/4, N/2, N
```

  - Call this number of multiplications **x**:

$$2^x = N$$
$$\mathbf{x = \log_2 N}$$

Binary search runtime

- For data of size **N**, it eliminates half until 1 element remains.

- Think of it from the other direction:
  - How many times do I have to multiply by 2 to reach **N**?

```
1, 2, 4, 8, ..., N/4, N/2, N
```

  - Call this number of multiplications **x**:

$$2^x = N$$
$$\mathbf{x = \log_2 N}$$

- Binary search has logarithmic Big-O: **`O(log N)`**

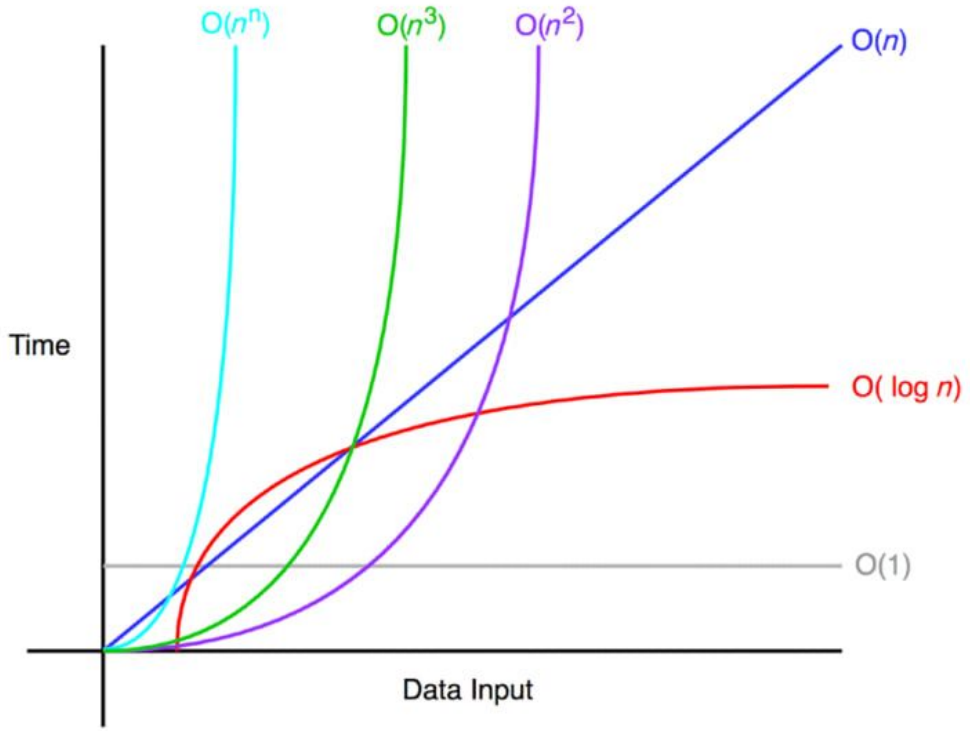
binarysearch.cpp

[demo]

Logarithmic runtime

- Better than linear

- A common runtime when you're able to "divide and conquer" in your algorithm, like with binary search



ADT Big-O Matrix

- Vectors
  - `.size() – O(1)`
  - `.add() – O(1)`
  - `v[i] – O(1)`
  - `.insert() – O(n)`
  - `.remove() – O(n)`
  - `.clear() - O(n)`
  - `traversal – O(n)`
- Grids
  - `.numRows()/.numCols() – O(1)`
  - `g[i][j] – O(1)`
  - `.inBounds() – O(1)`
  - `traversal` – $O(n^2)$
- Queues
  - `.size() – O(1)`
  - `.peek() – O(1)`
  - `.enqueue() – O(1)`
  - `.dequeue() – O(1)`
  - `.isEmpty() – O(1)`
  - `traversal – O(n)`
- Stacks
  - `.size() – O(1)`
  - `.peek() – O(1)`
  - `.push() – O(1)`
  - `.pop() – O(1)`
  - `.isEmpty() – O(1)`
  - `traversal – O(n)`
- Sets
  - `.size() – O(1)`
  - `.isEmpty() – O(1)`
  - `.add() – ???`
  - `.remove() – ???`
  - `.contains() – ???`
  - `traversal – O(n)`
- Maps
  - `.size() – O(1)`
  - `.isEmpty() – O(1)`
  - `m[key] – ???`
  - `.contains() – ???`
  - `traversal – O(n)`

ADT Big-O Matrix

- Vectors
  - `.size() - O(1)`
  - `.add() – O(1)`
  - `v[i] - O(1)`
  - `.insert() – O(n)`
  - `.remove() – O(n)`
  - `.clear() - O(n)`
  - `traversal – O(n)`
- Grids
  - `.numRows()/.numCols() – O(1)`
  - `g[i][j] - O(1)`
  - `.inBounds() – O(1)`
  - `traversal – O(n`^{2}`)`
- Queues
  - `.size() - O(1)`
  - `.peek() – O(1)`
  - `.enqueue() – O(1)`
  - `.dequeue() – O(1)`
  - `.isEmpty() – O(1)`
  - `traversal – O(n)`
- Stacks
  - `.size() - O(1)`
  - `.peek() – O(1)`
  - `.push() – O(1)`
  - `.pop() – O(1)`
  - `.isEmpty() – O(1)`
  - `traversal – O(n)`
- Sets
  - `.size() – O(1)`
  - `.isEmpty() – O(1)`
  - `.add() – O(log(n))`
  - `.remove() – O(log(n))`
  - `.contains() – O(log(n))`
  - `traversal – O(n)`
- Maps
  - `.size() – O(1)`
  - `.isEmpty() – O(1)`
  - `m[key] – O(log(n))`
  - `.contains() – O(log(n))`
  - `traversal – O(n)`

Announcements

Announcements

- The grace period for A2 **expires** 7/9 at 11:59pm PDT. We will **not** be accepting submissions after that!

- Assignment 3 will be released some time today. If you saw a draft this morning on the website, please redownload the assignment code, because we've changed some of the short answer Qs.

- We will be releasing more concrete information about the diagnostic (including practice problems) over the weekend.

- The A3 YEAH video will be posted shortly on Ed. Watching that video / slide combo is the **best** way to start this assignment!

A **dynamic** example: Exploring many possibilities

Limits of iteration

- So far, we've seen problems that could be solved iteratively or recursively.
  - Depending on the problem, you could make the argument that one of the approaches was stylistically preferable or easier to understand.
  - But both got the job done!

Limits of iteration

- So far, we've seen problems that could be solved iteratively or recursively.

- However, there is a whole class of problems that are very difficult, or nearly impossible, to solve with an iterative approach.
    - These problems have the goal of exploring **many different possibilities or solutions.**
    - Because iteration is inherently linear (and not dynamic), it is usually used to build up a single solution without exploring many possible alternatives.
    - Recursion allows us to explore many potential possibilities at once via **the power of branching** that comes when we have multiple recursive calls.

Limits of iteration

- So far, we've seen problems that could be solved iteratively or recursively.

- However, there is a whole class of problems that are very difficult, or nearly impossible, to solve with an iterative approach.

- To solve these problems and generate many possible solutions, we will have to learn a new problem-solving technique called **recursive backtracking**.
    - The key steps in recursive backtracking are that you make a choice about how to generate a solution, you use recursion to explore that choice, and then you might make a different choice and repeat the process.
    - This paradigm is called “**choose-explore-unchoose**.”

Limits of iteration

- So far, we've seen problems that could be solved iteratively or recursively.
- However, there is a whole class of problems that are very difficult, or nearly impossible, to solve with an iterative approach.
- To solve these problems and generate many possible solutions, we will have to learn a new problem-solving technique called **recursive backtracking**.

Let's do an example!

LIBERTY
IN GOD WE TRUST

Generating coin sequences

- Let's say that you're playing a game that involves flipping a coin a certain number of times in a row. Your success in the game depends on the exact sequence of "heads" and "tails" that you get.

Generating coin sequences

- Let's say that you're playing a game that involves flipping a coin a certain number of times in a row. Your success in the game depends on the exact sequence of "heads" and "tails" that you get.

- In the first version of this game, you get 2 coin flips on your turn. What are all the possible outcomes that you could get?

Generating coin sequences

- Let's say that you're playing a game that involves flipping a coin a certain number of times in a row. Your success in the game depends on the exact sequence of "heads" and "tails" that you get.

- In the first version of this game, you get 2 coin flips on your turn. What are all the possible outcomes that you could get?

HH

Generating coin sequences

- Let's say that you're playing a game that involves flipping a coin a certain number of times in a row. Your success in the game depends on the exact sequence of "heads" and "tails" that you get.

- In the first version of this game, you get 2 coin flips on your turn. What are all the possible outcomes that you could get?

HH **HT**

Generating coin sequences

- Let's say that you're playing a game that involves flipping a coin a certain number of times in a row. Your success in the game depends on the exact sequence of "heads" and "tails" that you get.

- In the first version of this game, you get 2 coin flips on your turn. What are all the possible outcomes that you could get?

HH **HT** **TH**

Generating coin sequences

- Let's say that you're playing a game that involves flipping a coin a certain number of times in a row. Your success in the game depends on the exact sequence of "heads" and "tails" that you get.

- In the first version of this game, you get 2 coin flips on your turn. What are all the possible outcomes that you could get?

HH **HT** **TH** **TT**

Generating coin sequences

- Let's say that you're playing a game that involves flipping a coin a certain number of times in a row. Your success in the game depends on the exact sequence of "heads" and "tails" that you get.

- In a different version of the game, you instead get three flips of the coin on your turn. What are all the possible ways that your turn could go?

Generating coin sequences

- Let's say that you're playing a game that involves flipping a coin a certain number of times in a row. Your success in the game depends on the exact sequence of "heads" and "tails" that you get.

- In a different version of the game, you instead get three flips of the coin on your turn. What are all the possible ways that your turn could go?

HHH HHT HTH HTT THH THT TTH TTT

Generating coin sequences

- Let's say that you're playing a game that involves flipping a coin a certain number of times in a row. Your success in the game depends on the exact sequence of "heads" and "tails" that you get.

- In a different version of the game, you instead get three flips of the coin on your turn. What are all the possible ways that your turn could go?

HHH HHT HTH HTT THH THT TTH TTT

How do we know that we got all the possibilities? How do we avoid repeats?

Generating coin sequences

- Let's say that you're playing a game that involves flipping a coin a certain number of times in a row. Your success in the game depends on the exact sequence of "heads" and "tails" that you get.

- Can we observe any patterns between the outcomes in the game with 2 flips and the outcomes in the game with 3 flips?

Generating coin sequences

- Let's say that you're playing a game that involves flipping a coin a certain number of times in a row. Your success in the game depends on the exact sequence of "heads" and "tails" that you get.

- Can we observe any patterns between the outcomes in the game with 2 flips and the outcomes in the game with 3 flips?

$$\begin{tikzcd}
\mathtt{HH} \arrow[d] \arrow[dr] & & \mathtt{HT} \arrow[d] \arrow[dr] & & \mathtt{TH} \arrow[d] \arrow[dr] & & \mathtt{TT} \arrow[d] \arrow[dr] & \\
\mathtt{HHH} & \mathtt{HHT} & \mathtt{HTH} & \mathtt{HTT} & \mathtt{THH} & \mathtt{THT} & \mathtt{TTH} & \mathtt{TTT}
\end{tikzcd}$$

Generating coin sequences

$$\begin{tikzcd}
& \mathtt{HH} \arrow[dl] \arrow[dr] & & & \mathtt{HT} \arrow[dl] \arrow[dr] & & & \mathtt{TH} \arrow[dl] \arrow[dr] & & & \mathtt{TT} \arrow[dl] \arrow[dr] & \\
\mathtt{HHH} & & \mathtt{HHT} & \mathtt{HTH} & & \mathtt{HTT} & \mathtt{THH} & & \mathtt{THT} & \mathtt{TTH} & & \mathtt{TTT}
\end{tikzcd}$$

- Can we observe any patterns between the outcomes in the game with 2 flips and the outcomes in the game with 3 flips?
  - There is a self-similar tree-like relationship between the possible outcomes of 2 flips and the possible outcomes of 3 flips.
  - The branching in the tree comes from deciding whether or not to add an H or a T to the existing sequence.
  - Together these branching sequences of decisions define a **decision tree.**

Why decision trees?

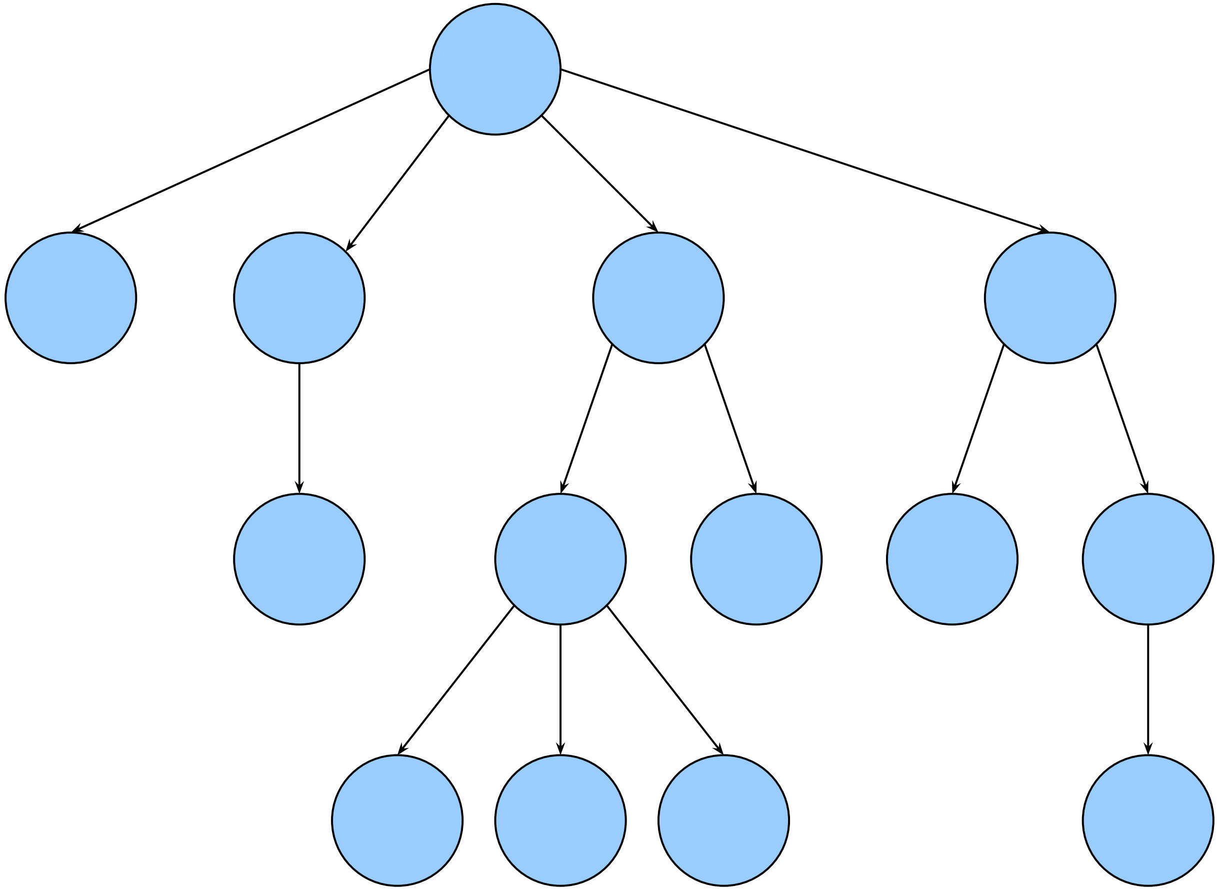
- We've seen trees in the context of fractals (drawing pretty shapes), but now we're going to apply meaningful context to these trees.

Why decision trees?

- We've seen trees in the context of fractals (drawing pretty shapes), but now we're going to apply meaningful context to these trees.

- In problems where we care about many possible outcomes, decision trees can help illustrate the recursive backtracking strategy for generating outcomes. They model the **options we can choose from** and the **“decisions” we make along the way**.

This structure is called a **tree**. Knowing how to model, represent, and manipulate trees in software makes it possible to solve interesting problems.



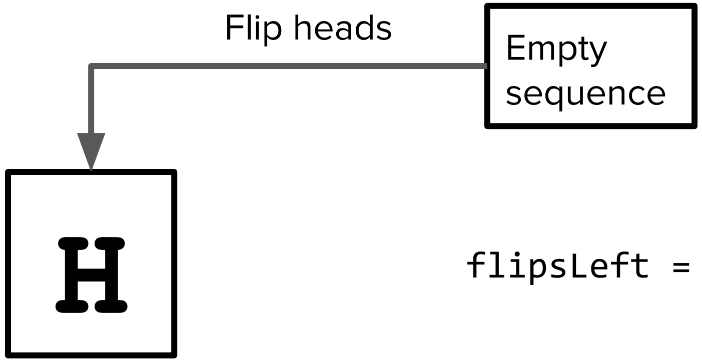
Why decision trees?

- We've seen trees in the context of fractals (drawing pretty shapes), but now we're going to apply meaningful context to these trees.

- In problems where we care about many possible outcomes, decision trees can help illustrate the recursive backtracking strategy for generating outcomes. They model the options we can choose from and the “decisions” we make along the way.

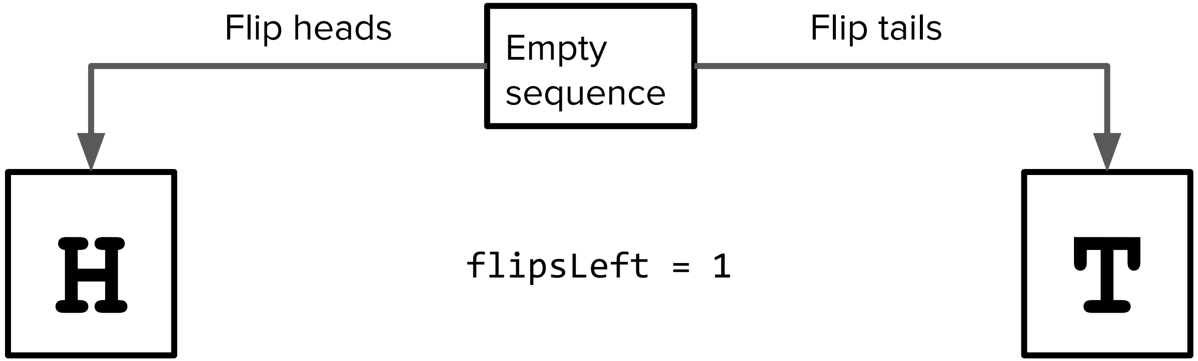
- Let's create a visualization of the possible space of outcomes that could result from N coin flips. Each decision is one flip, and the options for a single flip are either heads or tails.

Example decision tree for N=2



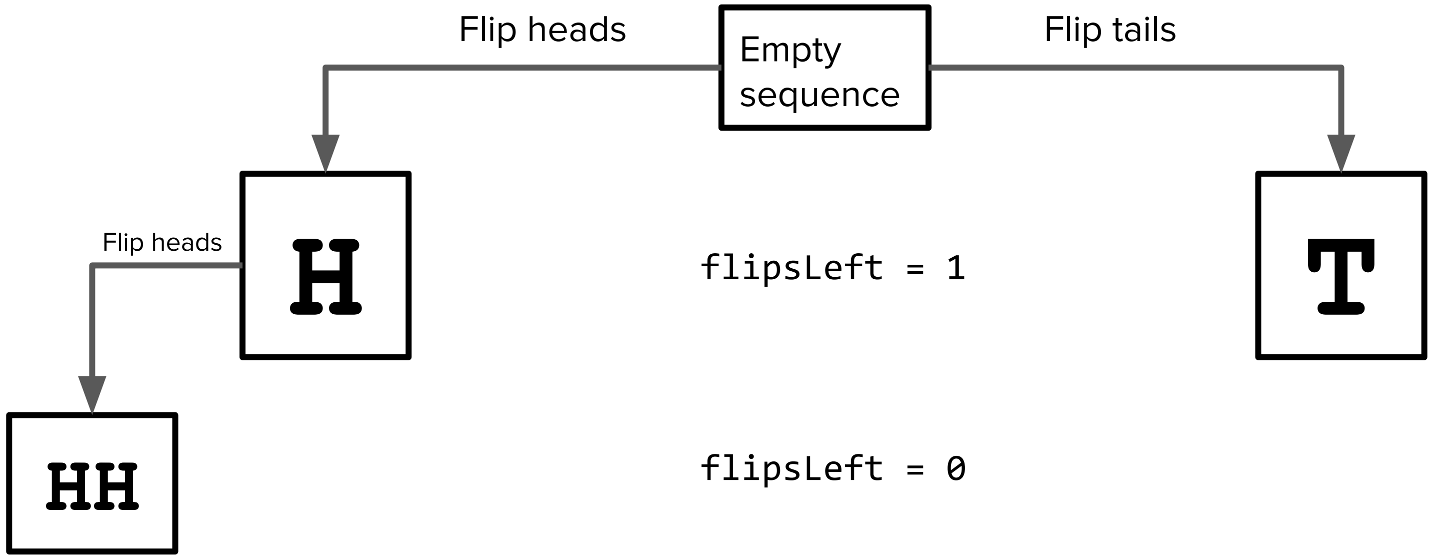
```
flipsLeft = 1
```

Example decision tree for N=2

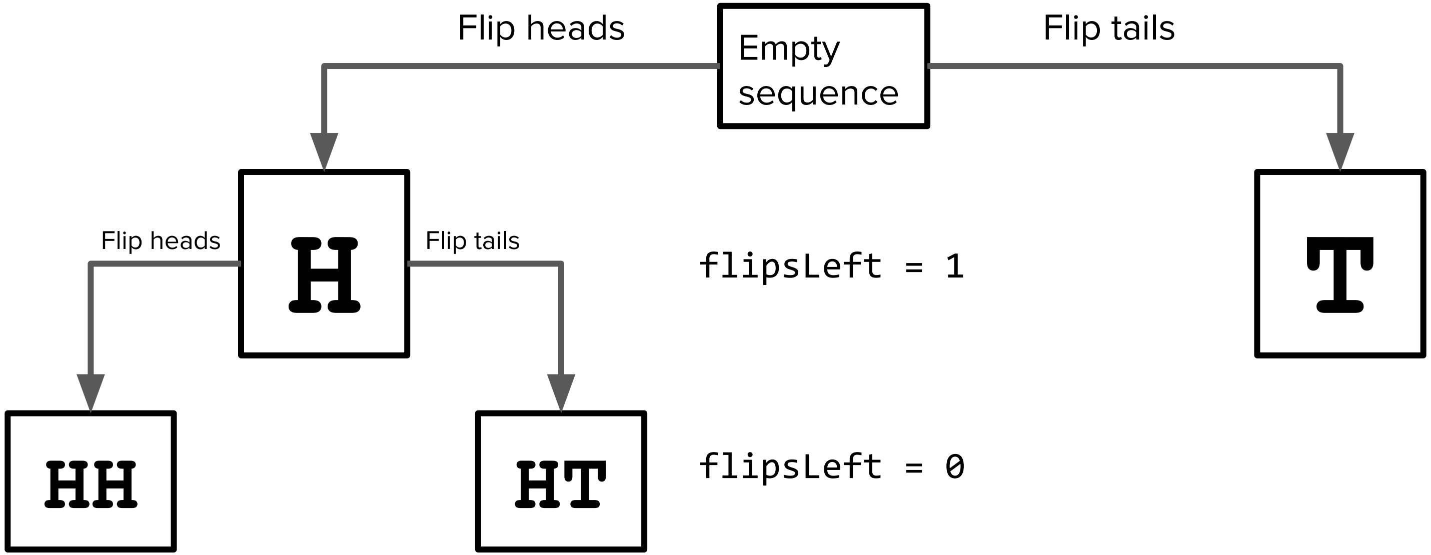


