Recursive Backtracking 1

Elyse Cornwall

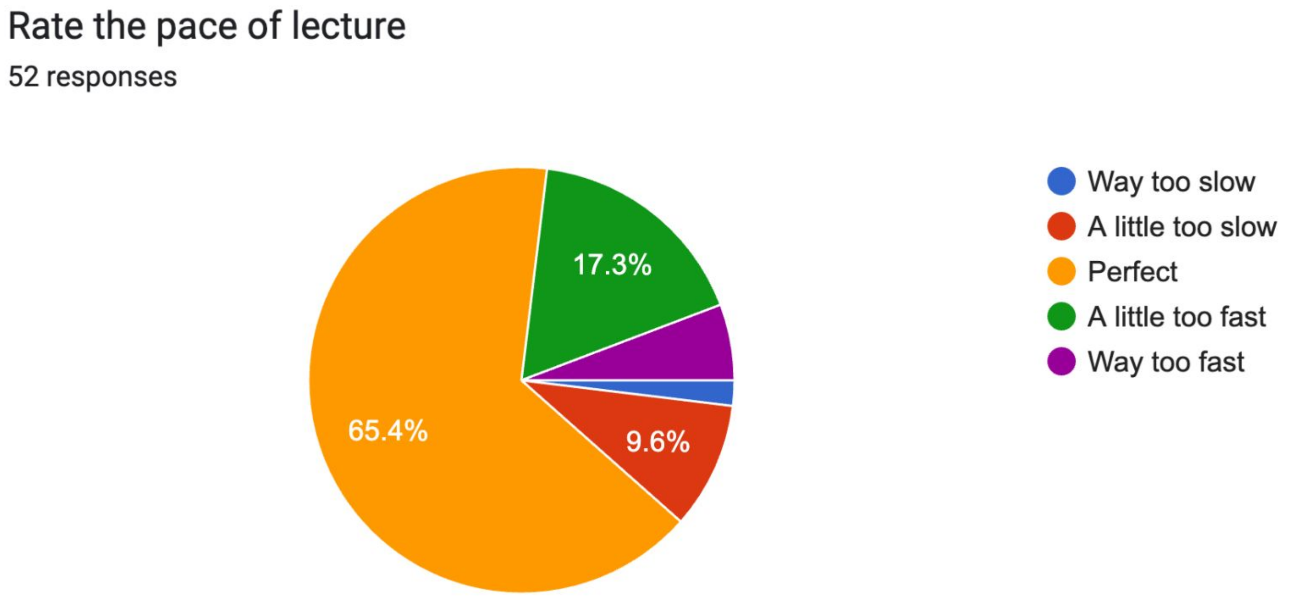
July 18th, 2023

Contributions made from previous CS106B Instructors

Announcements

- Exam is confidential through the end of the week
  - No discussing the exam with other students or posting about it on Ed
  - Grades will be released over the weekend
- Second part of Assignment 3 will be released Wednesday
  - Part 2 will cover recursive backtracking, this week's topic!
  - Assignment 3 YEAH hours 3pm on Wednesday

Week 3 Feedback



Rate the pace of lecture

52 responses

Week 3 Feedback

Things you liked:

“I like when you go over our **feedback**.” (meta 🤯)

“I like how you guys do **whiteboarding** explanations and went through more **examples** in class.”

“loved **exam review**”

“I really like the **stop and code** parts of the lecture, as it helps us think and learn on the spot!”

Week 3 Feedback

Places we can improve:

“Maybe more practice examples on unwrapping files”

“I wish instead of talking to the person next to us we just got to think.”

“Look at different approaches to solving the example questions and analyse them (which is better/worse)”

“I wish we had more opportunities to try to write code in class.”

Week 3 Feedback

We hear you…

“More review sessions like the midterm one.” **Friday Review Sessions!**

“LAIR during Fridays”

“Giving us a rubric for the midterm would be helpful”

“Sections are really good for learning. Aside from section being longer, I don't think anything could be improved.”

“Would it be too big of an ask to as that CS106B changes the weekly assignment submission date?” **Coincidentally, assignment deadlines will be on Wednesdays for the rest of the quarter.**

Week 3 Feedback

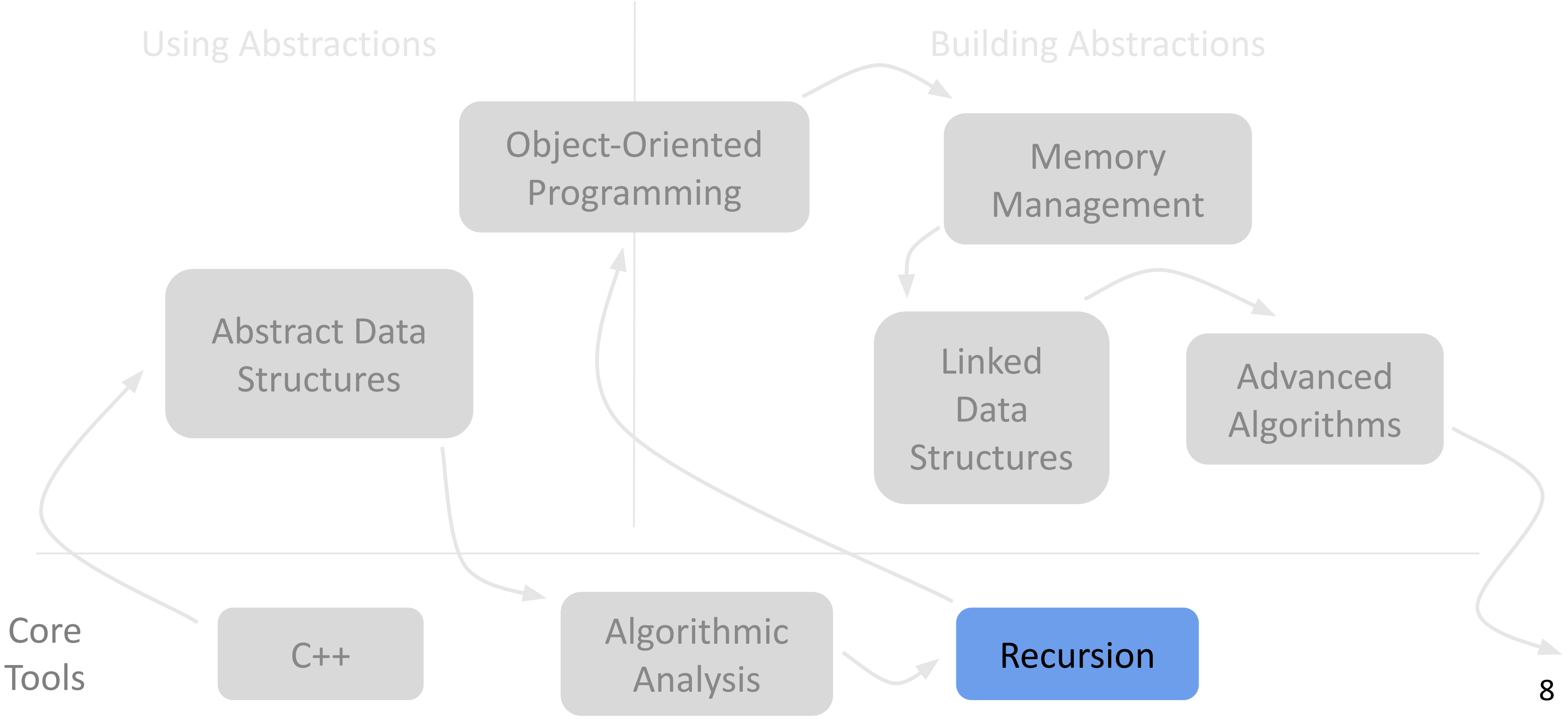
Anything else you would like us to know:

“Recursive code truly works like magic”