```
flipsLeft = 1
```

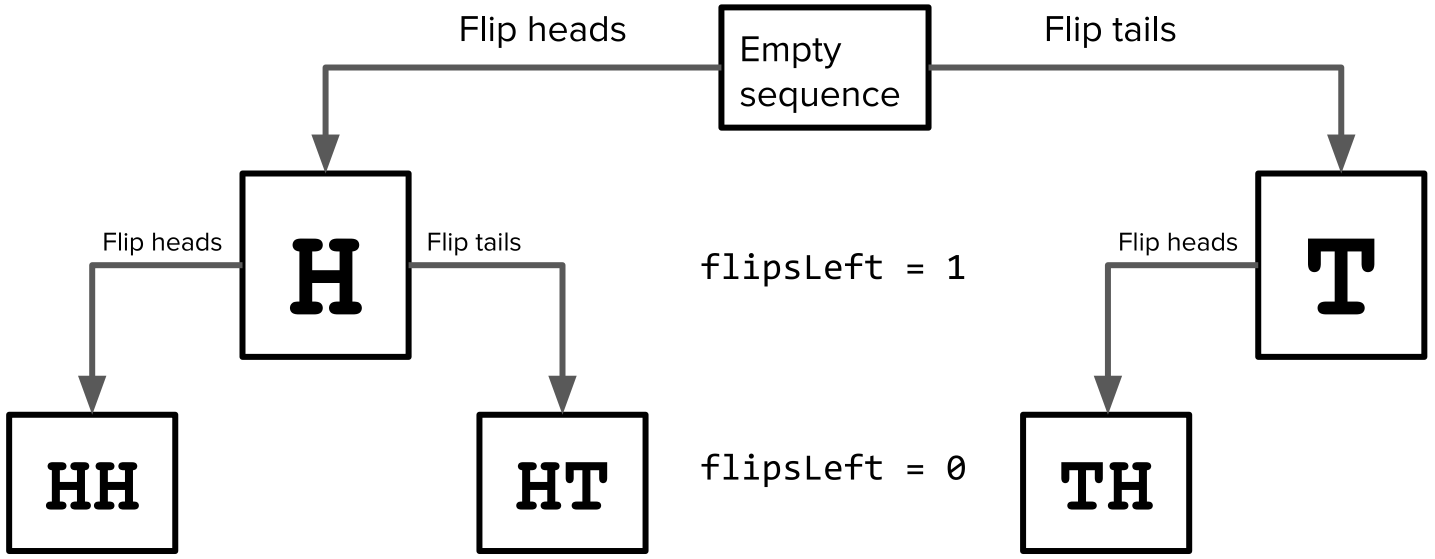
Example decision tree for N=2



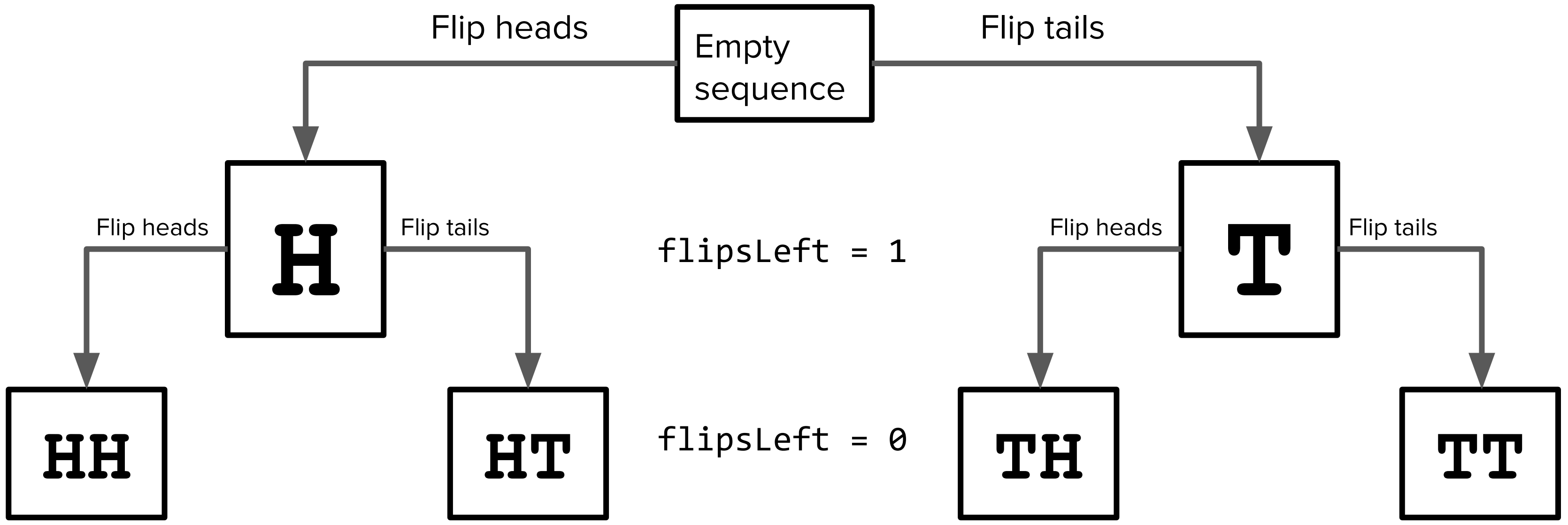
Example decision tree for N=2



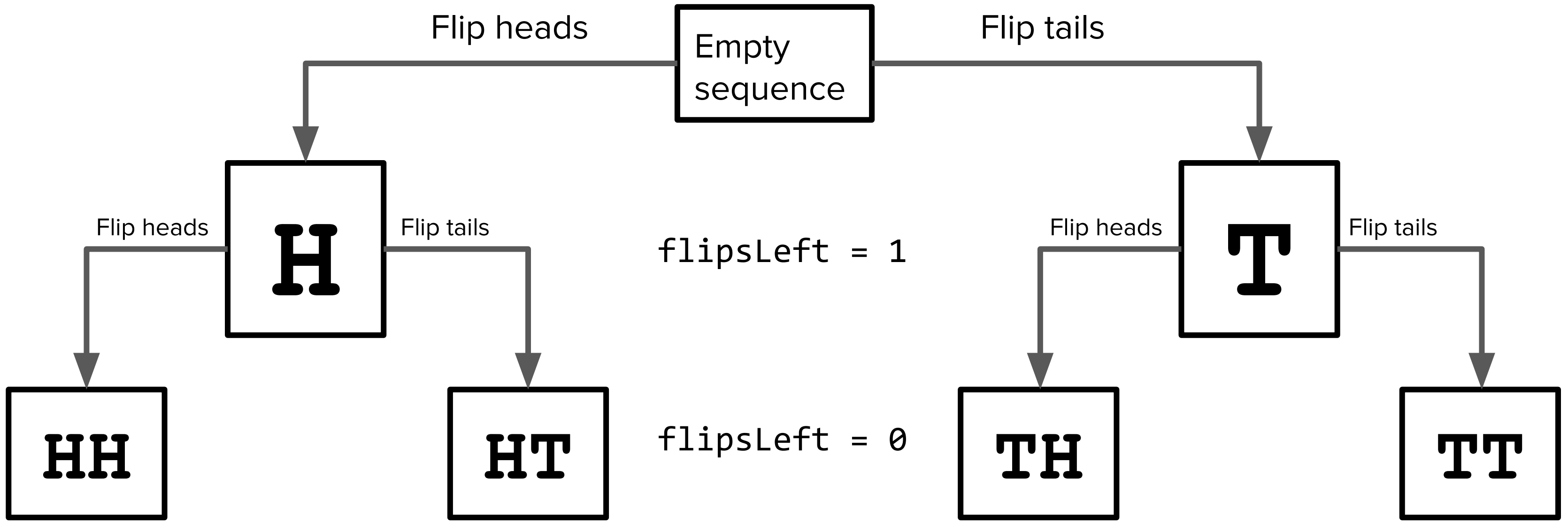
Example decision tree for N=2



Example decision tree for N=2

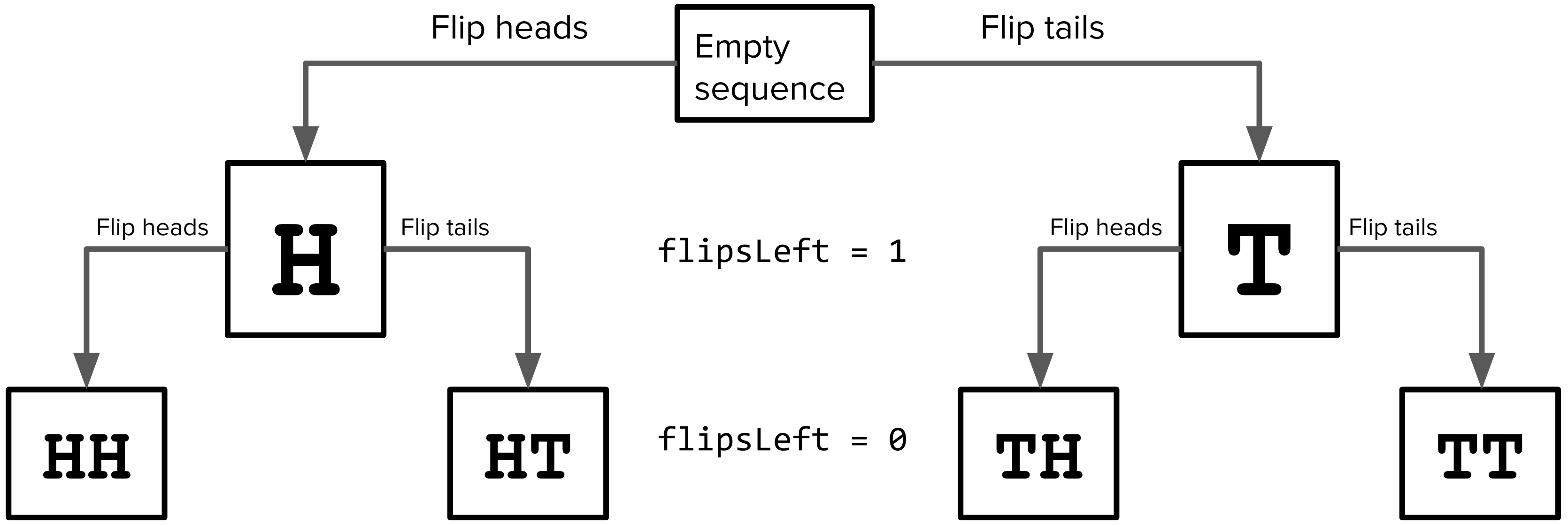


Example decision tree for N=2



Base case: when flipsLeft = 0

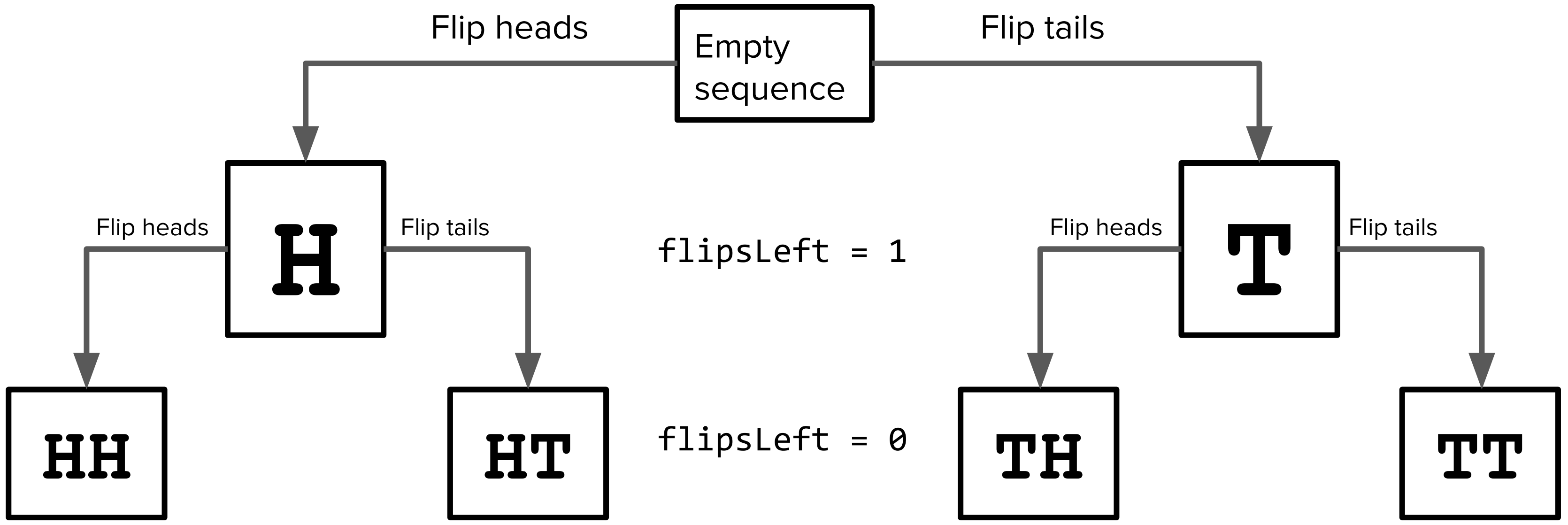
Example decision tree for N=2



Base case: when flipsLeft = 0

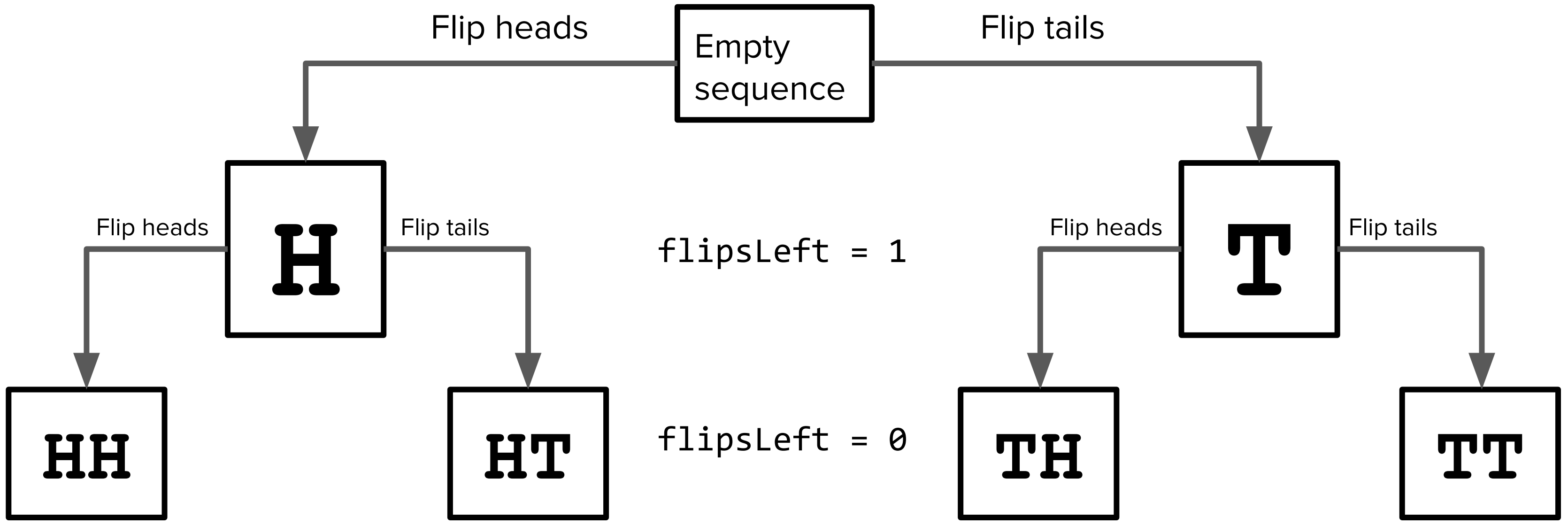
We reach the base case when we reach the leaves of our decision tree.

Example decision tree for N=2



Recursive cases: add ‘H’ **or** ‘T’ to the sequence

Example decision tree for N=2



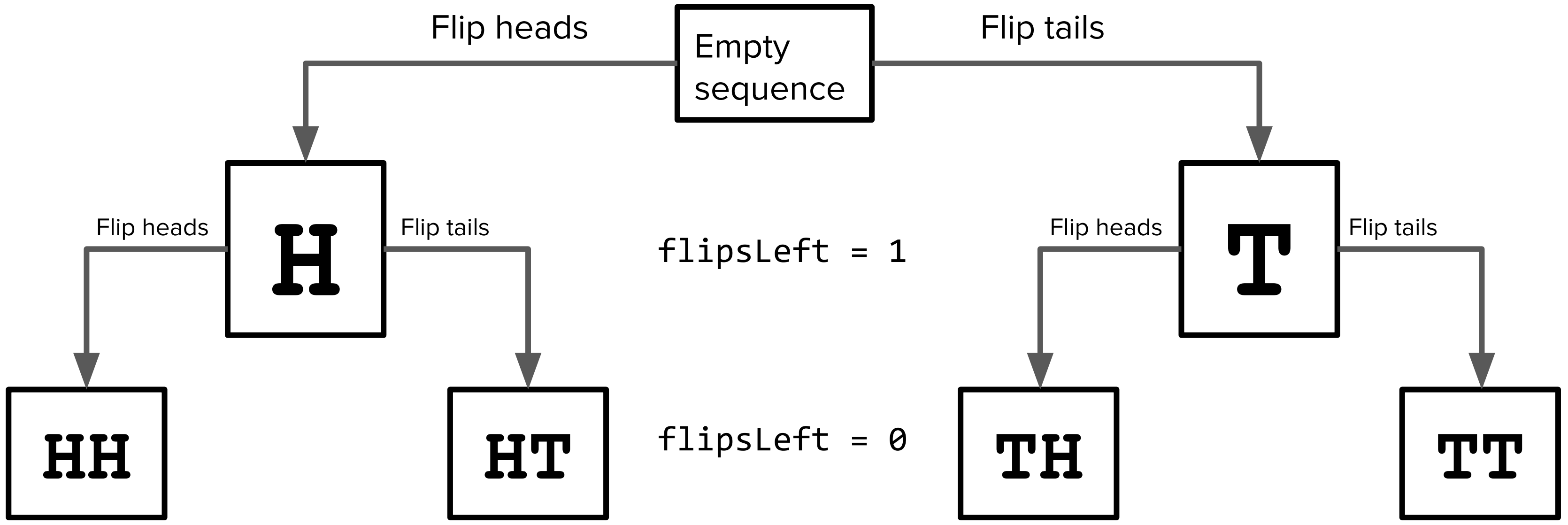
Recursive cases: add ‘H’ **or** ‘T’ to the sequence

The branching points in our tree. We'll have a recursive call for each option.

Let’s code it!

```
void generateSequences(int length);
```

Example decision tree for N=2



Recursive cases: add ‘H’ **or** ‘T’ to the sequence

The branching points in our tree. We'll have a recursive call for each option.

Let’s code it!

```
void generateSequences(int length);
```

Takeaways: recursive backtracking + decision trees

- Unlike our previous recursion paradigm in which a solution gets built up as recursive calls return, in backtracking our final outputs occur at our base cases (leaves) and get built up as we go down the decision tree.

Takeaways: recursive backtracking + decision trees

- Unlike our previous recursion paradigm in which a solution gets built up as recursive calls return, in backtracking our final outputs occur at our base cases (leaves) and get built up as we go down the decision tree.

- The **height** of the tree corresponds to the **number of decisions** we have to make. The **width** at each decision point corresponds to the **number of options**.

Takeaways: recursive backtracking + decision trees

- Unlike our previous recursion paradigm in which a solution gets built up as recursive calls return, in backtracking our final outputs occur at our base cases (leaves) and get built up as we go down the decision tree.

- The **height** of the tree corresponds to the **number of decisions** we have to make. The **width** at each decision point corresponds to the **number of options**.

- To exhaustively explore the entire search space, we must **try every possible option for every possible decision**. That can be a lot of paths to walk!
    - For the previous example, we have to make N decisions, with 2 choices for each decision. This means 2^N total possible outcomes!

Why do we use recursion?

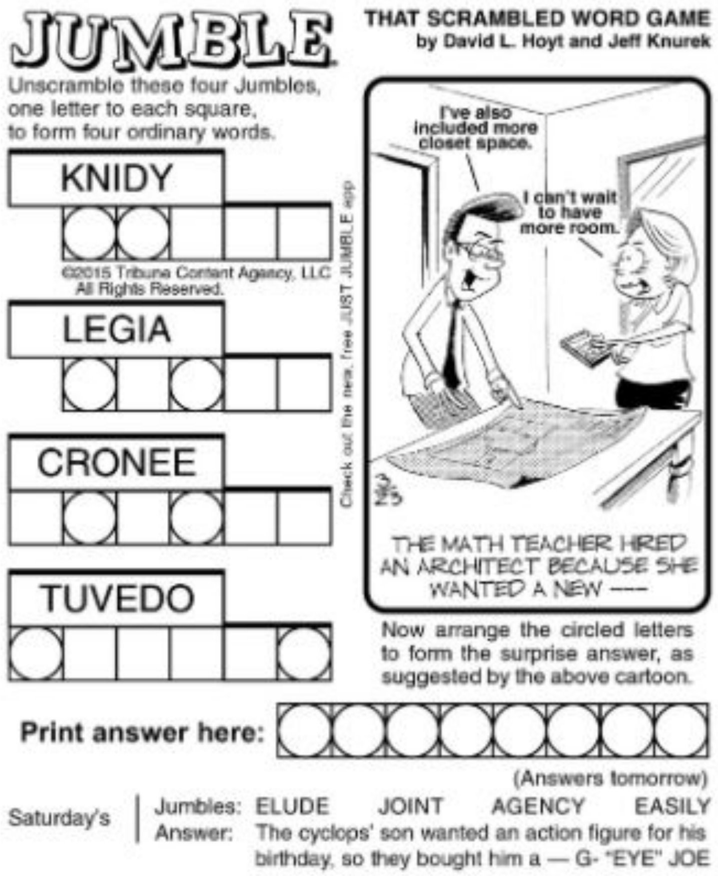
- Elegance
  - Allows us to solve problems with very clean and concise code
- Efficiency
  - Allows us to accomplish better runtimes when solving problems
- Dynamic
  - Allows us to solve problems that are hard to solve iteratively

Word Scramble

Jumble

- Since 1954, the JUMBLE word puzzle has been a staple in newspapers.

- The basic idea is to unscramble the provided letters to make the words on the left, and then use the letters in the circles as another set of letters to unscramble to answer the pun in the comic.


I've also included more closet space.
I can't wait to have more room.

JUMBLE THAT SCRAMBLED WORD GAME by David L. Hoyt and Jeff Knurek

Unscramble these four Jumbles, one letter to each square, to form four ordinary words.

KNIDY

LEGIA

CRONEE

TUVEDO

©2015 Tribune Content Agency, LLC All Rights Reserved.

Check out the new, free JUST JUMBLE app

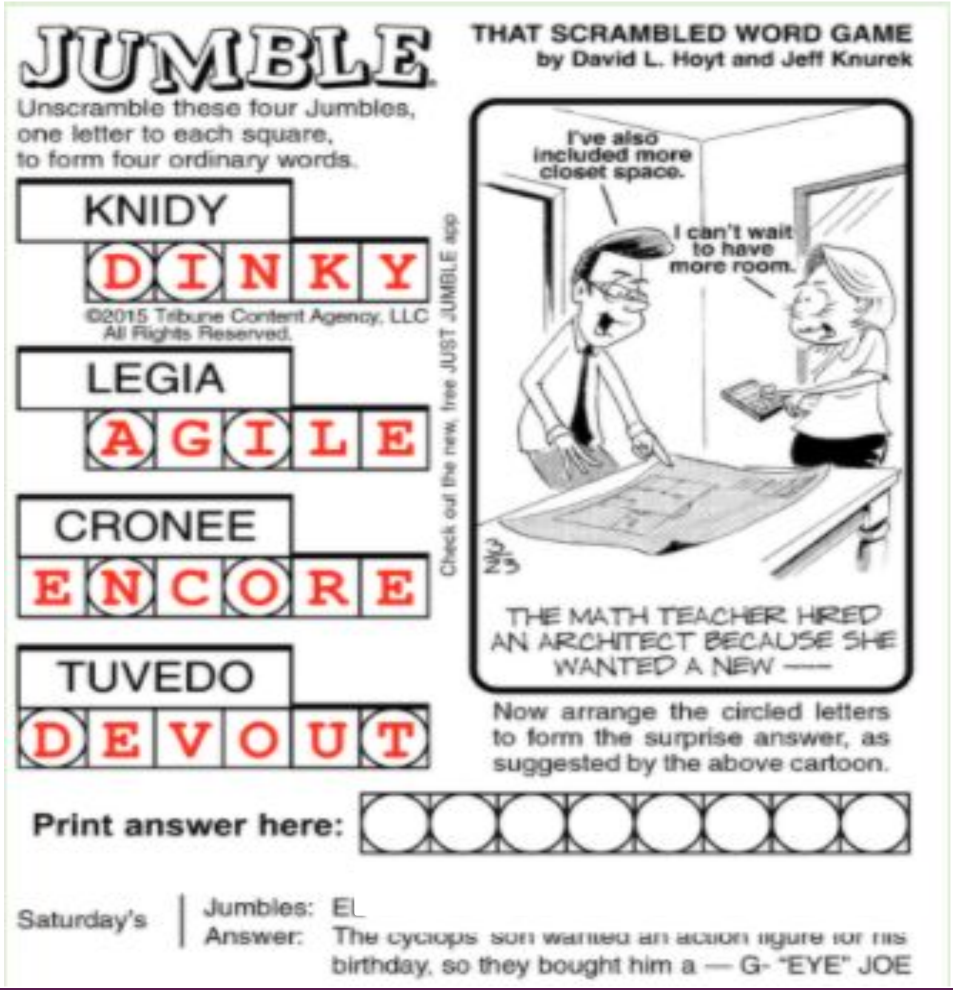
THE MATH TEACHER HIRED AN ARCHITECT BECAUSE SHE WANTED A NEW ---

Now arrange the circled letters to form the surprise answer, as suggested by the above cartoon.

Print answer here:

(Answers tomorrow)

Saturday's	Jumbles:	ELUDE	JOINT	AGENCY	EASILY
	Answer:	The cyclops' son wanted an action figure for his birthday, so they bought him a — G- "EYE" JOE			



JUMBLE

THAT SCRAMBLED WORD GAME
by David L. Hoyt and Jeff Knurek

Unscramble these four Jumbles, one letter to each square, to form four ordinary words.

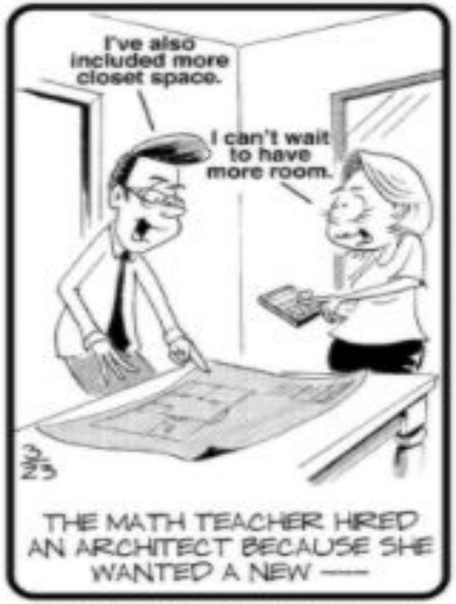
KNIDY
D I N K Y

LEGIA
A G I L E

CRONEE
E N C O R E

TUVEDO
D E V O U T

Check out the new, free JUST JUMBLE app


I've also included more closet space.
I can't wait to have more room.
THE MATH TEACHER HIRED AN ARCHITECT BECAUSE SHE WANTED A NEW ——

Now arrange the circled letters to form the surprise answer, as suggested by the above cartoon.

Print answer here:

Saturday's | Jumbles: EL
Answer: The cyclops' son wanted an action figure for his birthday, so they bought him a — G- "EYE" JOE

JUMBLE

THAT SCRAMBLED WORD GAME
by David L. Hoyt and Jeff Knurek

Unscramble these four Jumbles, one letter to each square, to form four ordinary words.

KNIDY

D I N K Y

LEGIA

A G I L E

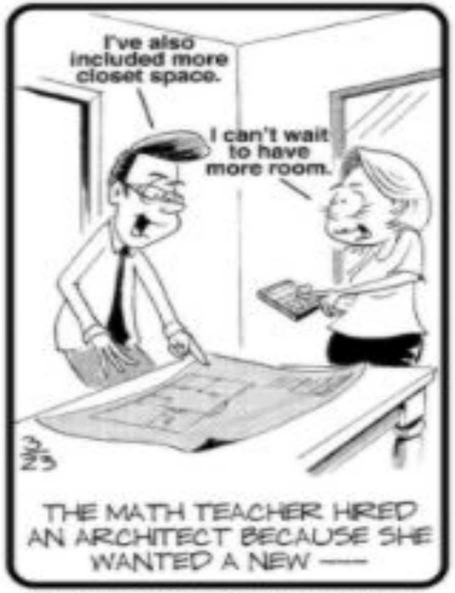
CRONEE

E N C O R E

TUVEDO

D E V O U T

Check out the new, free JUST JUMBLE app


I've also included more closet space.
I can't wait to have more room.

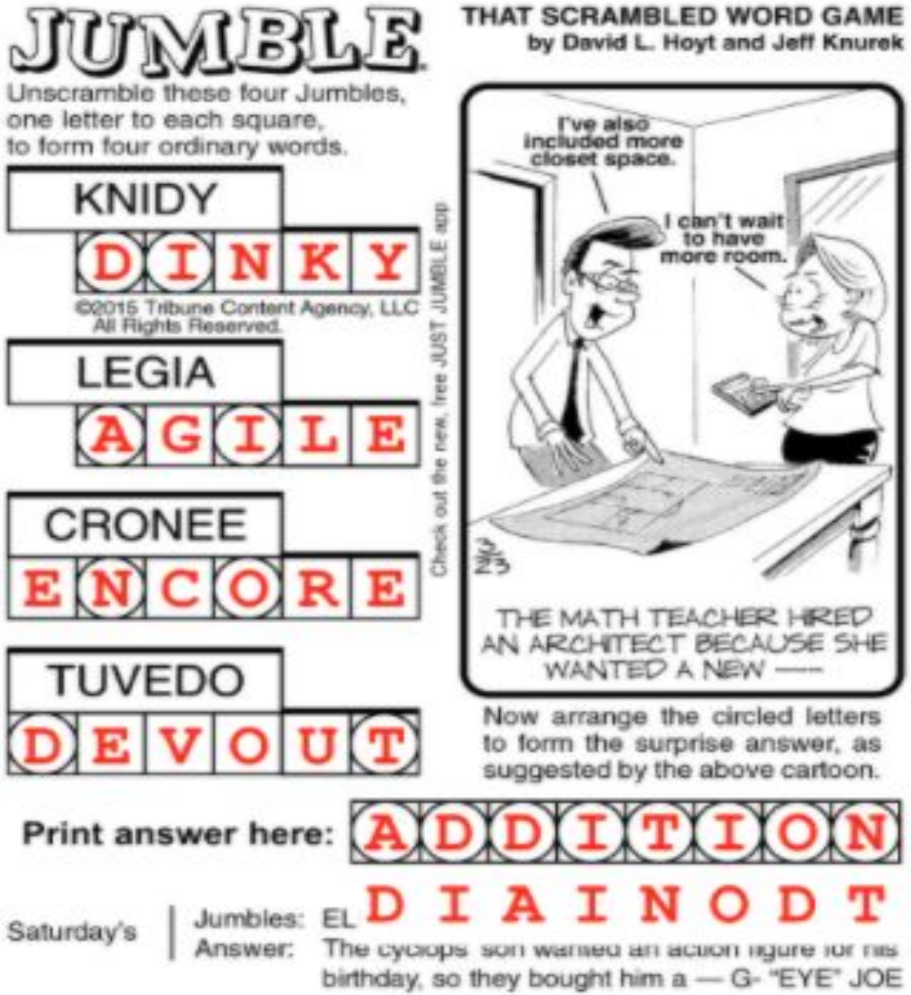
THE MATH TEACHER HIRED AN ARCHITECT BECAUSE SHE WANTED A NEW ——

Now arrange the circled letters to form the surprise answer, as suggested by the above cartoon.

Print answer here:

D I A I N O D T

Saturday's | Jumbles: EL
Answer: The cyclops' son wanted an action figure for his birthday, so they bought him a — G- "EYE" JOE



JUMBLE

THAT SCRAMBLED WORD GAME
by David L. Hoyt and Jeff Knurek

Unscramble these four Jumbles, one letter to each square, to form four ordinary words.

KNIDY

D I N K Y

LEGIA

A G I L E

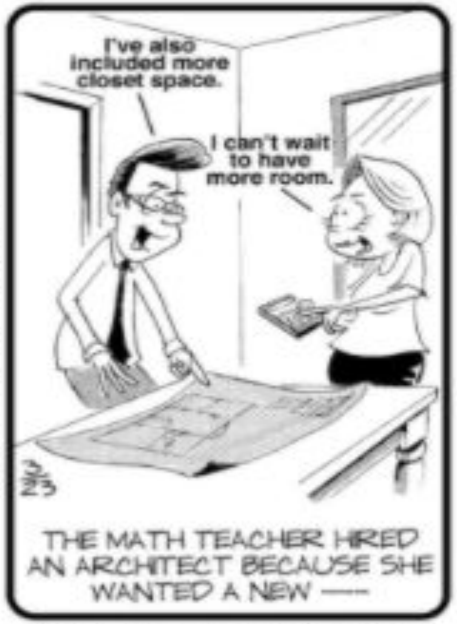
CRONEE

E N C O R E

TUVEDO

D E V O U T

Check out the new, free JUST JUMBLE app


I've also included more closet space.
I can't wait to have more room.
THE MATH TEACHER HIRED AN ARCHITECT BECAUSE SHE WANTED A NEW ——

Now arrange the circled letters to form the surprise answer, as suggested by the above cartoon.

Print answer here: A D D I T I O N

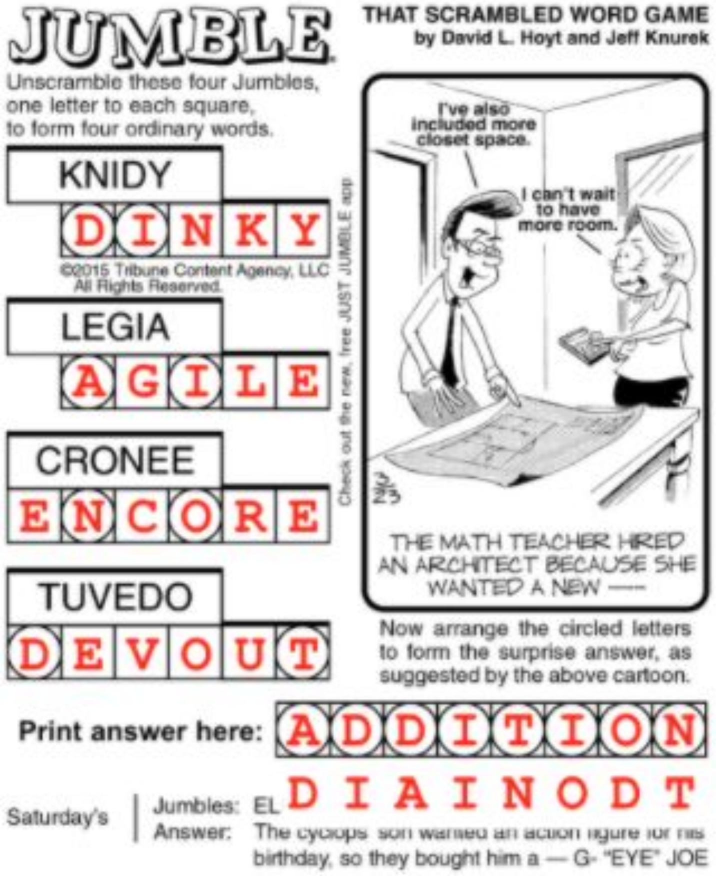
D I A I N O D T

Saturday's | Jumbles: EL
Answer: The cyclops' son wanted an action figure for his birthday, so they bought him a — G- "EYE" JOE

Jumble

- For some people solving puzzles like this comes pretty easily, but this is actually a pretty challenging problem!
    - For a 6-letter word, there are 6! = 720 possible arrangements of the letters

- Can we write a program to print out all the combinations to help us solve this puzzle?


JUMBLE THAT SCRAMBLED WORD GAME
by David L. Hoyt and Jeff Knurek

Unscramble these four Jumbles, one letter to each square, to form four ordinary words.

KNIDY
D I N K Y

©2015 Tribune Content Agency, LLC
All Rights Reserved.

LEGIA
A G I L E

CRONEE
E N C O R E

TUVEDO
D E V O U T

Check out the new, free JUST JUMBLE app

I've also included more closet space.
I can't wait to have more room.

THE MATH TEACHER HIRED AN ARCHITECT BECAUSE SHE WANTED A NEW —

Now arrange the circled letters to form the surprise answer, as suggested by the above cartoon.

Print answer here: A D D I T I O N

D I A I N O D T

Saturday's | Jumbles: EL
Answer: The cyclops son wanted an action figure for his birthday, so they bought him a — G- "EYE" JOE

Permutations

Permutations

- A **permutation** of a sequence is a sequence with the same elements, though possibly in a different order.

Permutations

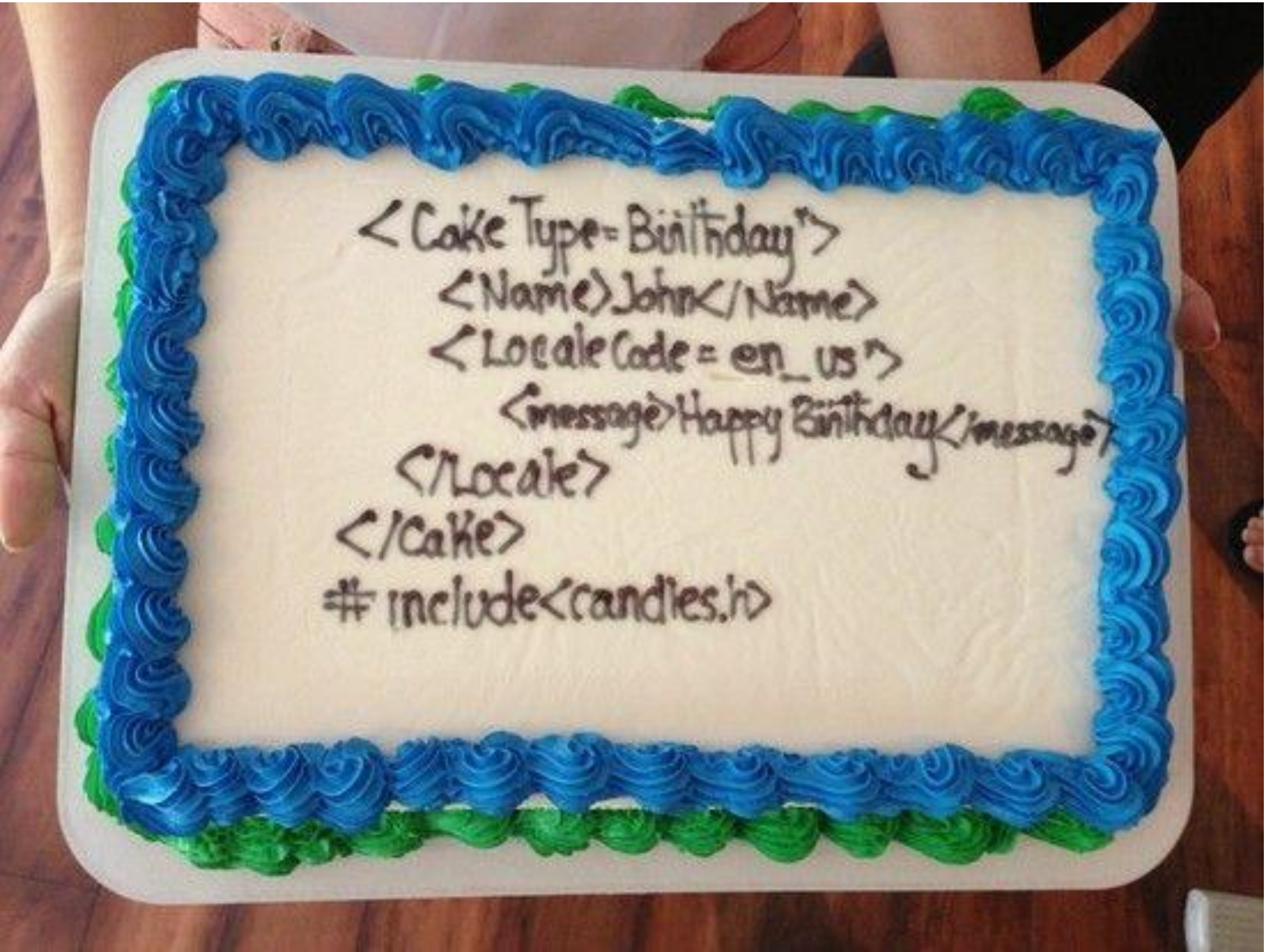
- A **permutation** of a sequence is a sequence with the same elements, though possibly in a different order.

Permutations

- A **permutation** of a sequence is a sequence with the same elements, though possibly in a different order.
- For example, permutations of the words in the motto "E Pluribus Unum" would be:
  - E Pluribus Unum
  - E Unum Pluribus
  - Pluribus E Unum
  - Pluribus Unum E
  - Unum E Pluribus
  - Unum Pluribus E

Permutations

- A **permutation** of a sequence is a sequence with the same elements, though possibly in a different order.
- We can think of permutations as an extension of the coin flip sequences we generated yesterday.
  - Rather than having 2 fixed options (heads and tails), the components of our original sequence define the options we can use to build our new sequence.

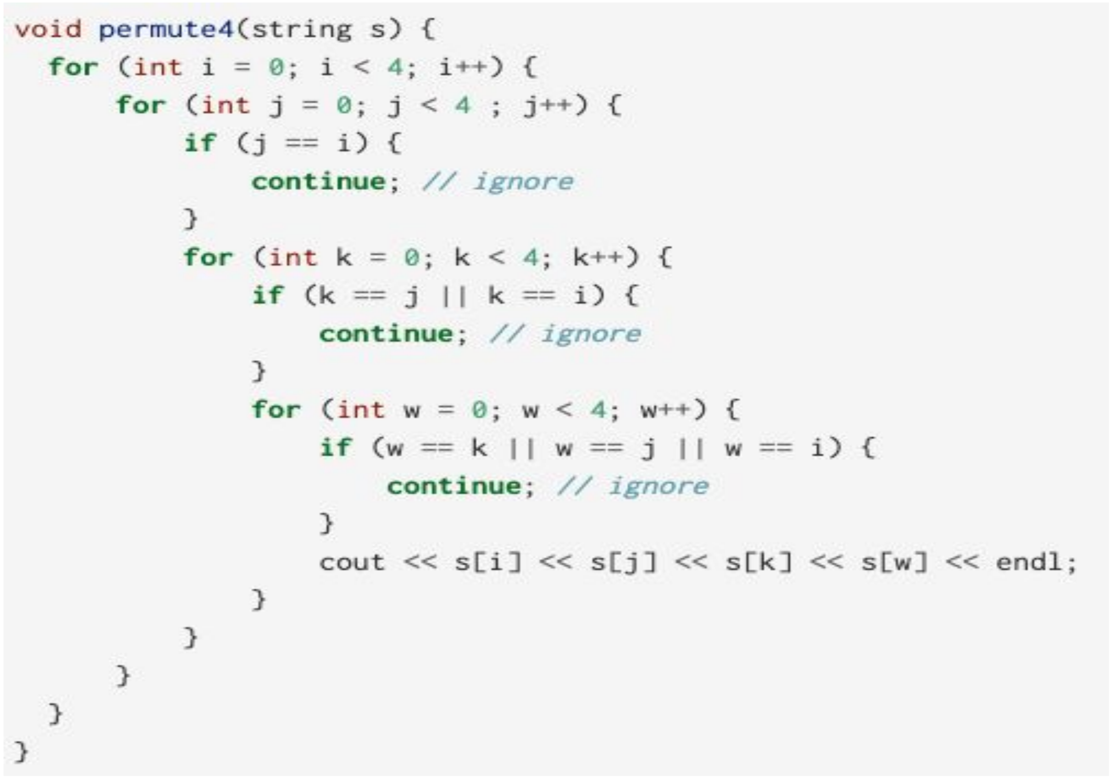