“Keep it up!” 😎

Roadmap

Even more recursion…



Using Abstractions

Building Abstractions

Object-Oriented Programming

Memory Management

Abstract Data Structures

Linked Data Structures

Advanced Algorithms

Core Tools

C++

Algorithmic Analysis

Recursion

Recursion Recap

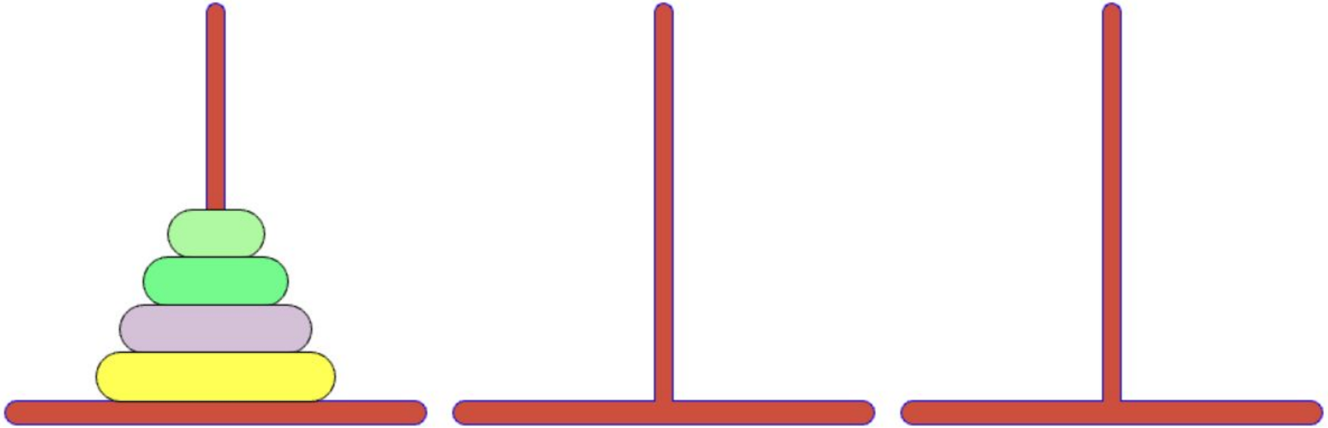
Why do we use recursion?

Why do we use recursion?

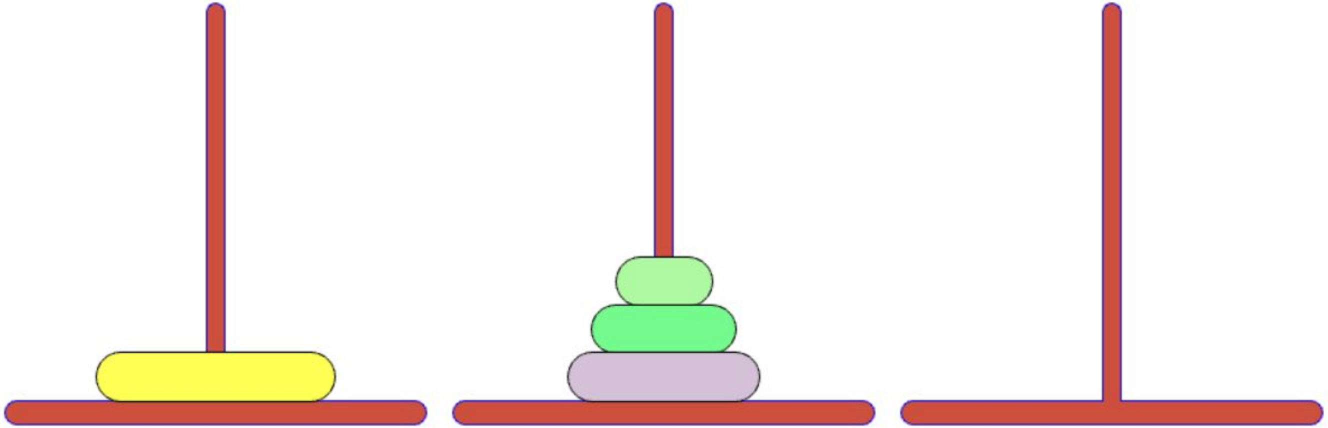
- Elegant
  - Some problems have beautiful, concise recursive solutions
- Efficient
  - Recursive solutions can have faster runtimes
- Dynamic
  - We’ll explore **recursive backtracking** TODAY

An *elegant* solution: Tower of Hanoi

Solving with 4 Disks

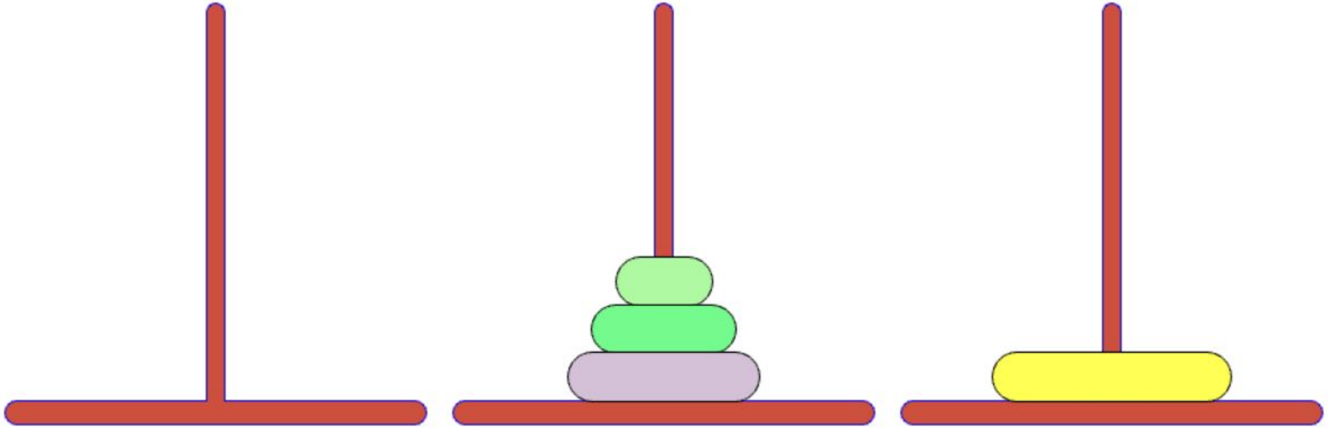


Solving with 4 Disks



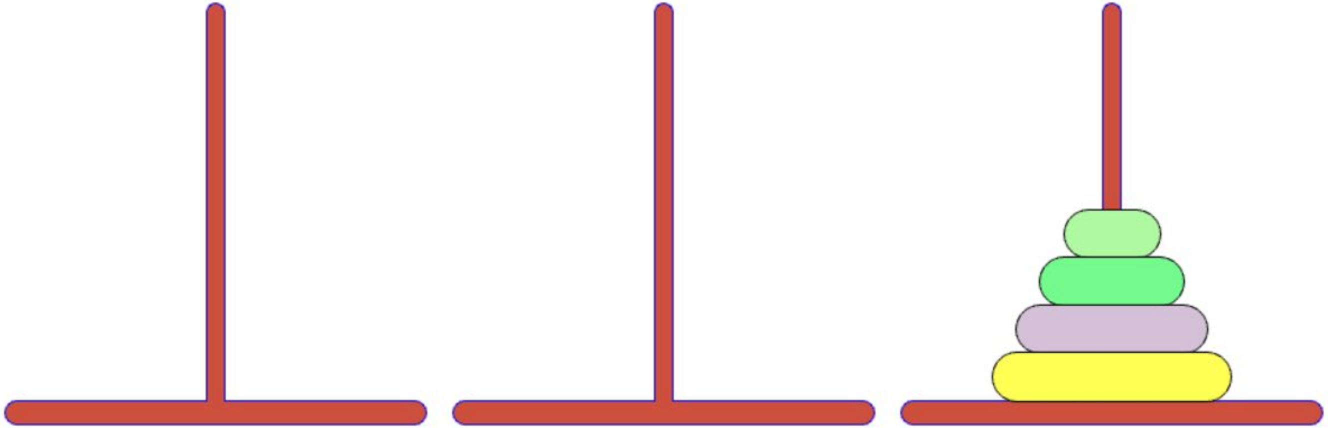
We’ll need to get the smaller 3 disks out of the way,

Solving with 4 Disks



Move the bottom piece over…

Solving with 4 Disks



Then stack the 3 smaller disks on top.