```
<Cake Type=Birthday">
    <Name>John</Name>
    <LocaleCode=en_us">
        <message>Happy Birthday</message>
   </Locale>
 </Cake>
#include<candies.h>
```

CAKE

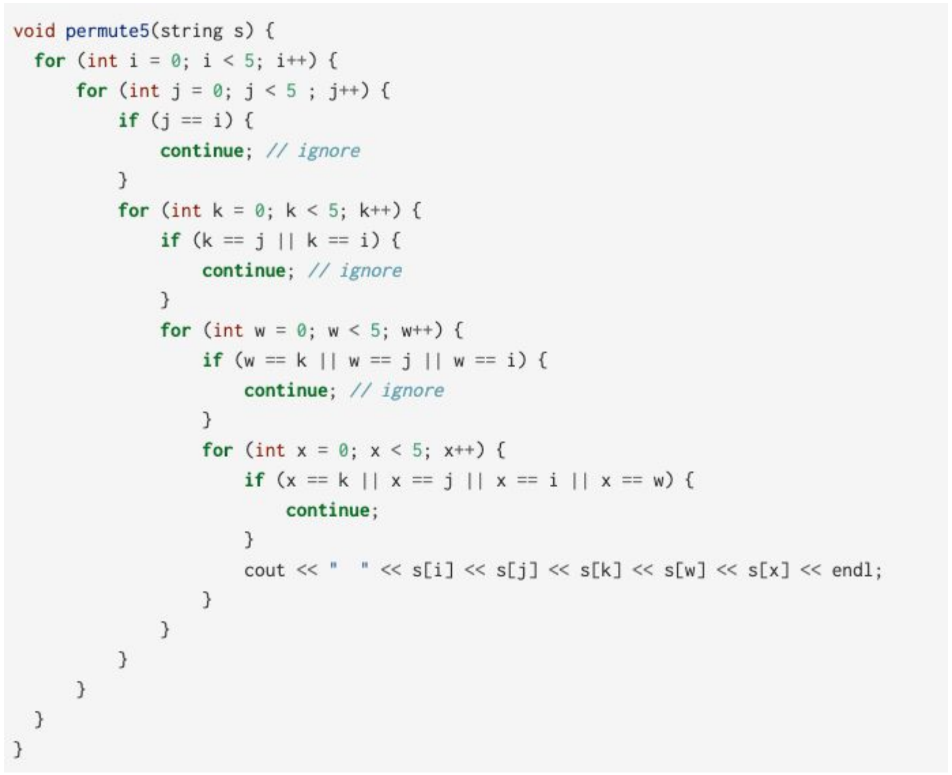
Can you solve a backtracking problem iteratively?

Can you solve a backtracking problem iteratively?



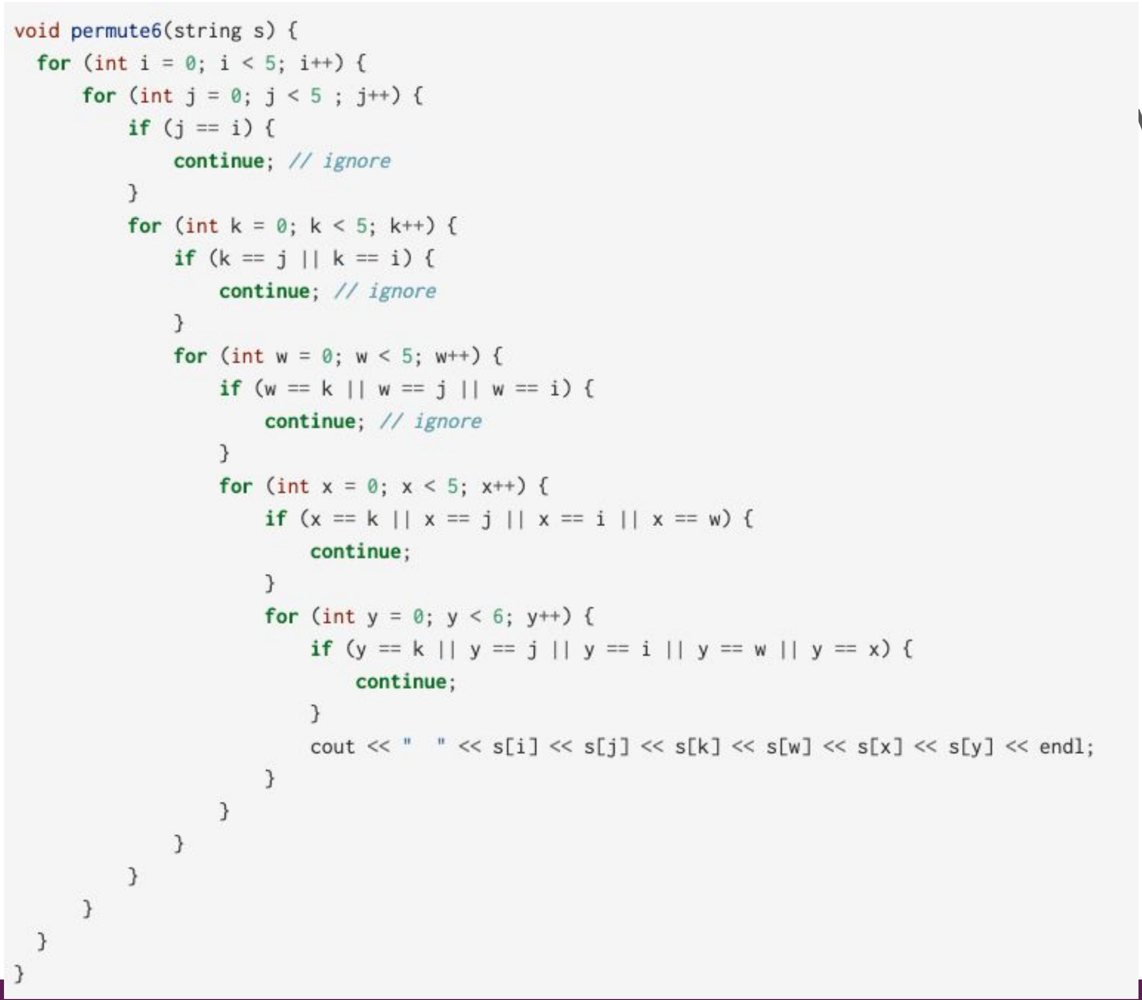
```
void permute4(string s) {
  for (int i = 0; i < 4; i++) {
      for (int j = 0; j < 4 ; j++) {
          if (j == i) {
              continue; // ignore
          }
          for (int k = 0; k < 4; k++) {
              if (k == j || k == i) {
                  continue; // ignore
              }
              for (int w = 0; w < 4; w++) {
                  if (w == k || w == j || w == i) {
                      continue; // ignore
                  }
                  cout << s[i] << s[j] << s[k] << s[w] << endl;
              }
          }
      }
  }
}
```

Can you solve a backtracking problem iteratively?



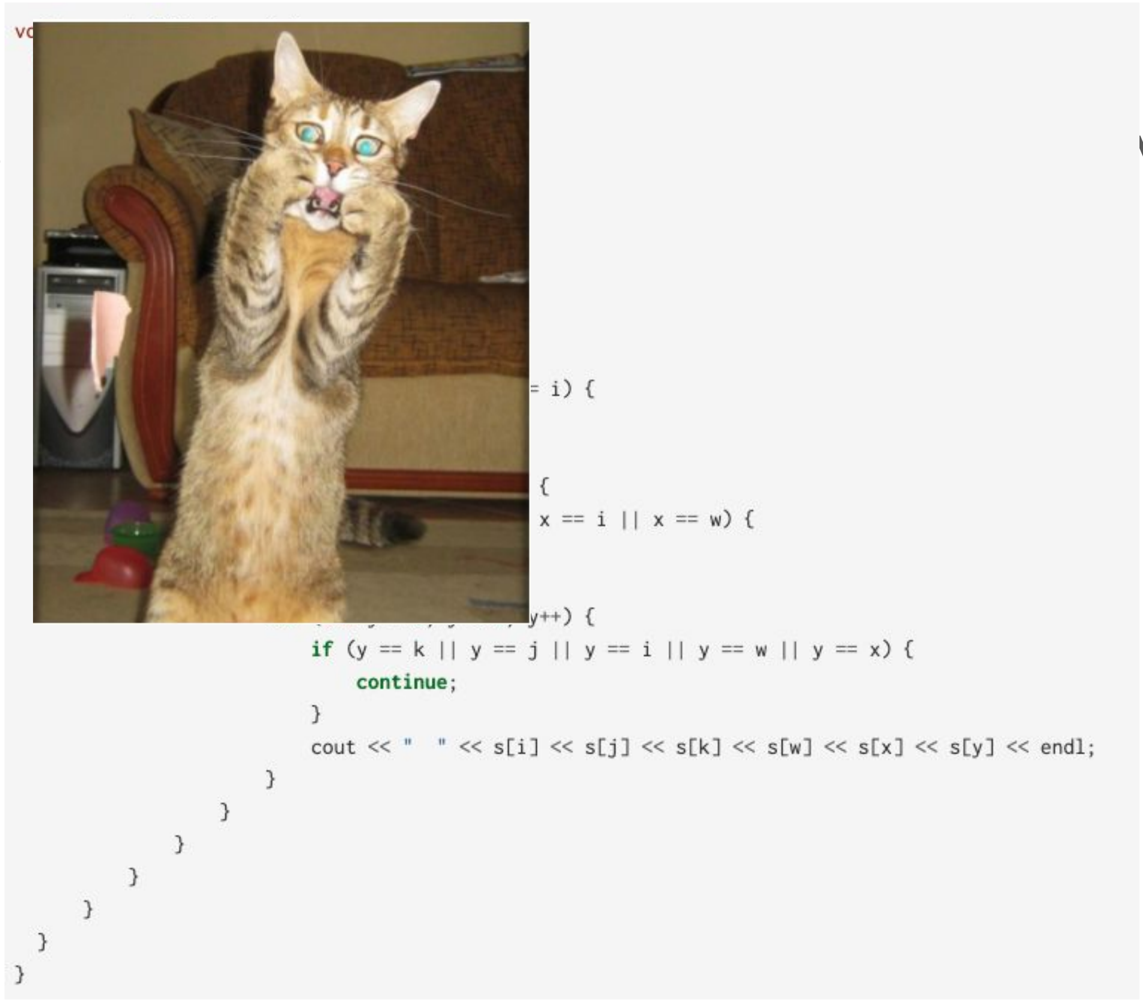
```
void permute5(string s) {
 for (int i = 0; i < 5; i++) {
   for (int j = 0; j < 5 ; j++) {
     if (j == i) {
       continue; // ignore
     }
     for (int k = 0; k < 5; k++) {
       if (k == j || k == i) {
         continue; // ignore
       }
       for (int w = 0; w < 5; w++) {
         if (w == k || w == j || w == i) {
           continue; // ignore
         }
         for (int x = 0; x < 5; x++) {
           if (x == k || x == j || x == i || x == w) {
             continue;
           }
           cout << "  " << s[i] << s[j] << s[k] << s[w] << s[x] << endl;
         }
       }
     }
   }
 }
}
```

Can you s vely?

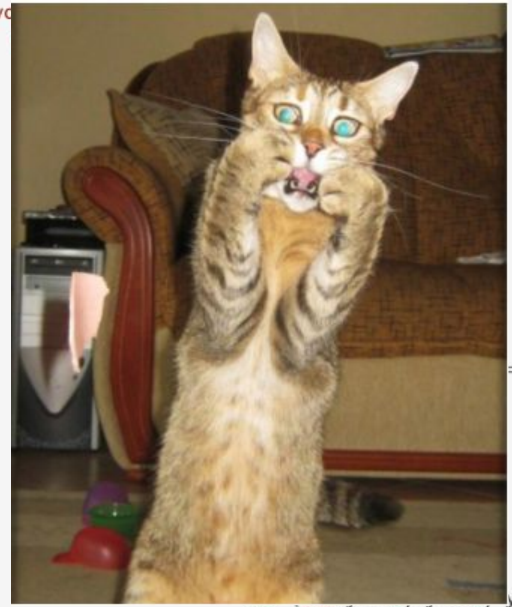