Solving with N Disks

1. Move tower of N-1 disks onto middle peg
2. Move Nth disk over
3. Move tower of N-1 disks onto end peg

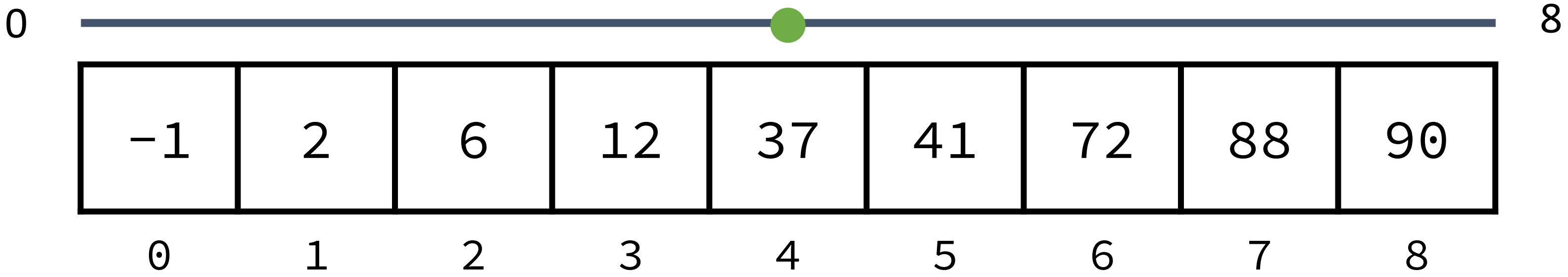
Solution

```
void solveTowers(int n, char start, char end, char aux) {
    if (n == 0) {
        return;
    }
    solveTowers(n-1, start, aux, end);
    moveSingleDisk(start, end);
    solveTowers(n-1, aux, end, start);
}
```

An *efficient* solution: Binary Search

Binary Search

- Let’s say we have a sorted Vector of integers
- Can we use the same algorithm as before to look up a number?



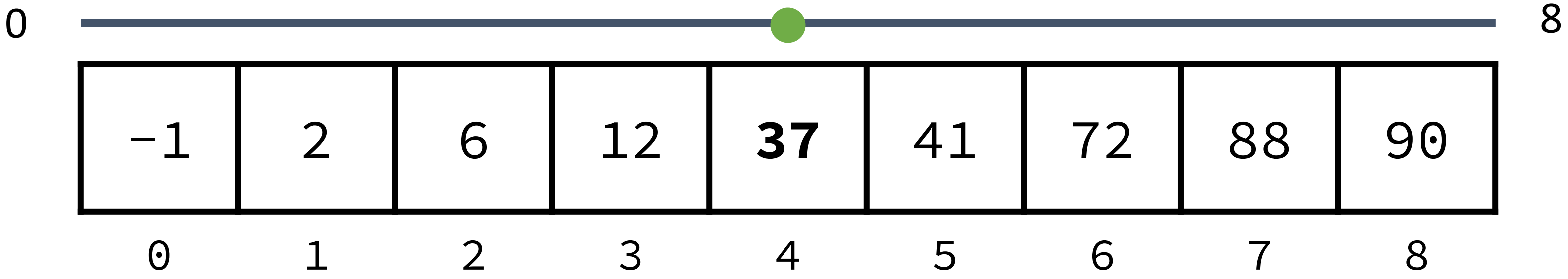
-1	2	6	12	37	41	72	88	90
0	1	2	3	4	5	6	7	8

Let’s try to find the number 6 in our Vector

Binary Search

Let's try to find the number 6

- Let's say we have a sorted Vector of integers
- Can we use the same algorithm as before to look up a number?



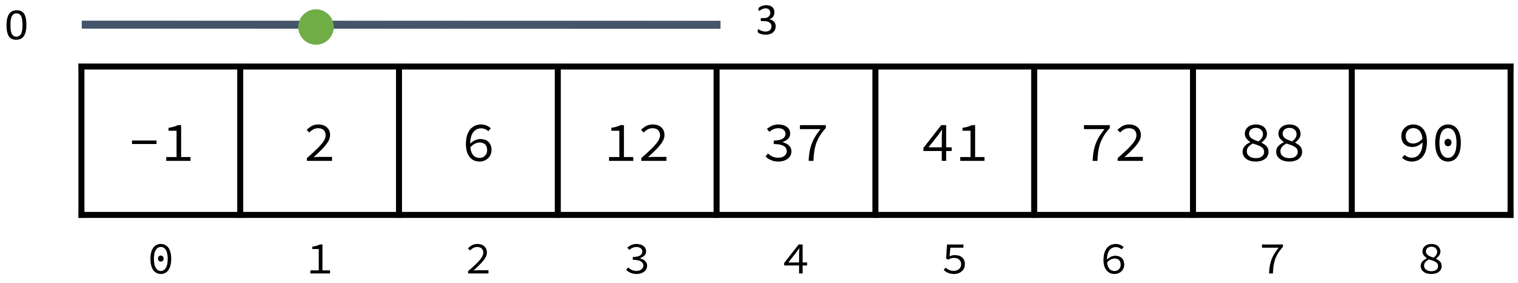
-1	2	6	12	**37**	41	72	88	90
0	1	2	3	4	5	6	7	8

Too big, look left

Binary Search

Let’s try to find the number 6

- Let’s say we have a sorted Vector of integers
- Can we use the same algorithm as before to look up a number?

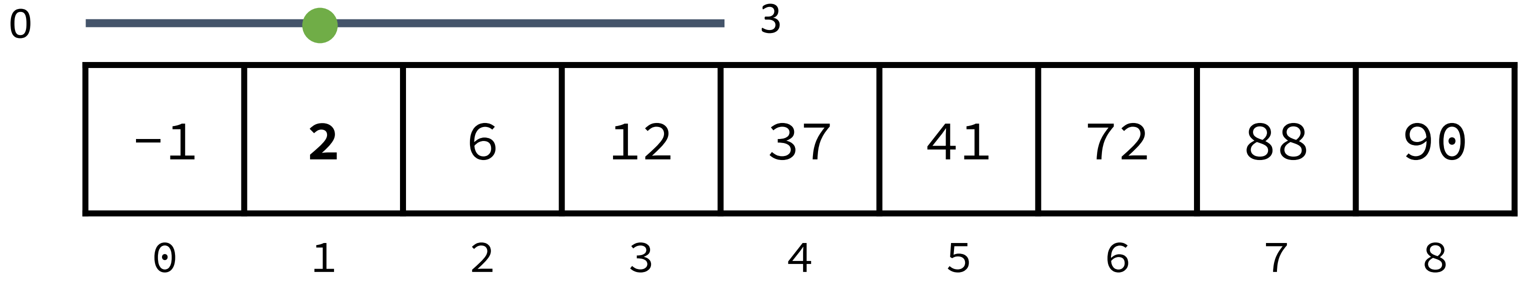


0	1	2	3	4	5	6	7	8
-1	2	6	12	37	41	72	88	90

Binary Search

Let’s try to find the number 6

- Let’s say we have a sorted Vector of integers
- Can we use the same algorithm as before to look up a number?



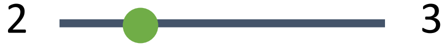
-1	**2**	6	12	37	41	72	88	90
0	1	2	3	4	5	6	7	8

Too small, look right

Binary Search

Let’s try to find the number 6

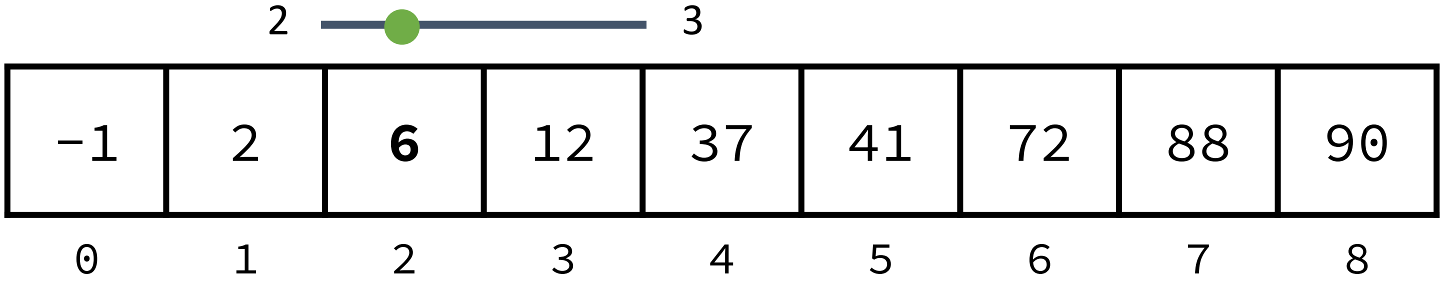
- Let’s say we have a sorted Vector of integers
- Can we use the same algorithm as before to look up a number?



-1	2	6	12	37	41	72	88	90
0	1	2	3	4	5	6	7	8

Binary Search

- Let’s say we have a sorted Vector of integers
- Can we use the same algorithm as before to look up a number?



-1	2	**6**	12	37	41	72	88	90
0	1	2	3	4	5	6	7	8

Found it! 🎉🎉🎉

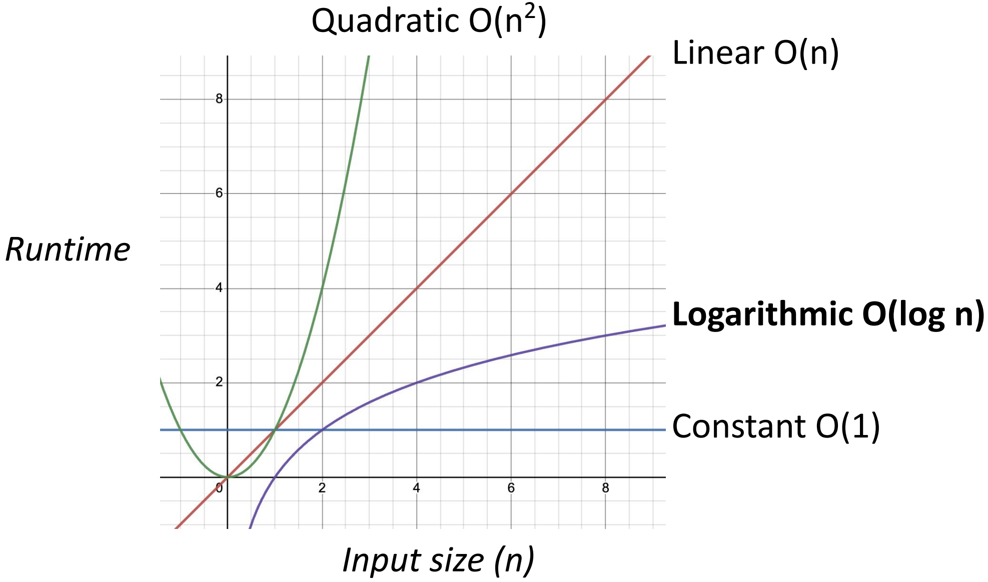
Binary Search as a Recursive Process

Binary search over some range of sorted elements:

1. Choose element in the middle of the range
2. If this element is our target, success!
3. If element is less than our target, do **binary search** to the right
4. If element is greater than our target, do **binary search** to the left

Runtime of Binary Search

- Binary search has runtime O(log n)
  - Common runtime for algorithms that halve search space at every step



Recursive Backtracking

The Limits of Iteration

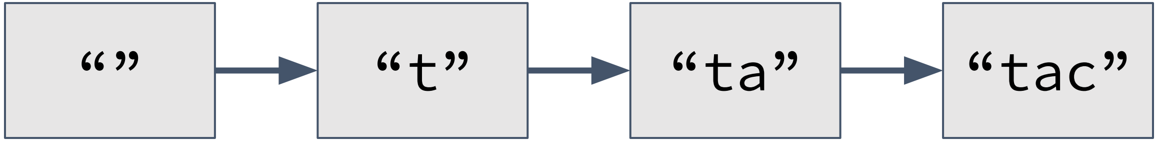
- We’ve seen how problems can be solved iteratively or recursively
  - The approach we chose was mostly a stylistic choice

The Limits of Iteration

- We’ve seen how problems can be solved iteratively or recursively
  - The approach we chose was mostly a stylistic choice
- However, some problems are nearly impossible to solve without recursion!
  - Iterative approaches are inherently *linear*: each step builds upon the next moving from the start to the end of your solution
  - Recursion allows us to explore many possible solutions by branching into multiple recursive calls

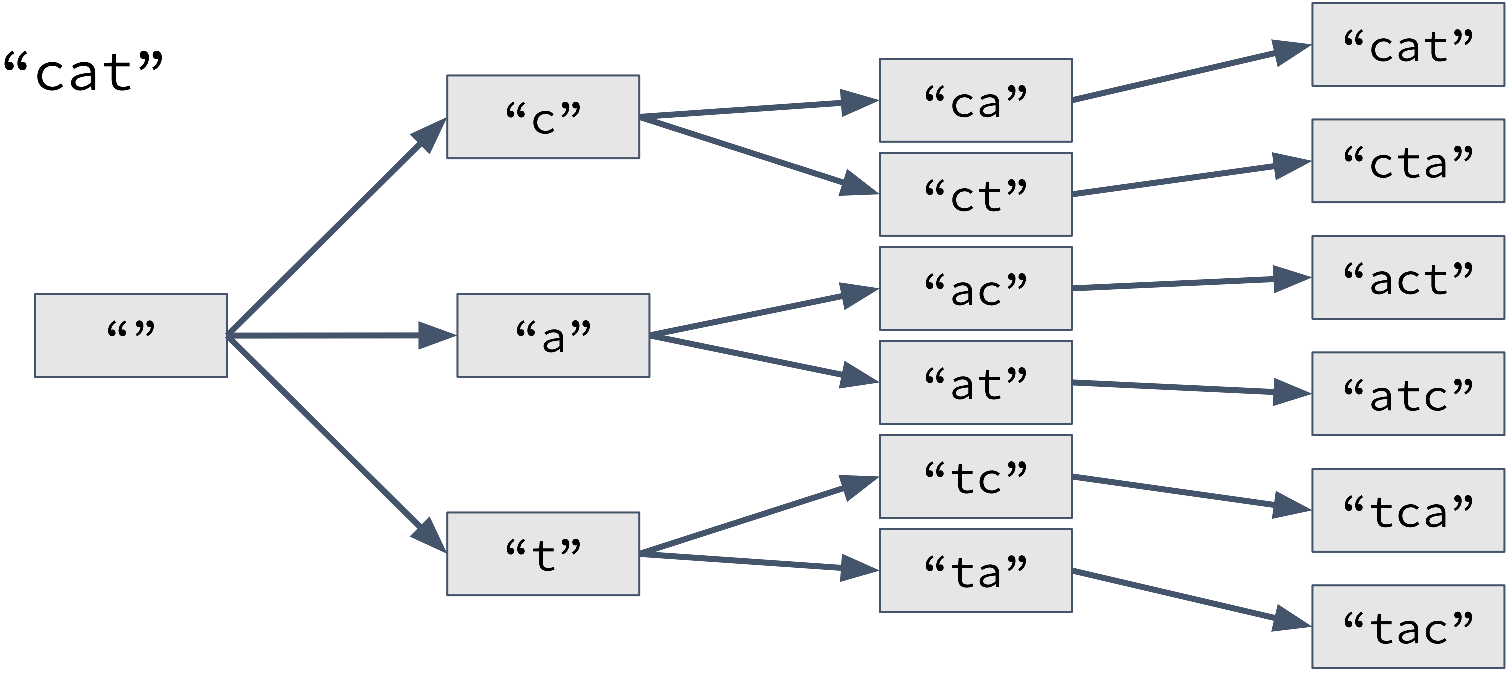
A Linear Program: Reverse a String

“cat”



An iterative approach works well; we can do this with a loop!

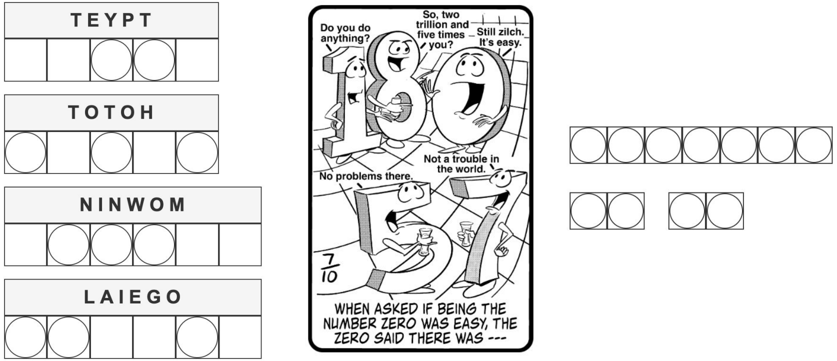
A Branching Program: Permute a String



Can we do this iteratively?