```
void permute6(string s) {
  for (int i = 0; i < 5; i++) {
      for (int j = 0; j < 5 ; j++) {
          if (j == i) {
              continue; // ignore
          }
          for (int k = 0; k < 5; k++) {
              if (k == j || k == i) {
                  continue; // ignore
              }
              for (int w = 0; w < 5; w++) {
                  if (w == k || w == j || w == i) {
                      continue; // ignore
                  }
                  for (int x = 0; x < 5; x++) {
                      if (x == k || x == j || x == i || x == w) {
                          continue;
                      }
                      for (int y = 0; y < 6; y++) {
                          if (y == k || y == j || y == i || y == w || y == x) {
                              continue;
                          }
                          cout << "  " << s[i] << s[j] << s[k] << s[w] << s[x] << s[y] << endl;
                      }
                  }
              }
          }
      }
  }
}
```

Can you s vely?

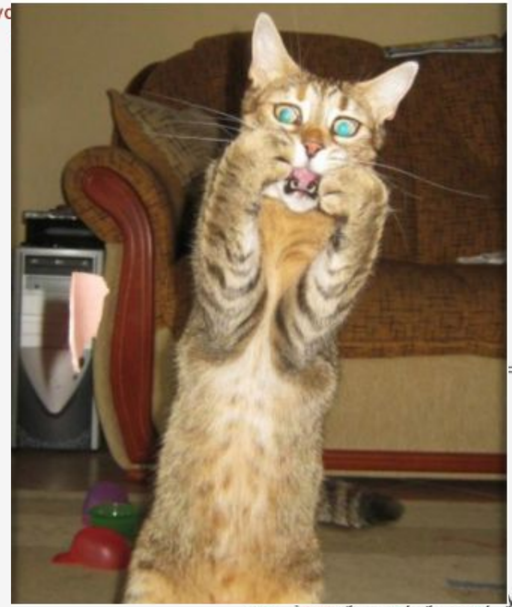


```
vo
                                             = i) {


                                            {
                                            x == i || x == w) {


                                            y++) {
                          if (y == k || y == j || y == i || y == w || y == x) {
                              continue;
                          }
                          cout << "  " << s[i] << s[j] << s[k] << s[w] << s[x] << s[y] << endl;
                      }
                  }
              }
          }
      }
  }
}
```

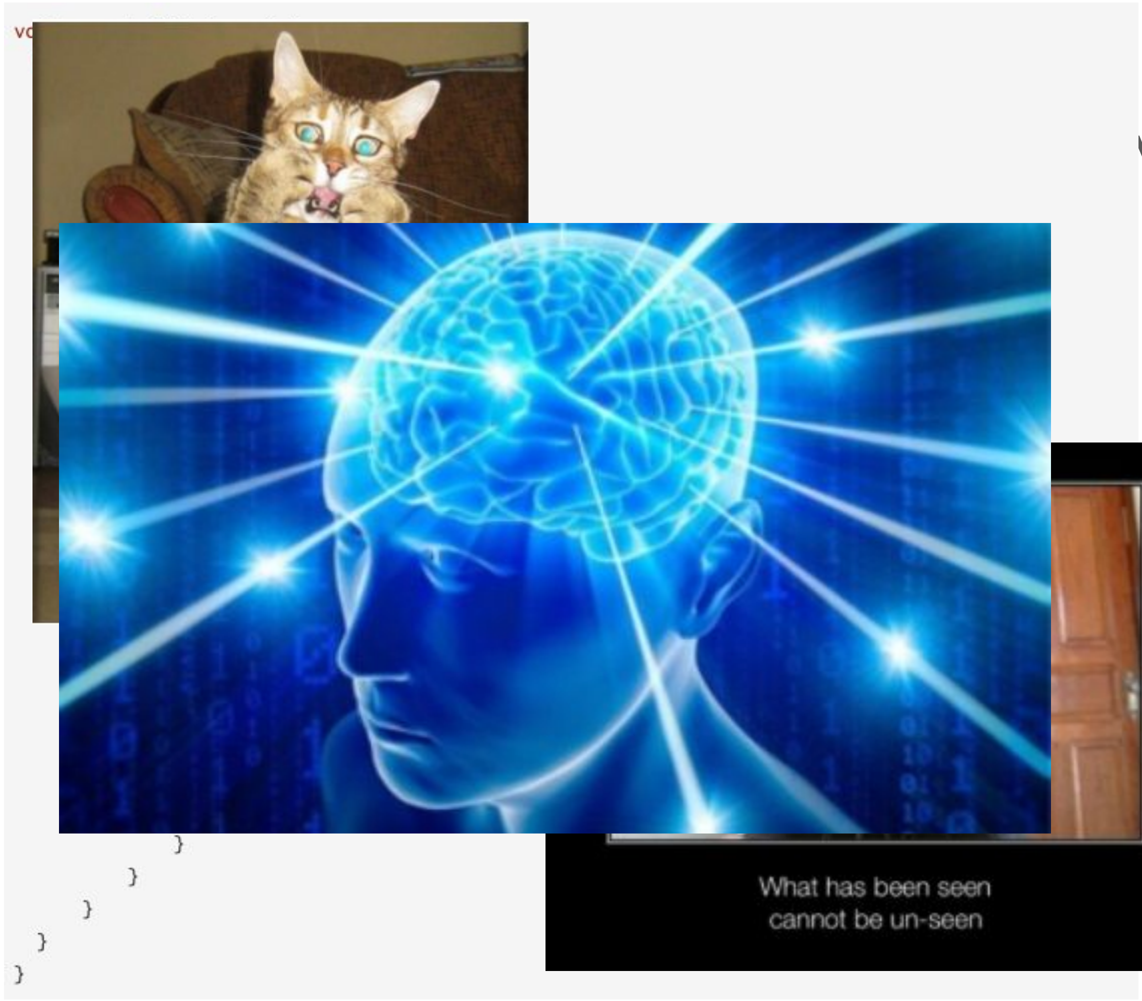


Can you s vely?

```
vo



                         = i) {



                         {
                         x



                         y+
          if (y == k || y == j
              continue;
          }
          cout << "  " << s[i]
        }
      }
    }
  }
 }
}
}
```



Can you s vely?