Not Really…

```
void permute5(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 or k == i) {
                    continue; // ignore
                }
                for (int w = 0; w < 4; w++) {
                    if (w == k or w == j or w == i) {
                        continue; // ignore
                    }
                    for (int x = 0; x < 5; x++) {
                        if (x == k or x == j or x == i or x == w) {
                            continue;
                        }
                        cout << "  " << s[i] << s[j] << s[k] << s[w] << s[x] << endl;
                    }
                }
            }
        }
    }
}
```



TEYPT
TOTOH
NINWOM
LAIEGO
Do you do anything?
So, two trillion and five times you?
Still zilch. It's easy.
Not a trouble in the world.
No problems there.
7/10
WHEN ASKED IF BEING THE NUMBER ZERO WAS EASY, THE ZERO SAID THERE WAS ---

A *dynamic* solution: Coin Sequences

Coin Sequences

- You’re playing a (rather boring) game in which you flip some number of coins one by one and see whether you get heads or tails
- You’d like to know all of the possible sequences you might flip

Coin Sequences

- You’re playing a (rather boring) game in which you flip some number of coins one by one and see whether you get heads or tails
- You’d like to know all of the possible sequences you might flip

🤔 *What are all of the possible sequences when flipping 2 coins?*

Coin Sequences

- You’re playing a (rather boring) game in which you flip some number of coins one by one and see whether you get heads or tails
- You’d like to know all of the possible sequences you might flip

HH HT TH TT

Coin Sequences

- You’re playing a (rather boring) game in which you flip some number of coins one by one and see whether you get heads or tails
- You’d like to know all of the possible sequences you might flip

How about for 3 coins?

Coin Sequences

- You're playing a (rather boring) game in which you flip some number of coins one by one and see whether you get heads or tails
- You'd like to know all of the possible sequences you might flip

HHH HHT HTH HTT THH THT TTH TTT

Coin Sequences

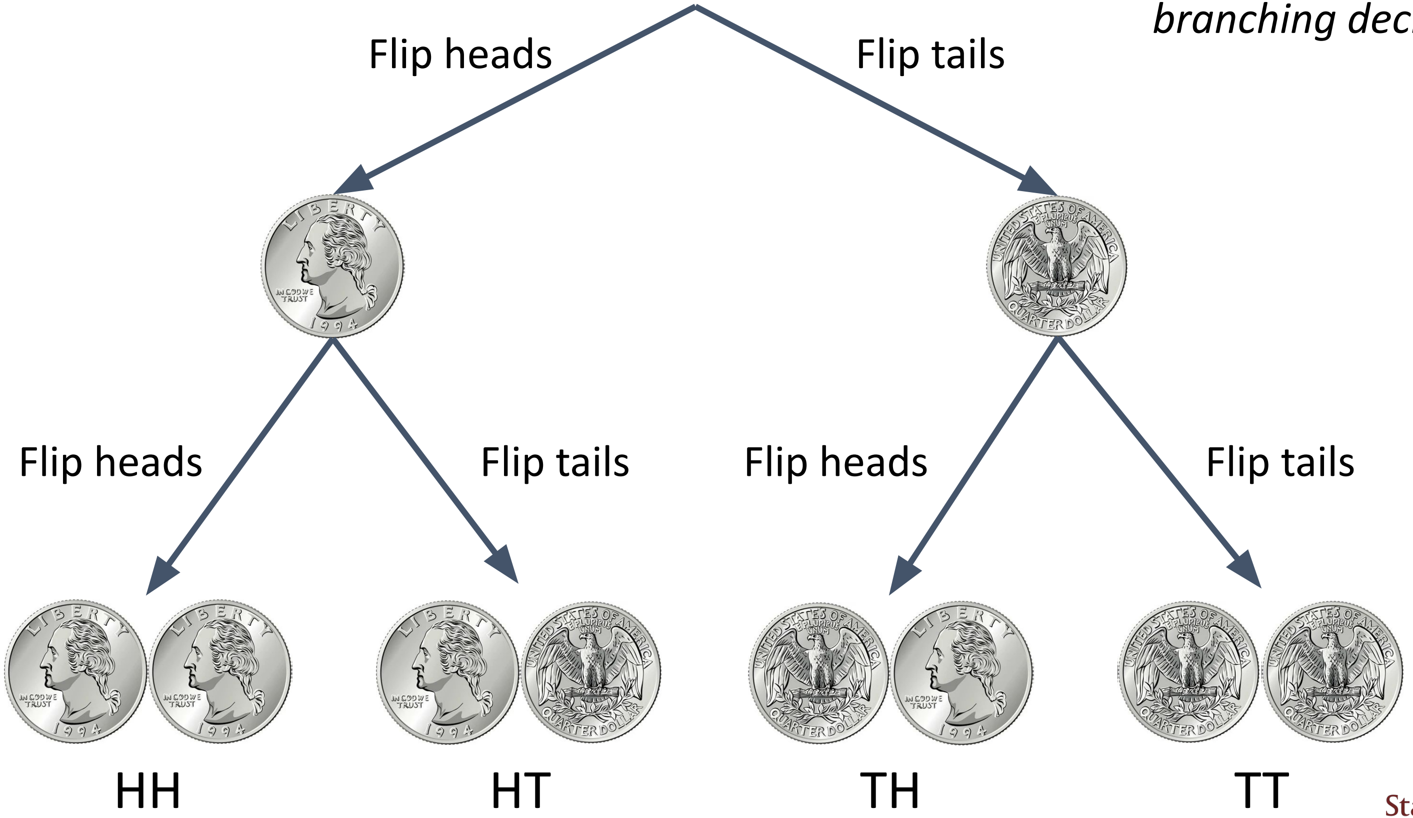
- You’re playing a (rather boring) game in which you flip some number of coins one by one and see whether you get heads or tails
- You’d like to know all of the possible sequences you might flip

HHH HHT HTH HTT THH THT TTH TTT

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

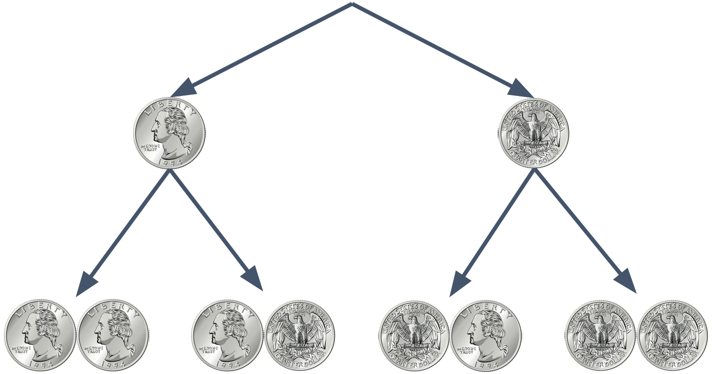
Coin Sequences

This is called a decision tree: at each point we have branching decisions.



Decision Trees

- Decisions trees can help us illustrate our recursive process
- At each point in the tree, we make some decision about how to proceed in our exploration (making a recursive call)

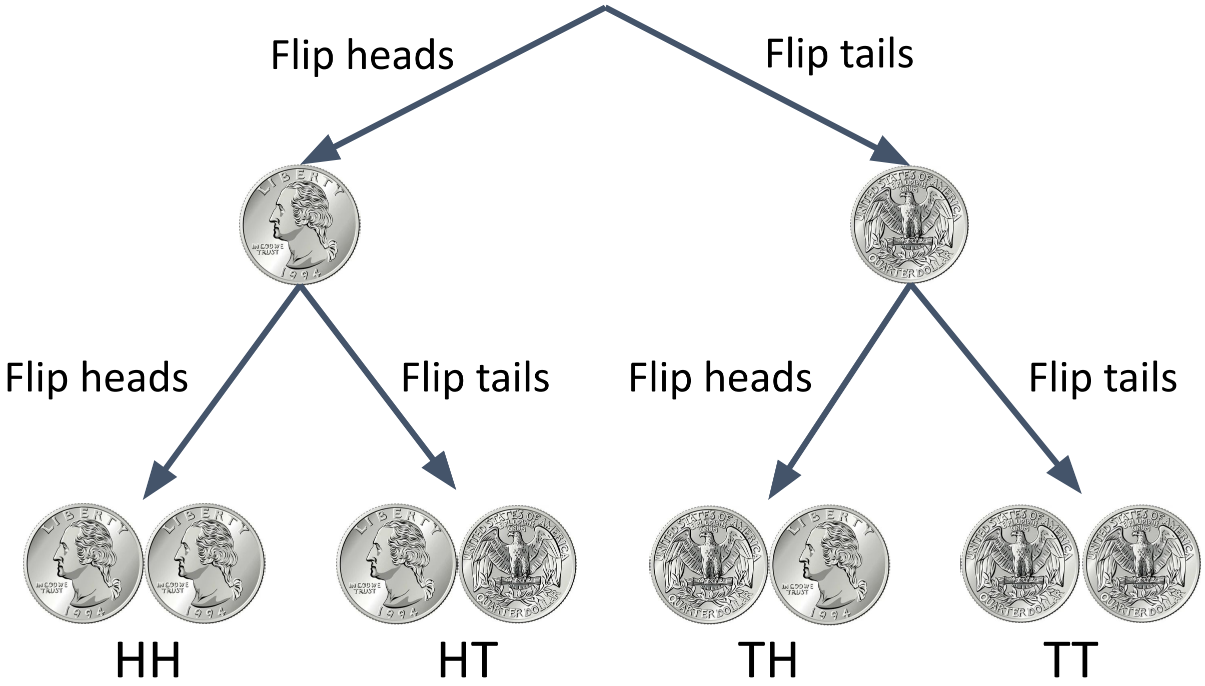


Coin Sequences as a Recursive Process

2 flips left

1 flip left

0 flips left



Flip heads

Flip tails

Flip heads

Flip tails

Flip heads

Flip tails

HH

HT

TH

TT

Coin Sequences as a Recursive Process

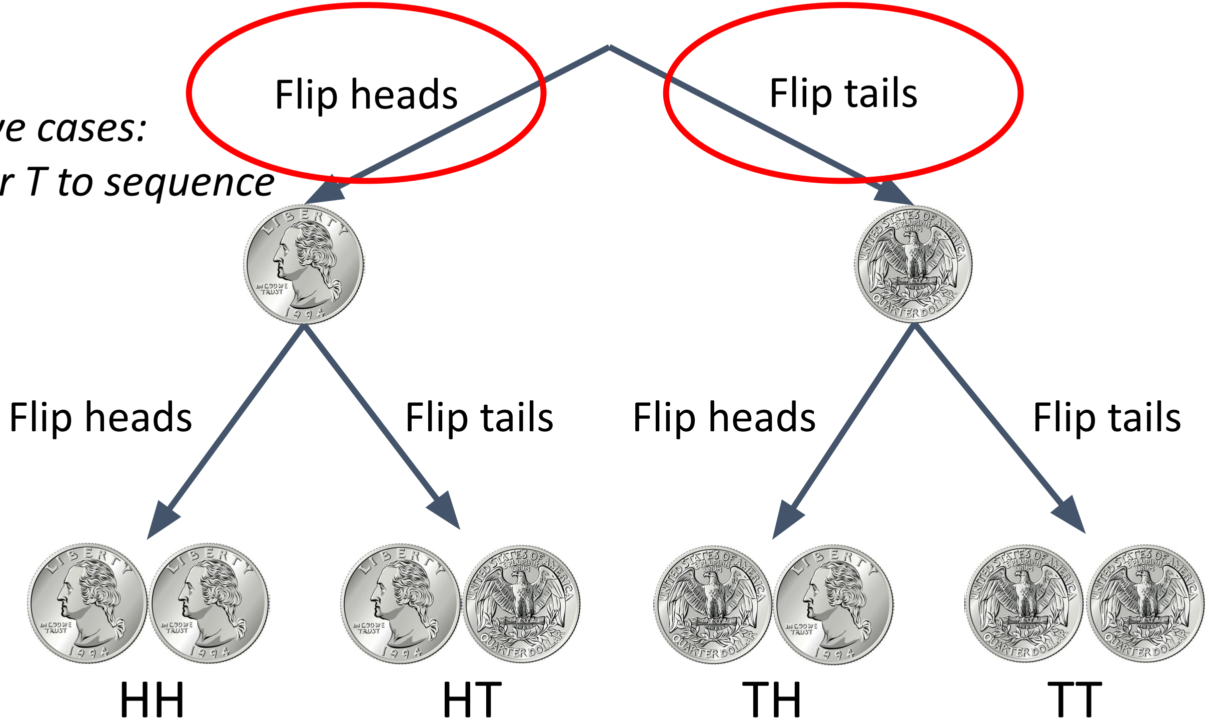
2 flips left

Recursive cases:
Add H or T to sequence

1 flip left

0 flips left

Base case:
Out of flips



Flip heads

Flip tails

Flip heads

Flip tails

Flip heads

Flip tails

HH

HT

TH

TT

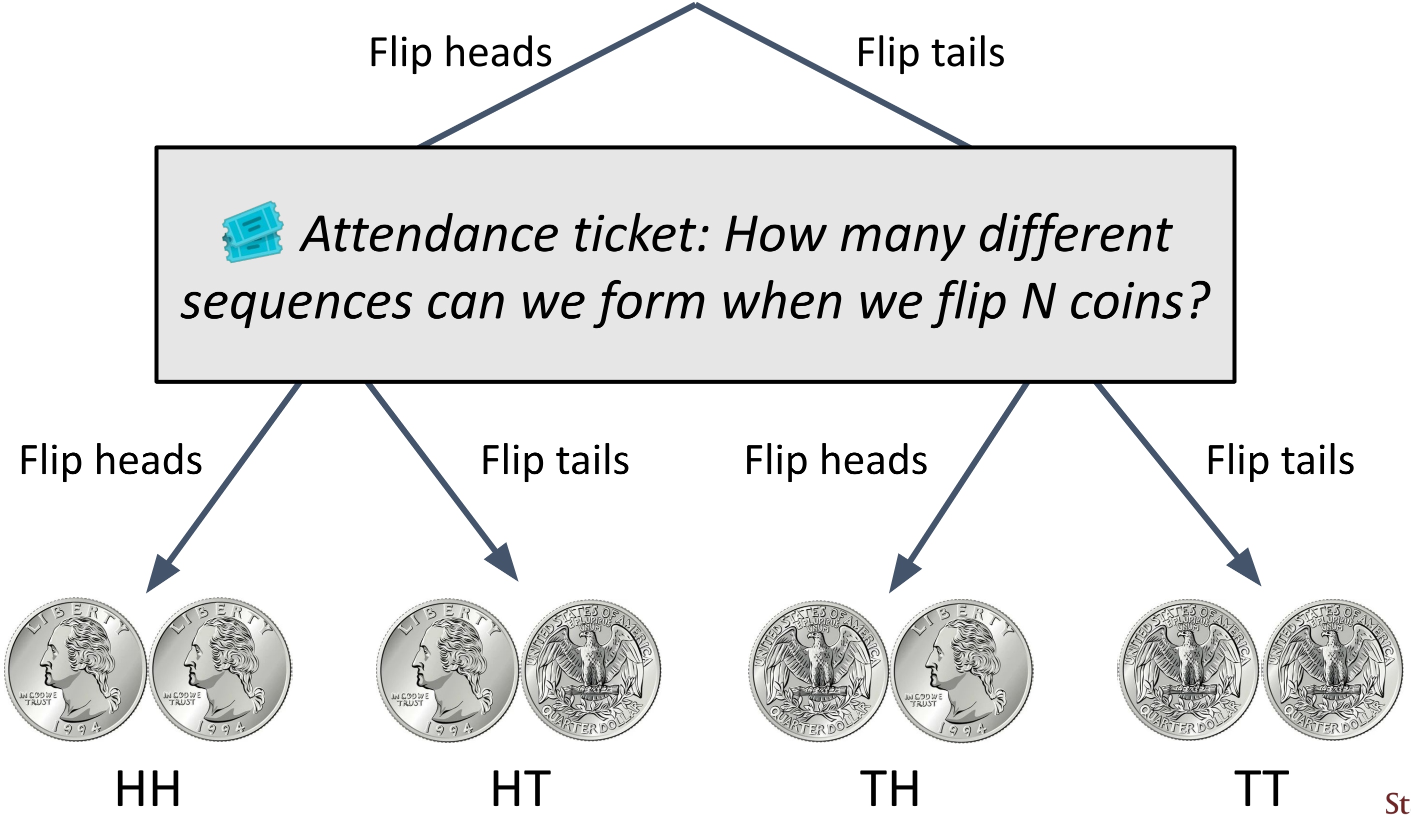
Let’s Code it Up!