```
vo
            }
          }
        }
      }
    }
  }
}
```

CAKE

What are all the permutations of the string “cake"?

- “cake"
- "caek"
- "ckae"
- "ckea"
- "ceka"
- "ceak"
- "acke"
- "acek"
- "akce"
- "akec"
- "aeck"
- "aekc"
- "kcae"
- "kcea"
- "kace"
- "kaec"
- "keac"
- "keca"
- "ekac"
- "ekca"
- "eakc"
- "eack"
- "ecka"
- "ecak"

What are all the permutations of the string “cake"?

- “cake"
- "caek"
- "ckae"
- "ckea"
- "ceka"
- "ceak"
- "acke"
- "acek"
- "akce"
- "akec"
- "aeck"
- "aekc"
- "kcae"
- "kcea"
- "kace"
- "kaec"
- "keac"
- "keca"
- "ekac"
- "ekca"
- "eakc"
- "eack"
- "ecka"
- "ecak"

A quarter of the permutations start with “c”, followed by all the permutations of "ake"

What are all the permutations of the string “cake"?

- “cake"
- "caek"
- "ckae"
- "ckea"
- "ceka"
- "ceak"
- "acke"
- "acek"
- "akce"
- "akec"
- "aeck"
- "aekc"
- "kcae"
- "kcea"
- "kace"
- "kaec"
- "keac"
- "keca"
- "ekac"
- "ekca"
- "eakc"
- "eack"
- "ecka"
- "ecak"

A quarter of the permutations start with “a”, followed by all the permutations of “cke”

What are all the permutations of the string “cake"?

- “cake"
- "caek"
- "ckae"
- "ckea"
- "ceka"
- "ceak"
- "acke"
- "acek"
- "akce"
- "akec"
- "aeck"
- "aekc"
- "kcae"
- "kcea"
- "kace"
- "kaec"
- "keac"
- "keca"
- "ekac"
- "ekca"
- "eakc"
- "eack"
- "ecka"
- "ecak"

A quarter of the permutations start with “k”, followed by all the permutations of “cae”

What are all the permutations of the string “cake"?

- “cake"
- "caek"
- "ckae"
- "ckea"
- "ceka"
- "ceak"
- "acke"
- "acek"
- "akce"
- "akec"
- "aeck"
- "aekc"

- "kcae"
- "kcea"
- "kace"
- "kaec"
- "keac"
- "keca"
- "ekac"
- "ekca"
- "eakc"
- "eack"
- "ecka"
- "ecak"

A quarter of the permutations start with “e”, followed by all the permutations of “cak”

Permutations Intution

- “cake"
- "caek"
- "ckae"
- "ckea"
- "ceka"
- "ceak"
- "acke"
- "acek"
- "akce"
- "akec"
- "aeck"
- "aekc"

- "kcae"
- "kcea"
- "eakc"
- "eack"
- "ecka"
- "ecak"

Can we formalize this intuition in a decision tree?

What defines our permutations decision tree?

What defines our permutations decision tree?

- **Decision** at each step (each level of the tree):
  - What is the next letter that is going to get added to the permutation?

What defines our permutations decision tree?

- **Decision** at each step (each level of the tree):
  - What is the next letter that is going to get added to the permutation?

- **Options** at each decision (branches from each node):
  - One option for every remaining element that hasn't been selected yet
  - **Note: The number of options will be different at each level of the tree!**

What defines our permutations decision tree?

- **Decision** at each step (each level of the tree):
  - What is the next letter that is going to get added to the permutation?

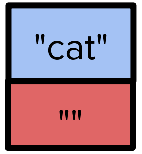
- **Options** at each decision (branches from each node):