Solution

```
void generateSequenceHelper(int flipsRemaining, string sequence) {
    // Base case: flipsRemaining = 0, no more flips
    if (flipsRemaining == 0) {
        cout << sequence << endl;
    } else {
        // Recursive cases (when flipsRemaining > 0)
        generateSequenceHelper(flipsRemaining - 1, sequence + 'H'); // Add H to the sequence
        generateSequenceHelper(flipsRemaining - 1, sequence + 'T'); // OR add T to the sequence
    }
}

void generateSequences(int numCoins) {
    generateSequenceHelper(numCoins, "");
}
```

Coin Sequences



Flip heads

Flip tails

🎟️ *Attendance ticket: How many different sequences can we form when we flip N coins?*

Flip heads

Flip tails

Flip heads

Flip tails

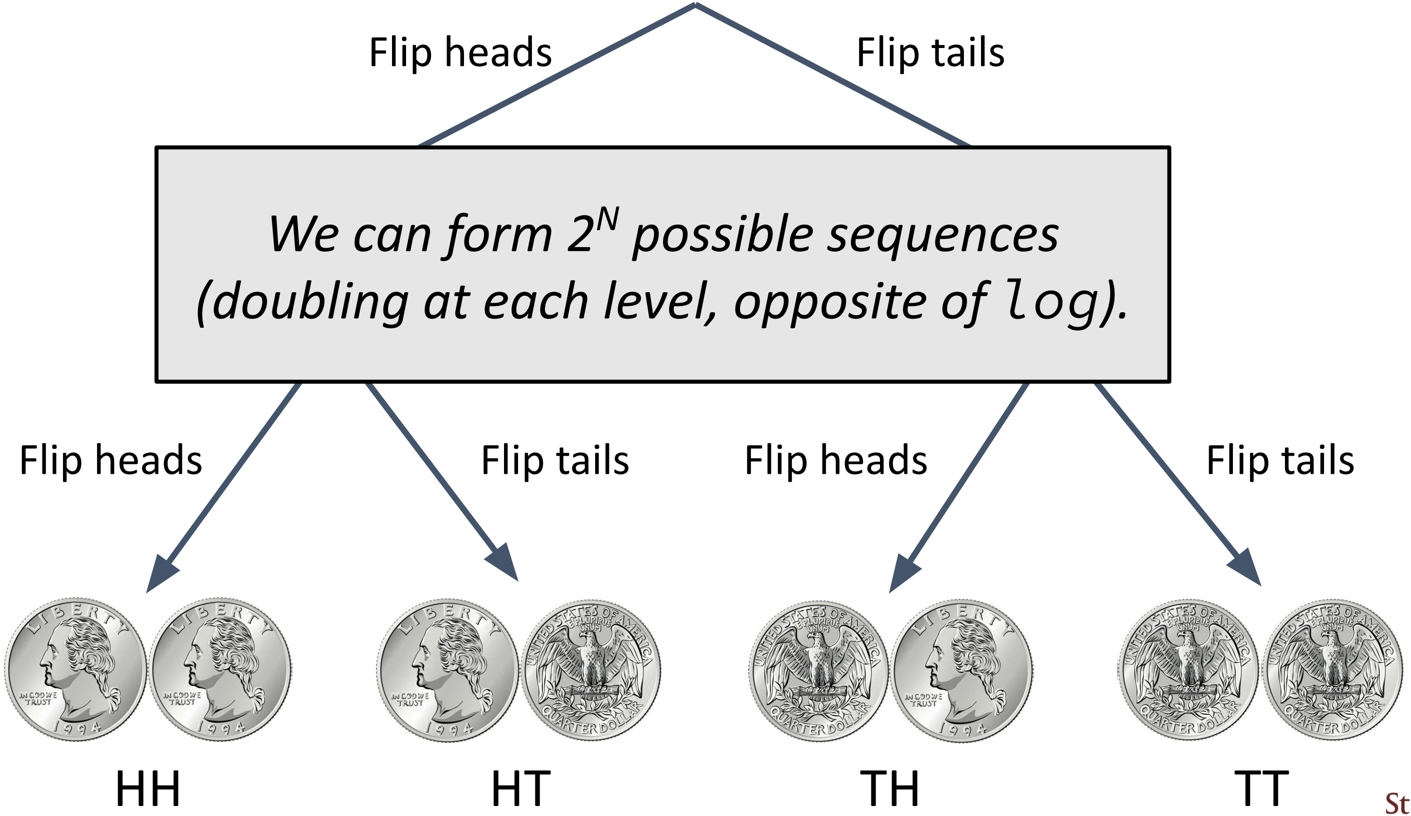
HH

HT

TH

TT

Coin Sequences

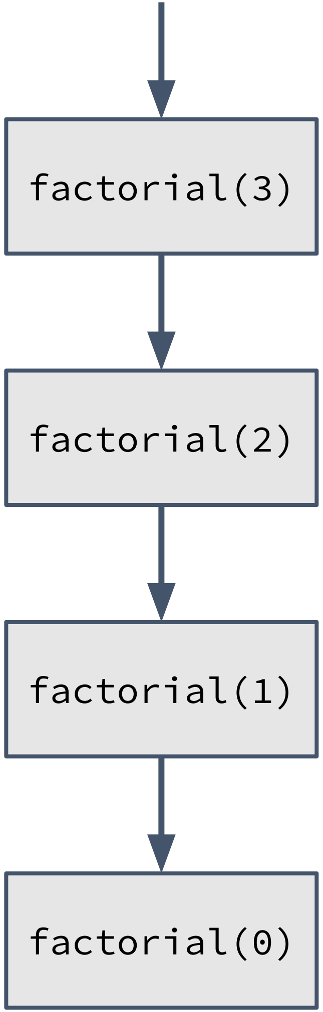


We can form 2^N possible sequences (doubling at each level, opposite of `log`).

Two Types of Recursion

Basic recursion

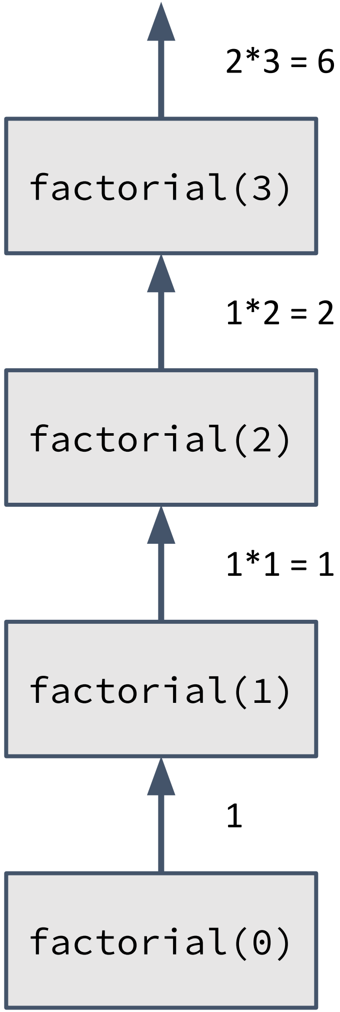
- One repeated task that builds up a solution as you come back up the call stack
- The final base case defines the initial seed of the solution and each call contributes a little bit to the solution