  - One option for every remaining element that hasn't been selected yet
  - **Note: The number of options will be different at each level of the tree!**

- Information we need to store along the way:
  - The permutation you've built so far
  - The remaining elements in the original sequence

🟦 Decisions yet to be made

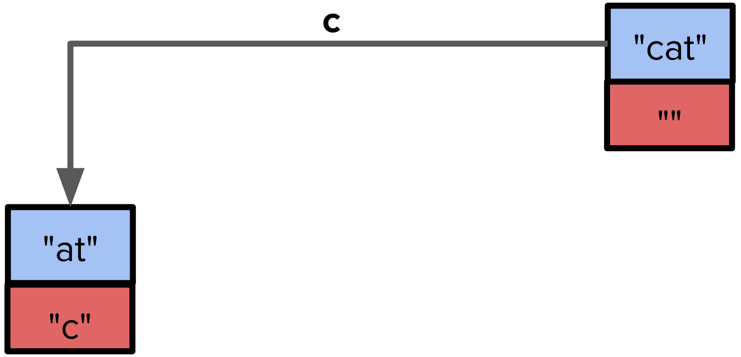
🟥 Decisions made so far

Decision tree: Find all permutations of "cat"



Decisions yet to be made
Decisions made so far

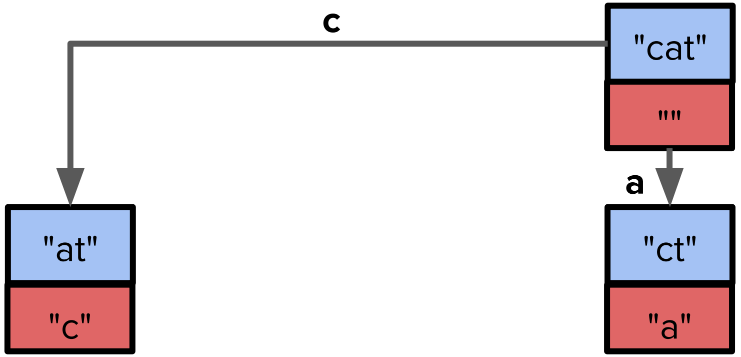
Decision tree: Find all permutations of "cat"



Decisions yet to be made

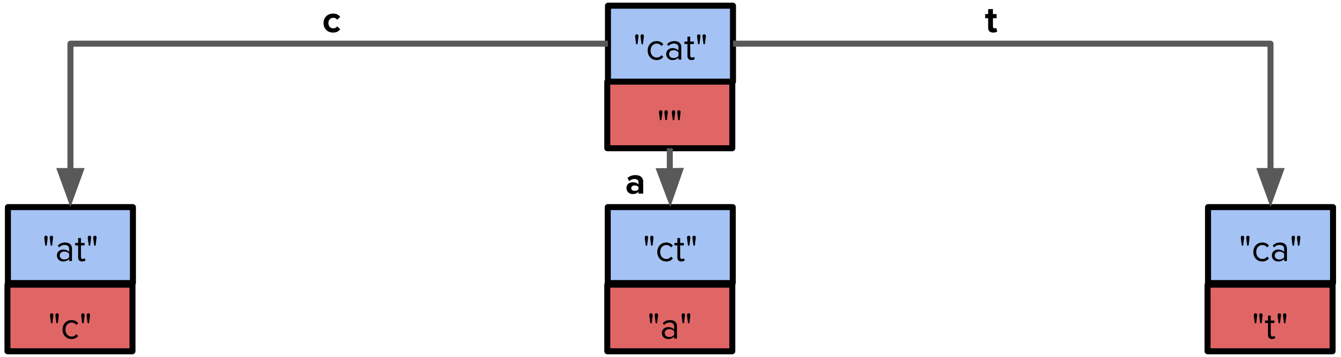
Decisions made so far

Decision tree: Find all permutations of "cat"



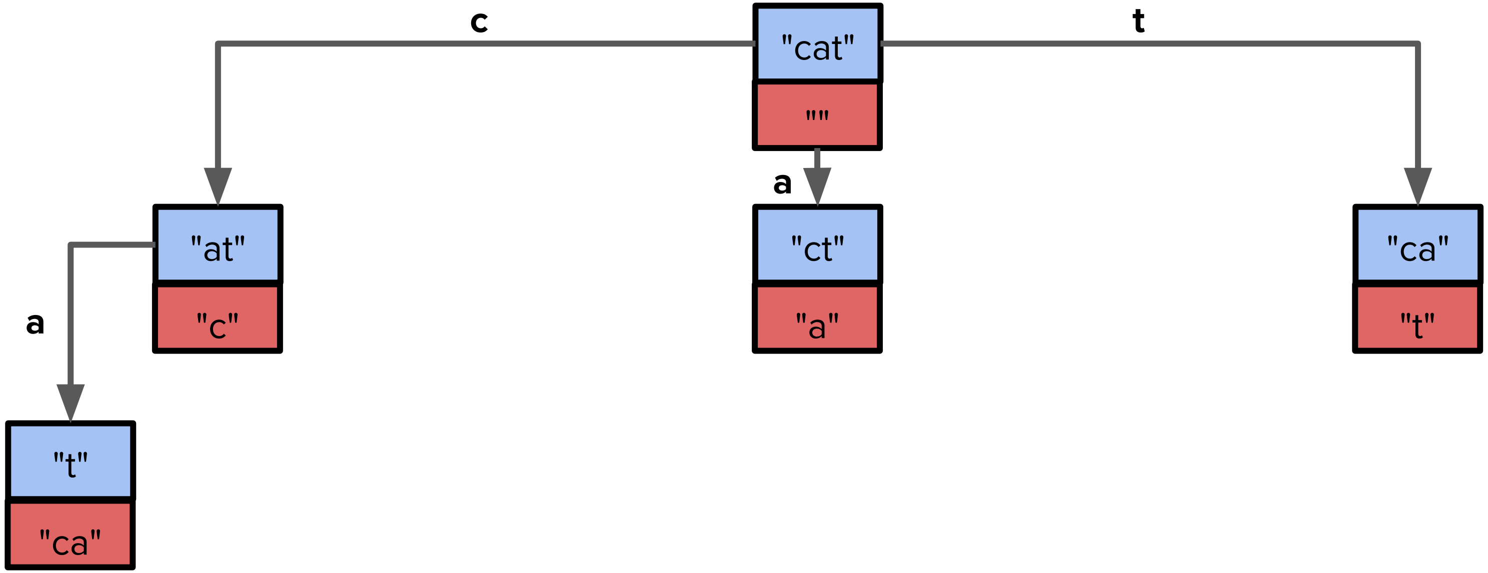
Decisions yet to be made
Decisions made so far

Decision tree: Find all permutations of "cat"



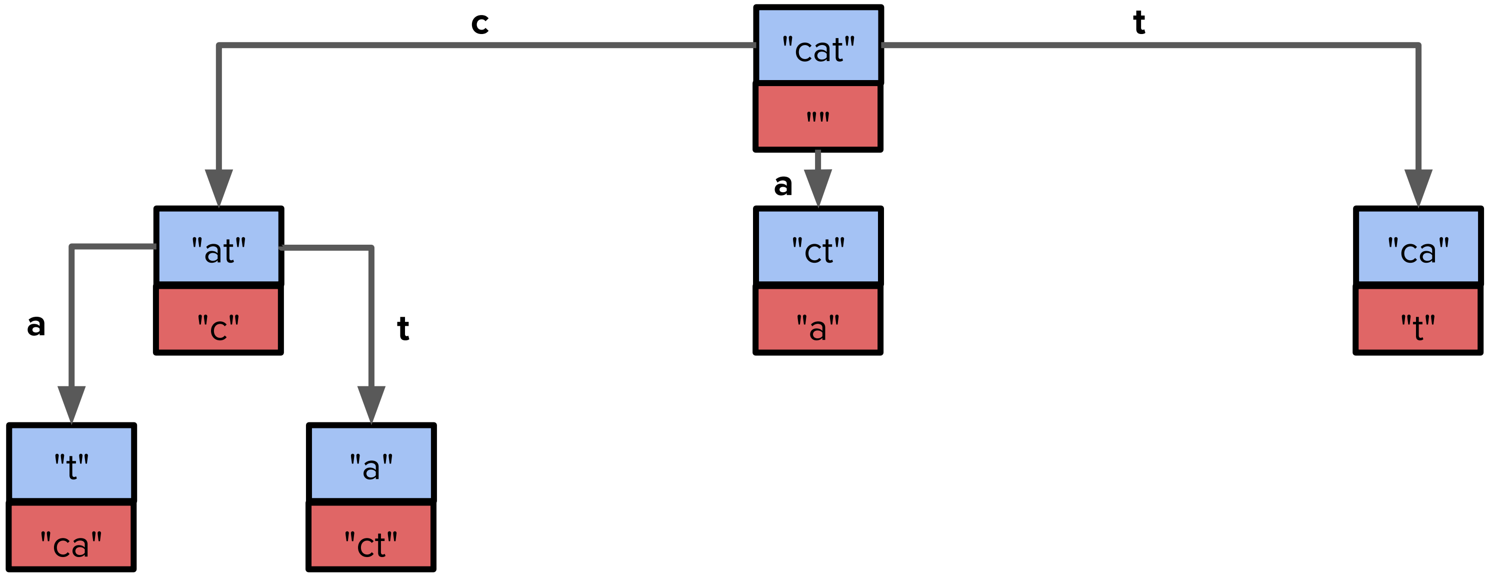
🟦 Decisions yet to be made
🟥 Decisions made so far

Decision tree: Find all permutations of "cat"



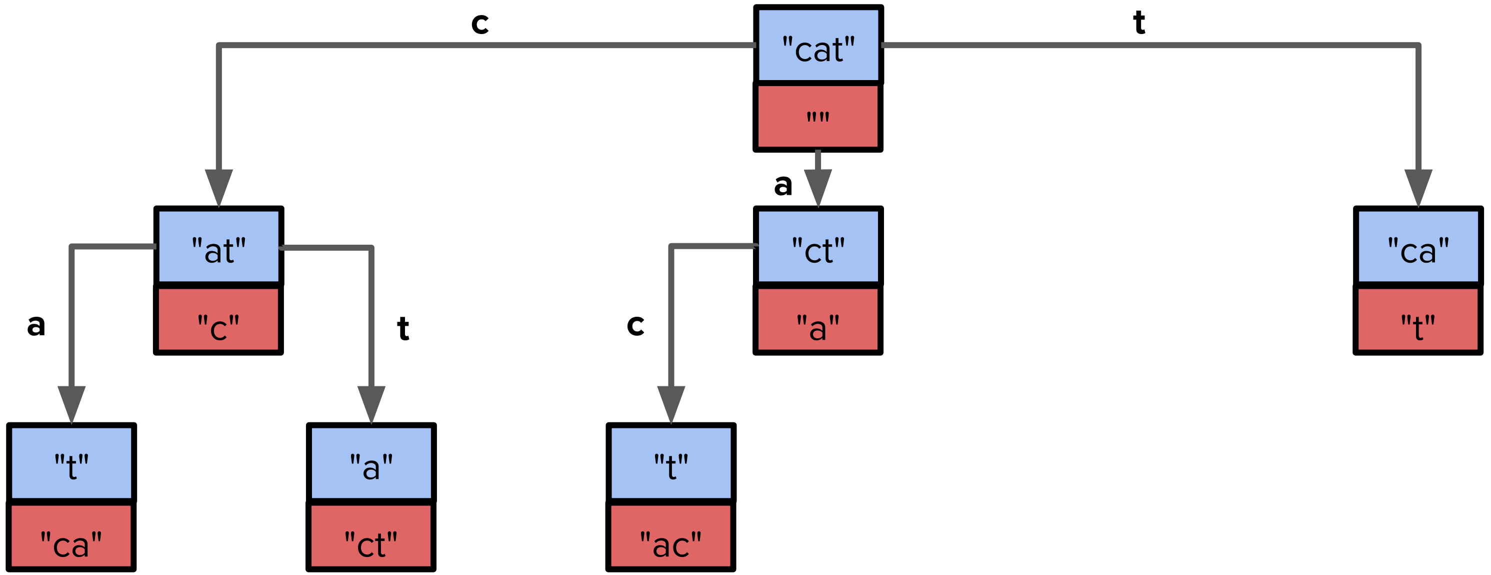
🟦 Decisions yet to be made
🟥 Decisions made so far

Decision tree: Find all permutations of "cat"



🟦 Decisions yet to be made
🟥 Decisions made so far

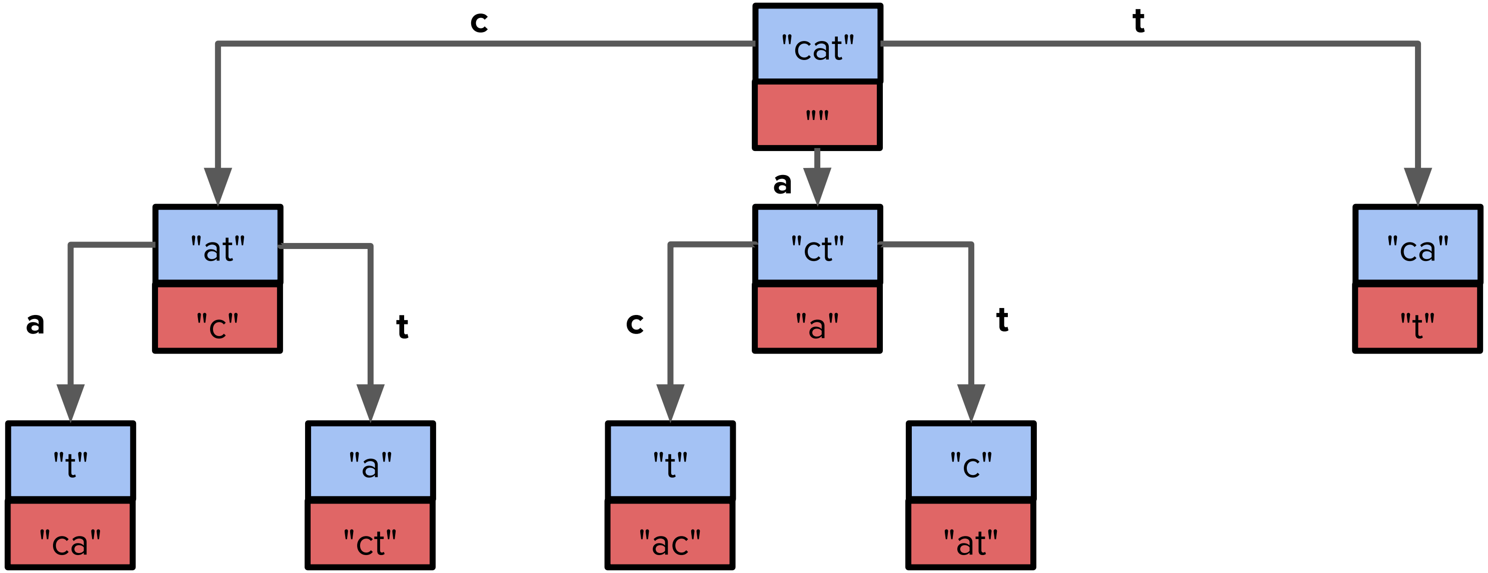
Decision tree: Find all permutations of "cat"



Decisions yet to be made

Decisions made so far

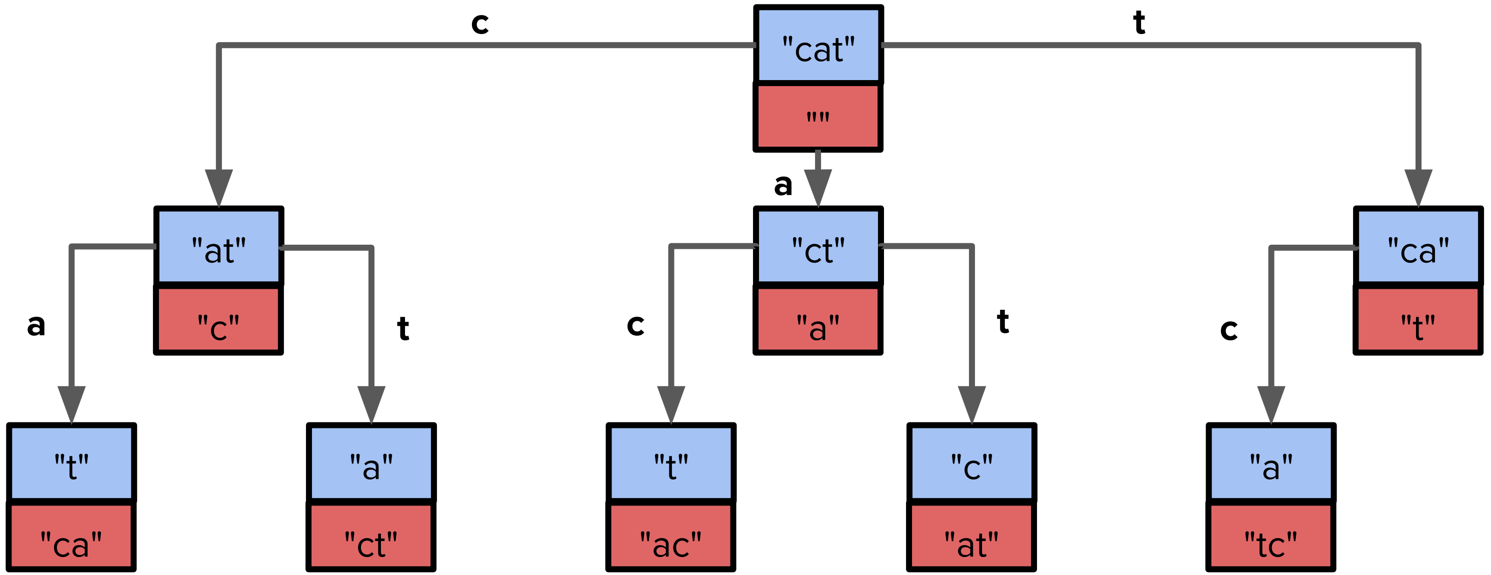
Decision tree: Find all permutations of "cat"



🟦 Decisions yet to be made

🟥 Decisions made so far

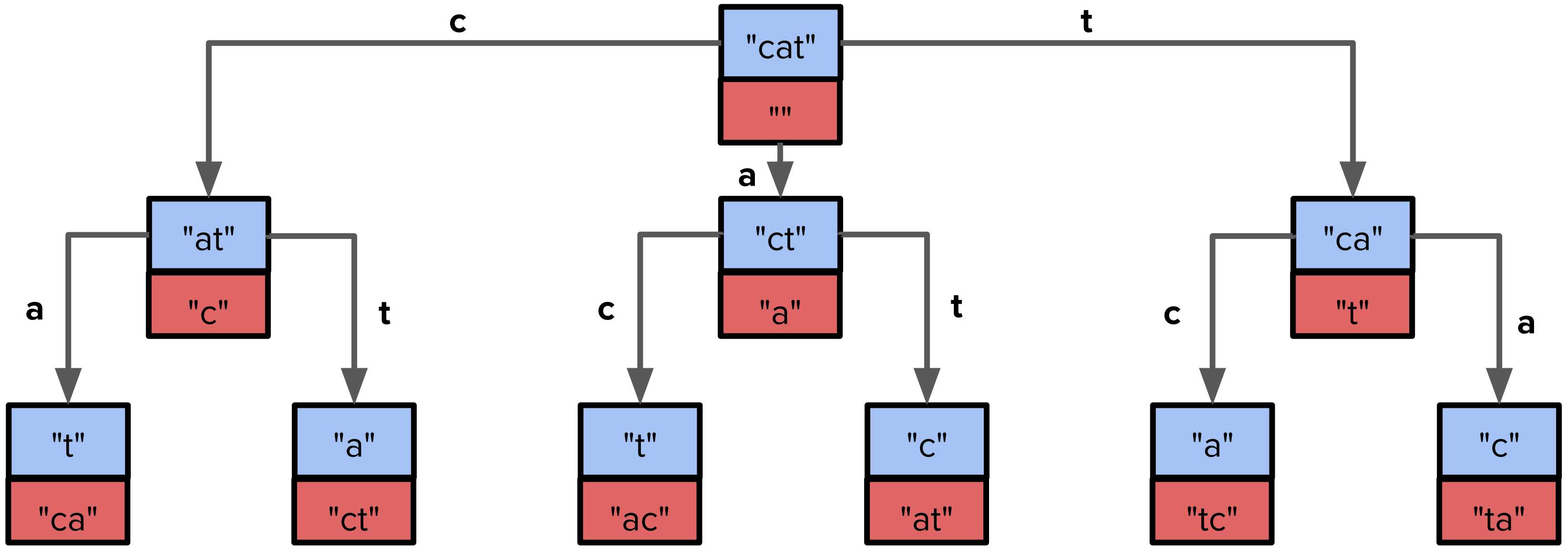
Decision tree: Find all permutations of "cat"



🟦 Decisions yet to be made

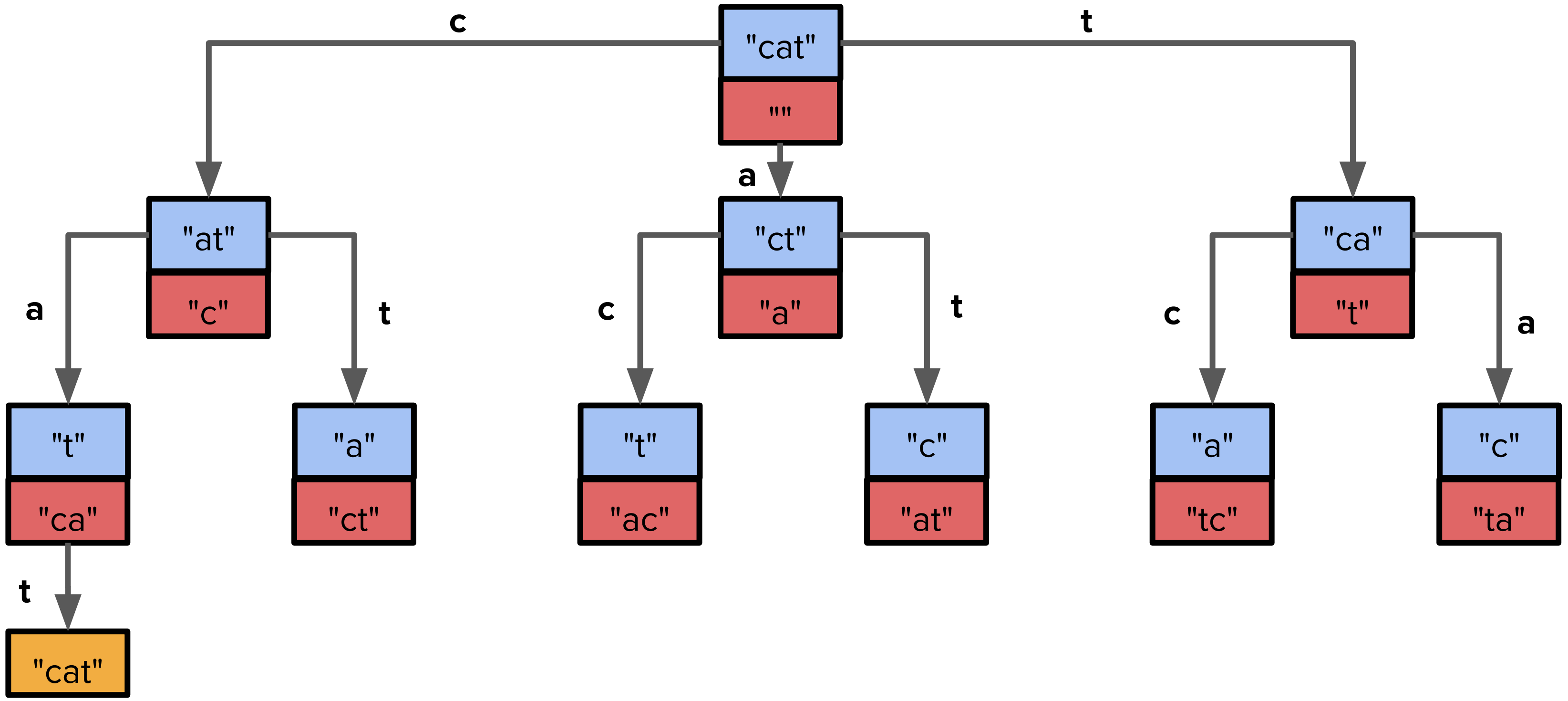
🟥 Decisions made so far

Decision tree: Find all permutations of "cat"



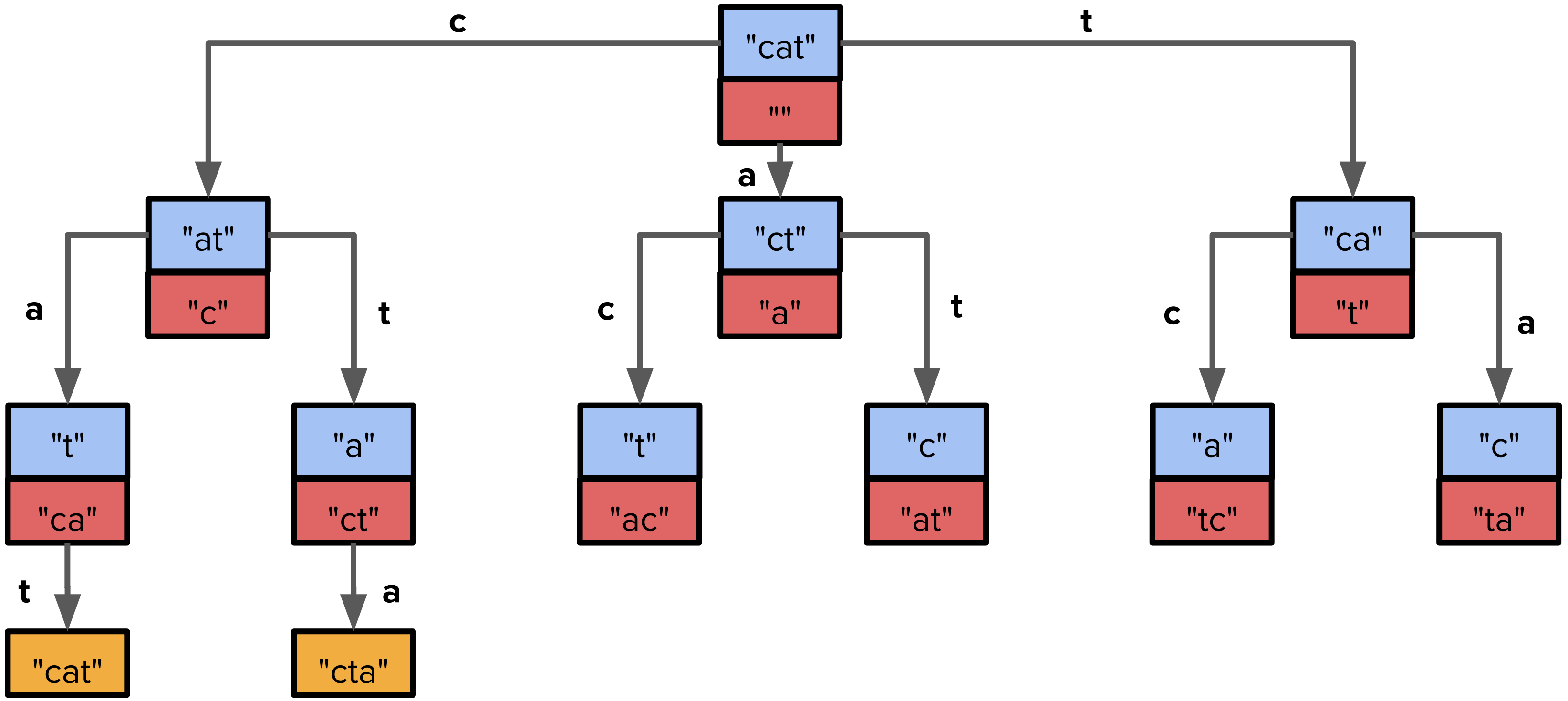
Decisions yet to be made
Decisions made so far

Decision tree: Find all permutations of "cat"



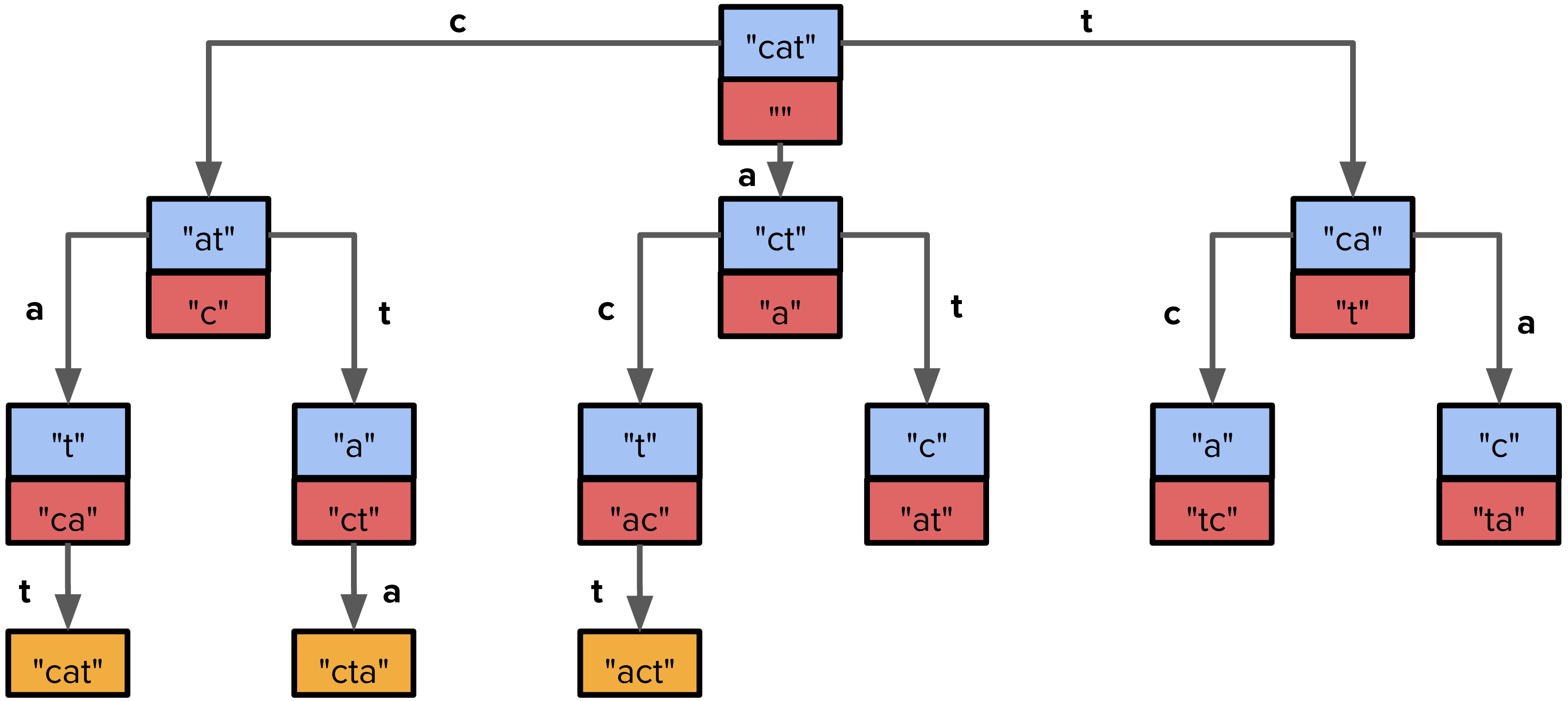
Decisions yet to be made
Decisions made so far

Decision tree: Find all permutations of "cat"



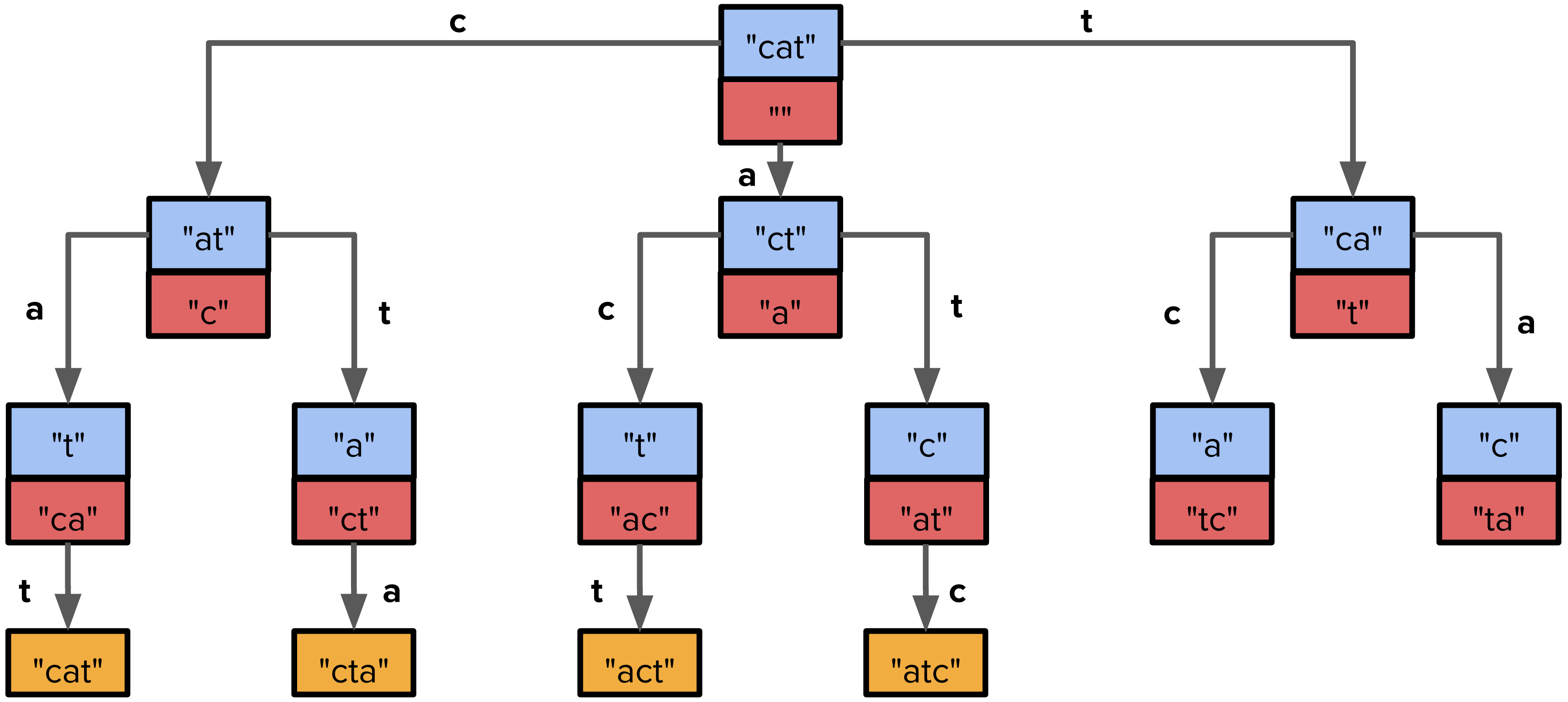
🟦 Decisions yet to be made
🟥 Decisions made so far

Decision tree: Find all permutations of "cat"



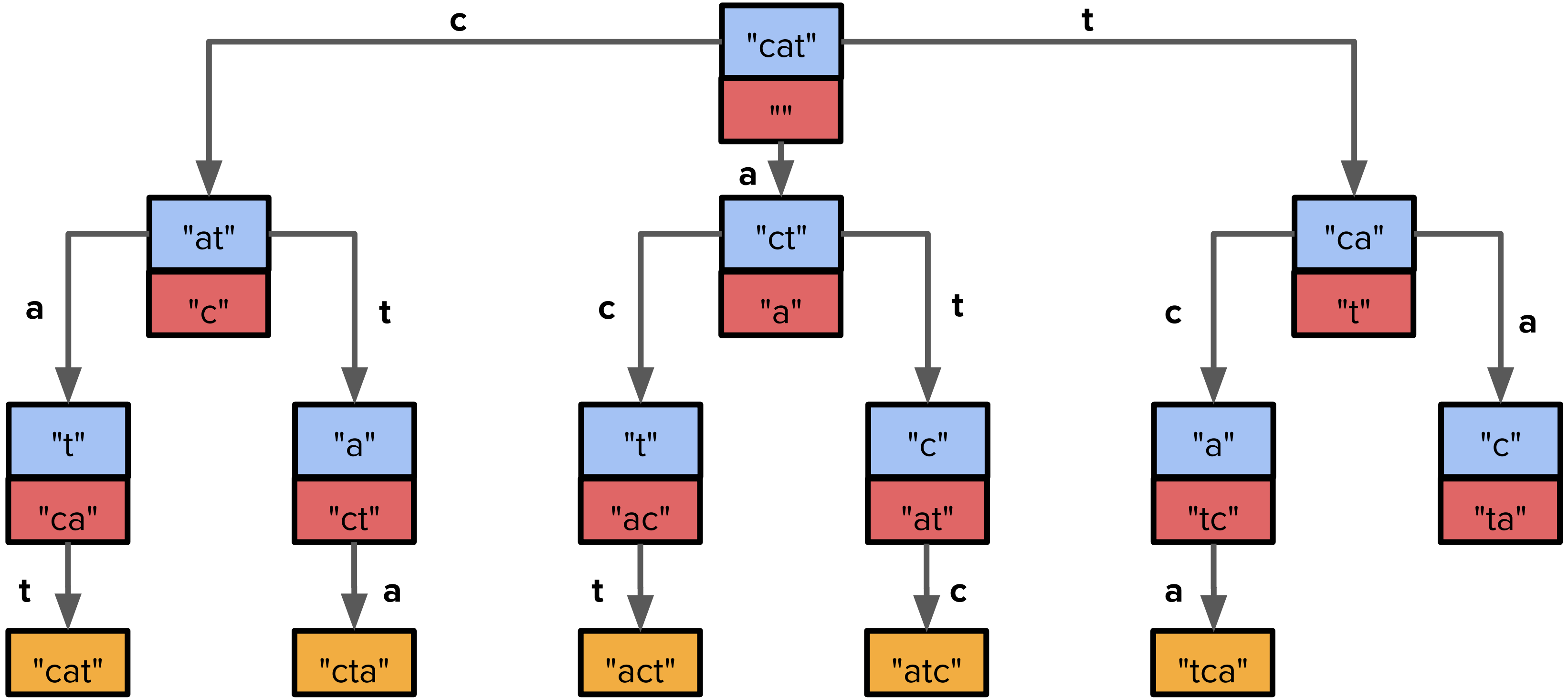
Decisions yet to be made
Decisions made so far

Decision tree: Find all permutations of "cat"



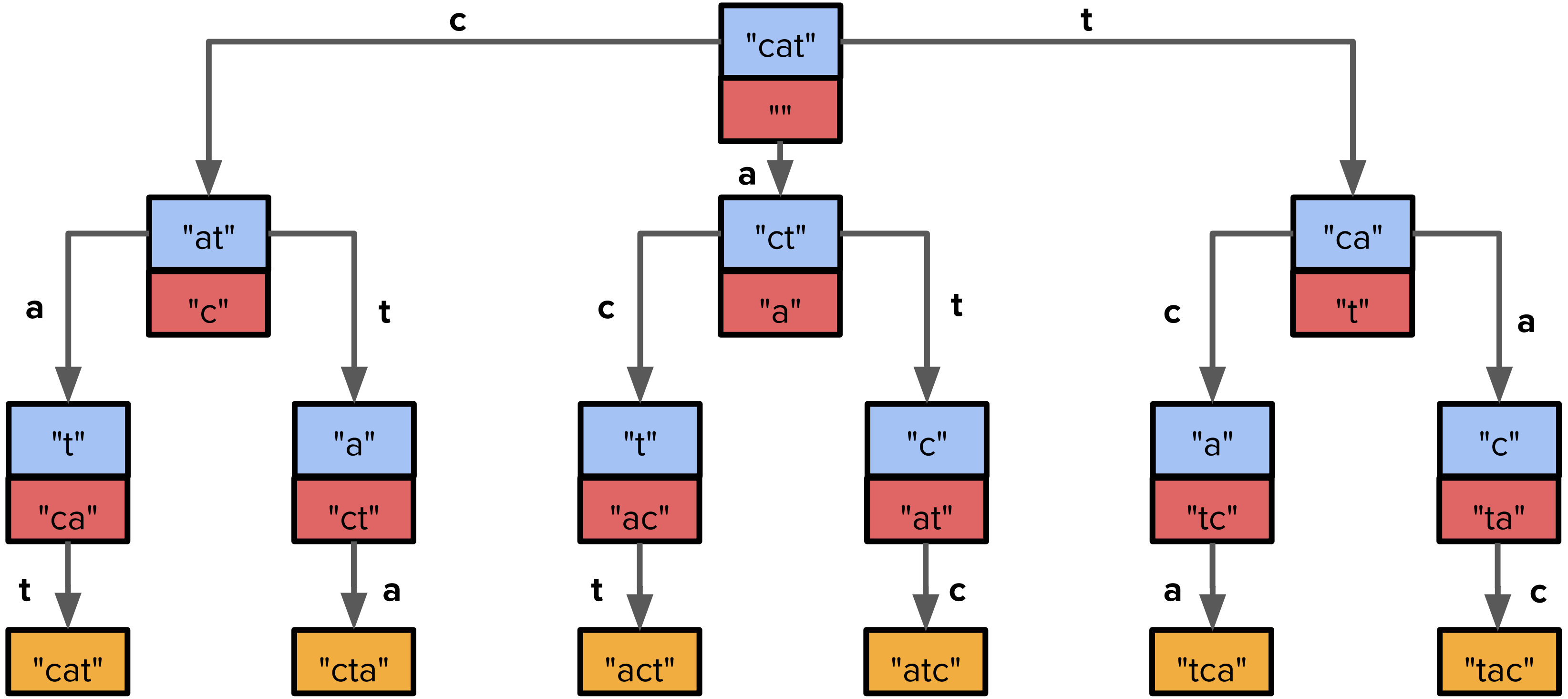
Decisions yet to be made
Decisions made so far

Decision tree: Find all permutations of "cat"



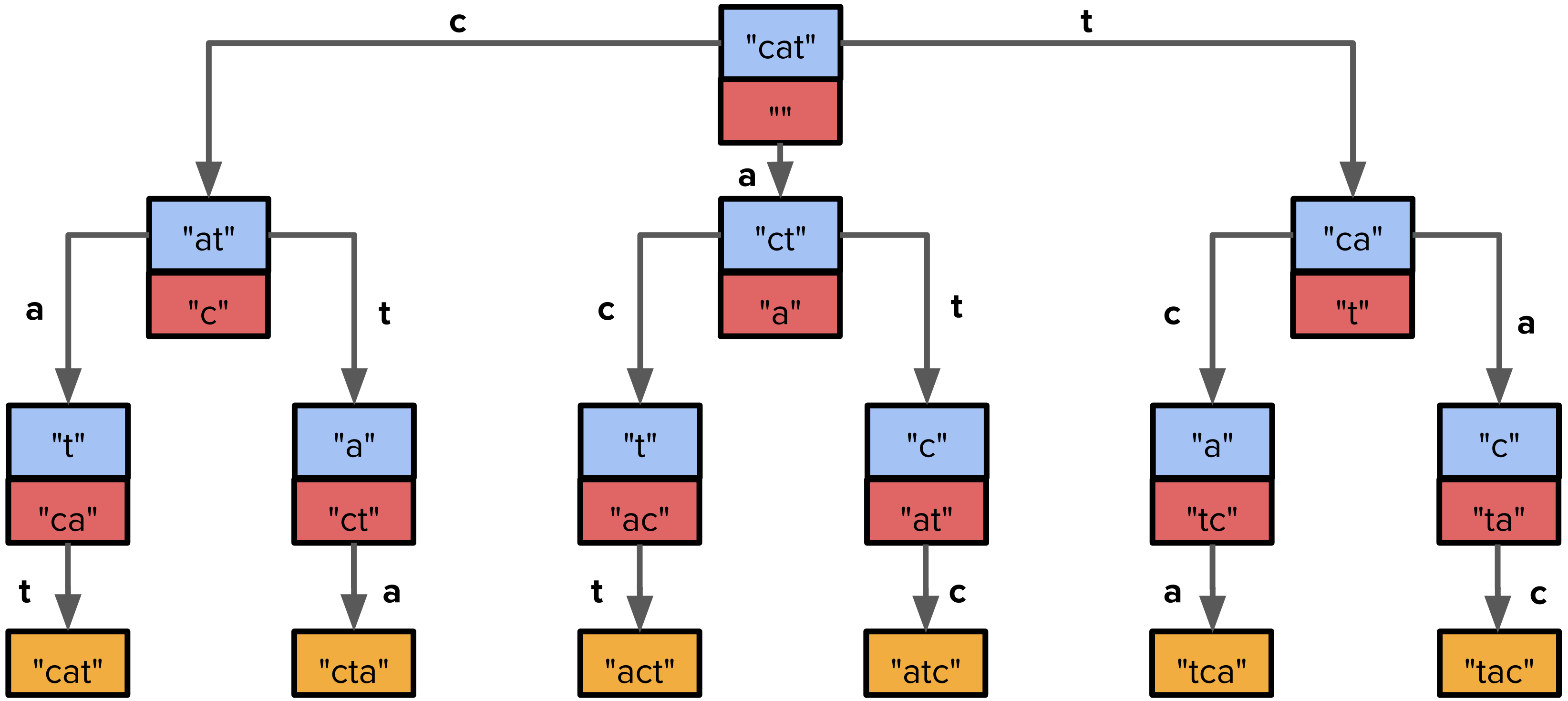
Decisions yet to be made
Decisions made so far

Decision tree: Find all permutations of "cat"



Decisions yet to be made
Decisions made so far

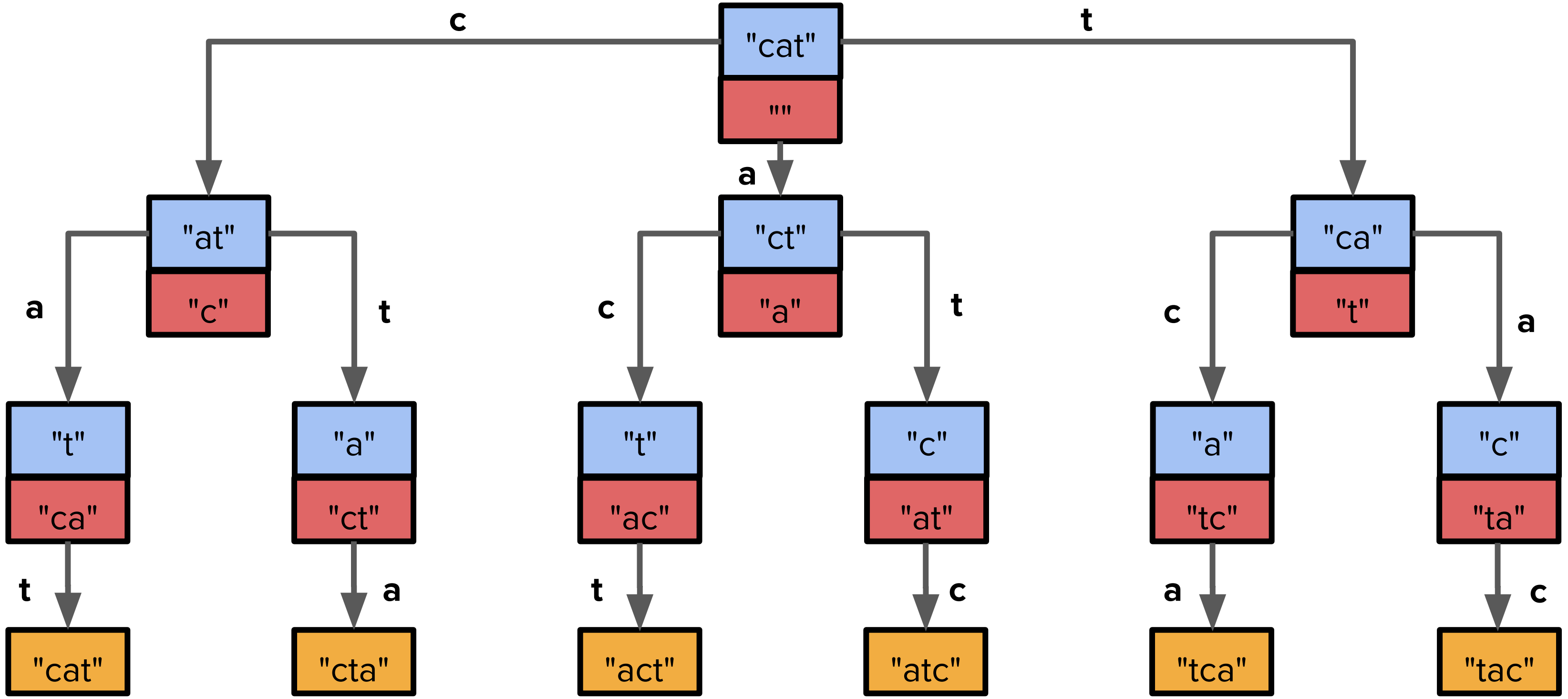
Decision tree: Find all permutations of "cat"



Base case: No letters remaining to choose!

Decisions yet to be made
Decisions made so far

Decision tree: Find all permutations of "cat"



Recursive case: For every letter remaining, add that letter to the current permutation and recurse!

Word scramble code

Permutations Code

```
void listPermutations(string s){
   listPermutationsHelper(s, "");
}

void listPermutationsHelper(string remaining, string soFar) {
   if (remaining.empty()) {
       cout << soFar << endl;
   } else {
       for (int i = 0; i < remaining.length(); i++) {
           char nextLetter = remaining[i];
           string rest = remaining.substr(0, i) + remaining.substr(i+1);
           listPermutationsHelper(rest, soFar + nextLetter);
       }
   }
}
```

Permutations Code

```
void listPermutations(string s){
   listPermutationsHelper(s, "");
}

void listPermutationsHelper(string remaining, string soFar) {
   if (remaining.empty()) {
       cout << soFar << endl;
   } else {
       for (int i = 0; i < remaining.length(); i++) {
           char nextLetter = remaining[i];
           string rest = remaining.substr(0, i) + remaining.substr(i+1);
           listPermutationsHelper(rest, soFar + nextLetter);
       }
   }
}
```

Use of recursive helper function with empty string as starting point

Permutations Code

Decisions yet to be made

```
void listPermutations(string s){
   listPermutationsHelper(s, "");
}

void listPermutationsHelper(string remaining, string soFar) {
   if (remaining.empty()) {
       cout << soFar << endl;
   } else {
       for (int i = 0; i < remaining.length(); i++) {
           char nextLetter = remaining[i];
           string rest = remaining.substr(0, i) + remaining.substr(i+1);
           listPermutationsHelper(rest, soFar + nextLetter);
       }
   }
}
```

Permutations Code

Decisions yet to be made

Decisions already made

```
void listPermutations(string s){
   listPermutationsHelper(s, "");
}

void listPermutationsHelper(string remaining, string soFar) {
   if (remaining.empty()) {
       cout << soFar << endl;
   } else {
       for (int i = 0; i < remaining.length(); i++) {
           char nextLetter = remaining[i];
           string rest = remaining.substr(0, i) + remaining.substr(i+1);
           listPermutationsHelper(rest, soFar + nextLetter);
       }
   }
}
```

Permutations Code

```
void listPermutations(string s){
   listPermutationsHelper(s, "");
}

void listPermutationsHelper(string remaining, string soFar) {
   if (remaining.empty()) {
       cout << soFar << endl;
   } else {
       for (int i = 0; i < remaining.length(); i++) {
           char nextLetter = remaining[i];
           string rest = remaining.substr(0, i) + remaining.substr(i+1);
           listPermutationsHelper(rest, soFar + nextLetter);
       }
   }
}
```

Decisions yet to be made

Decisions already made

Base case: No decisions remain

Permutations Code

```
void listPermutations(string s){
   listPermutationsHelper(s, "");
}

void listPermutationsHelper(string remaining, string soFar) {
   if (remaining.empty()) {
       cout << soFar << endl;
   } else {
       for (int i = 0; i < remaining.length(); i++) {
           char nextLetter = remaining[i];
           string rest = remaining.substr(0, i) + remaining.substr(i+1);
           listPermutationsHelper(rest, soFar + nextLetter);
       }
   }
}
```

Decisions yet to be made

Decisions already made

Base case: No decisions remain

Recursive case: Try all options for next decision

Takeaways

- The specific model of the general **"choose / explore / unchoose"** pattern in backtracking recursion that we applied here can be thought of as **"copy, edit, recurse"**
  - Since we passed all our parameters by value, each recursive stack frame had its own independent copy of the string data that it could edit as appropriate
  - The "unchoose" step is **implicit** since there is no need to undo anything by virtue of the fact that editing a copy only has local consequences.

Takeaways

- The specific model of the general **"choose / explore / unchoose"** pattern in backtracking recursion that we applied here can be thought of as **"copy, edit, recurse"**

- At each step of the recursive backtracking process, it is important to keep track of **the decisions we've made so far** and **the decisions we have left to make**

Takeaways

- The specific model of the general **"choose / explore / unchoose"** pattern in backtracking recursion that we applied here can be thought of as **"copy, edit, recurse"**
- At each step of the recursive backtracking process, it is important to keep track of **the decisions we've made so far** and **the decisions we have left to make**
- Backtracking recursion can have **variable branching factors** at each level

Takeaways

- The specific model of the general **"choose / explore / unchoose"** pattern in backtracking recursion that we applied here can be thought of as **"copy, edit, recurse"**

- At each step of the recursive backtracking process, it is important to keep track of **the decisions we've made so far** and **the decisions we have left to make**

- Backtracking recursion can have **variable branching factors** at each level

- Use of helper functions and initial empty params that get built up is common

Goals for this Course

Learn how to model and solve complex problems with computers.

- Explore common abstractions for representing problems.
- Harness recursion and understand how to think about problems recursively.
- Analyze different approaches for solving problems: efficiency, optimization, and ethics.

Two types of recursion

Basic recursion

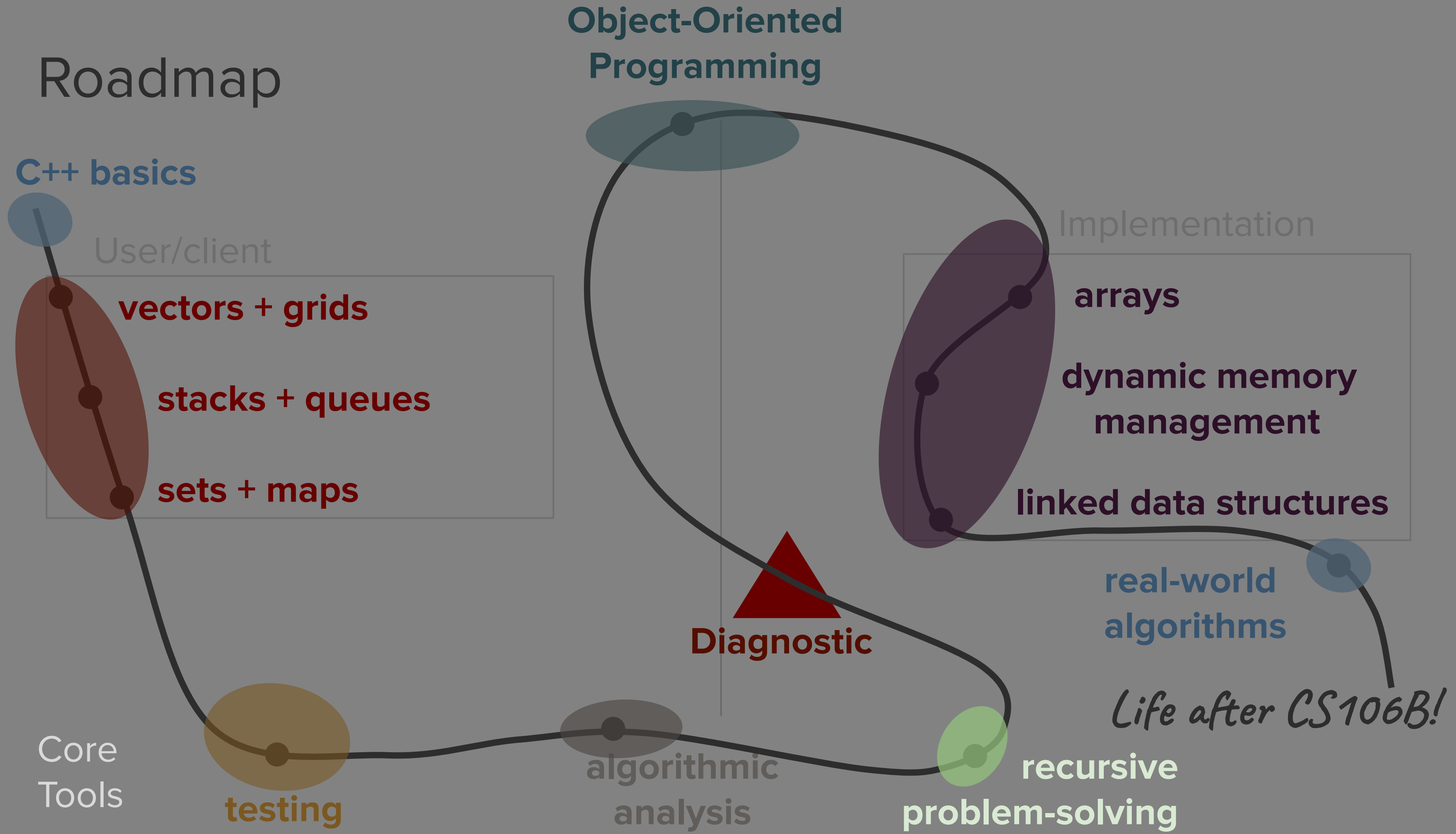
- One repeated task that builds up a solution as you come back up the call stack
- The final base case defines the initial seed of the solution and each call contributes a little bit to the solution
- Initial call to recursive function produces final solution

Backtracking recursion

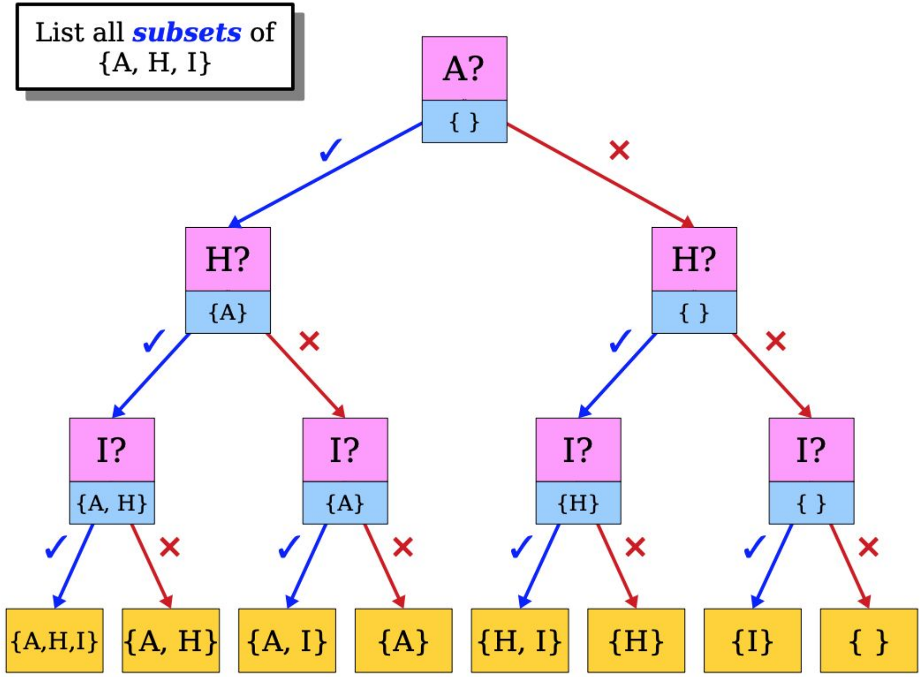
- Build up many possible solutions through multiple recursive calls at each step
- Seed the initial recursive call with an “empty” solution
- At each base case, you have a potential solution

What’s next?

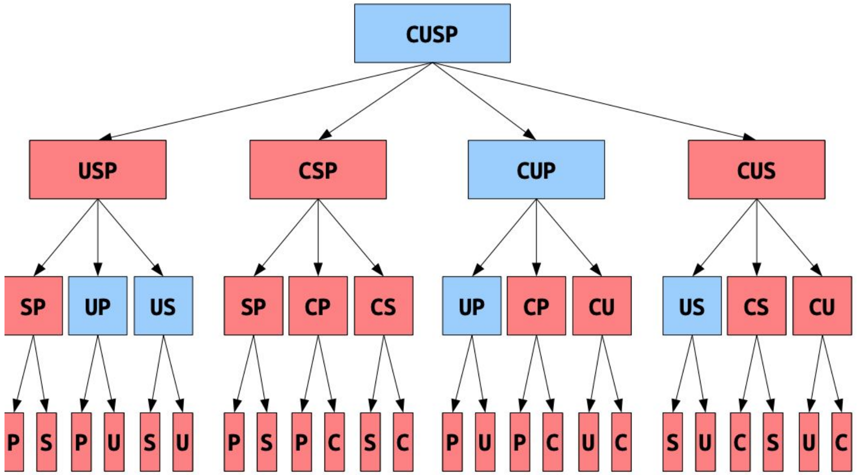
Roadmap


Object-Oriented Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

Recursive Backtracking



List all *subsets* of {A, H, I}
A?
{ }
✓
✗
H?
{A}
H?
{ }
I?
{A, H}
I?
{A}
I?
{H}
I?
{ }
{A,H,I}
{A, H}
{A, I}
{A}
{H, I}
{H}
{I}
{ }



CUSP
USP
CSP
CUP
CUS
SP
UP
US
SP
CP
CS
UP
CP
CU
US
CS
CU
P
S
P
U
S
U
P
S
P
C
S
C
P
U
P
C
U
C
S
U
C
S
U
C