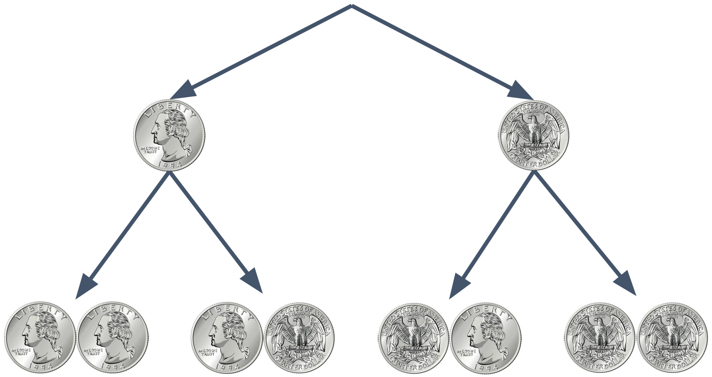
Two Types of Recursion

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



Two Types of Recursion



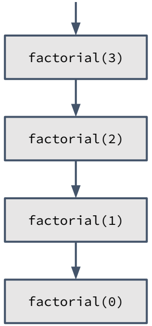
Backtracking recursion

- 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

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



factorial(3)

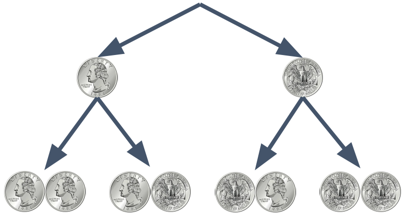
factorial(2)

factorial(1)

factorial(0)

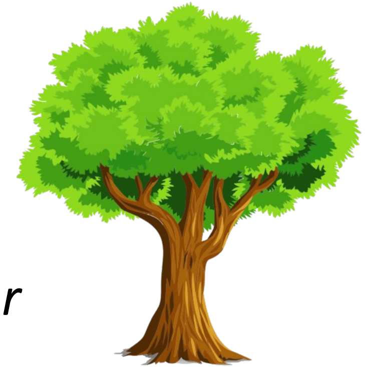
Backtracking recursion

- 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

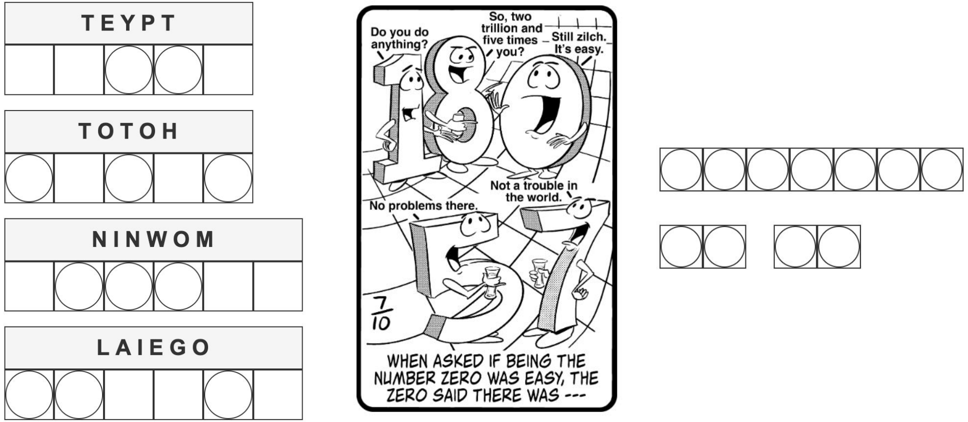


3 Problems to Solve with Backtracking

1. Generate all solutions to a problem or count number of solutions
2. Find one specific solution or prove that one exists
3. Find the best possible solution to a problem



All of these involve exploring many possible solutions, rather than proceeding down a linear path towards one solution.

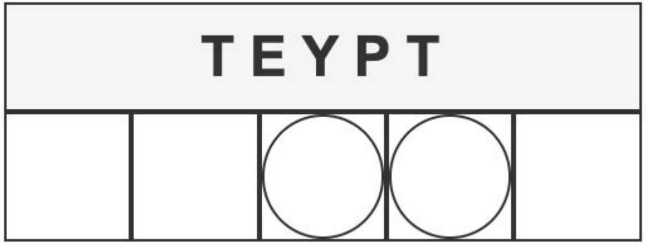

TEYPT
TOTOH
NINWOM
LAIEGO
Do you do anything?
So, two trillion and five times you?
Still zilch. It's easy.
No problems there.
Not a trouble in the world.
7/10
WHEN ASKED IF BEING THE NUMBER ZERO WAS EASY, THE ZERO SAID THERE WAS ---

Word Jumble

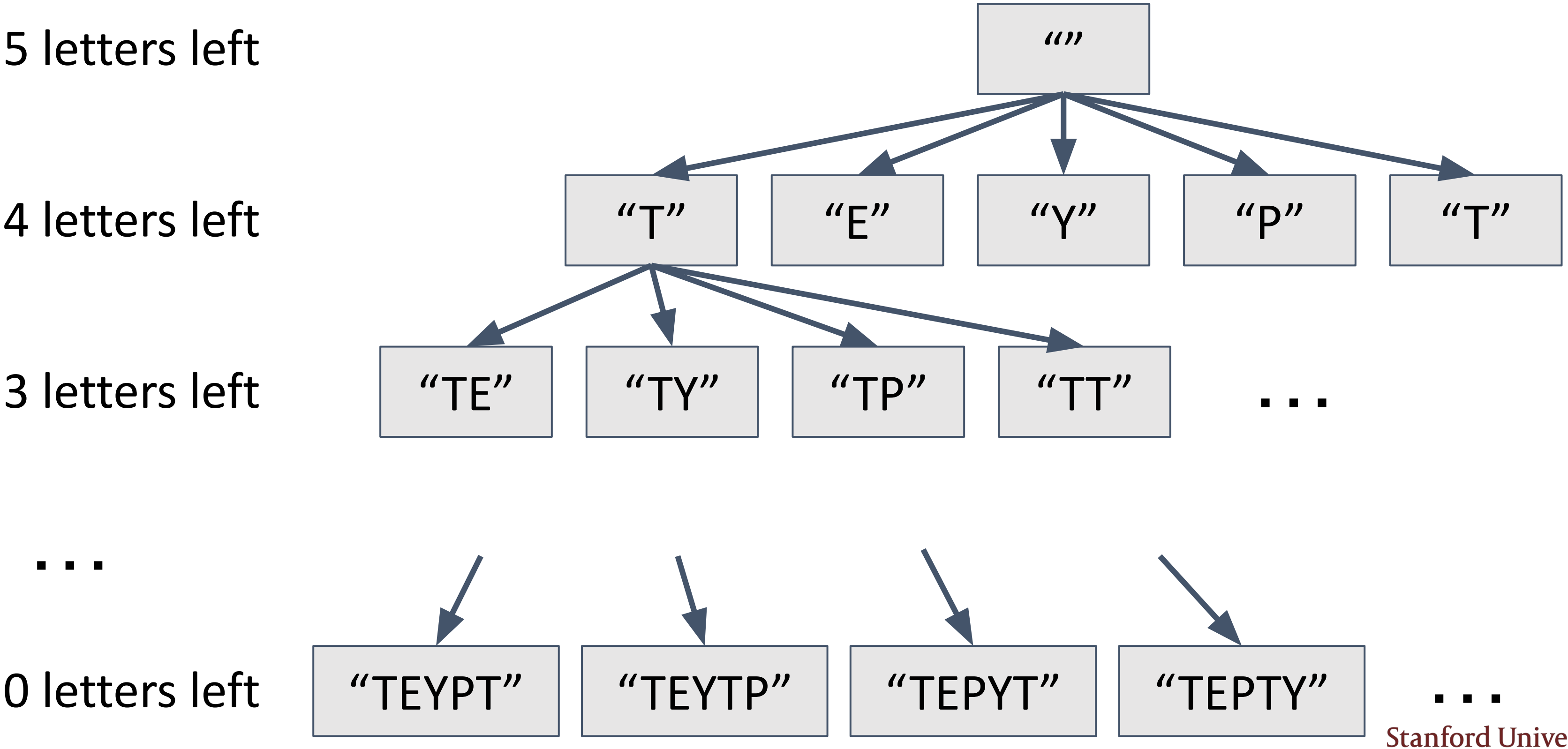
Generate all solutions

Word Jumble

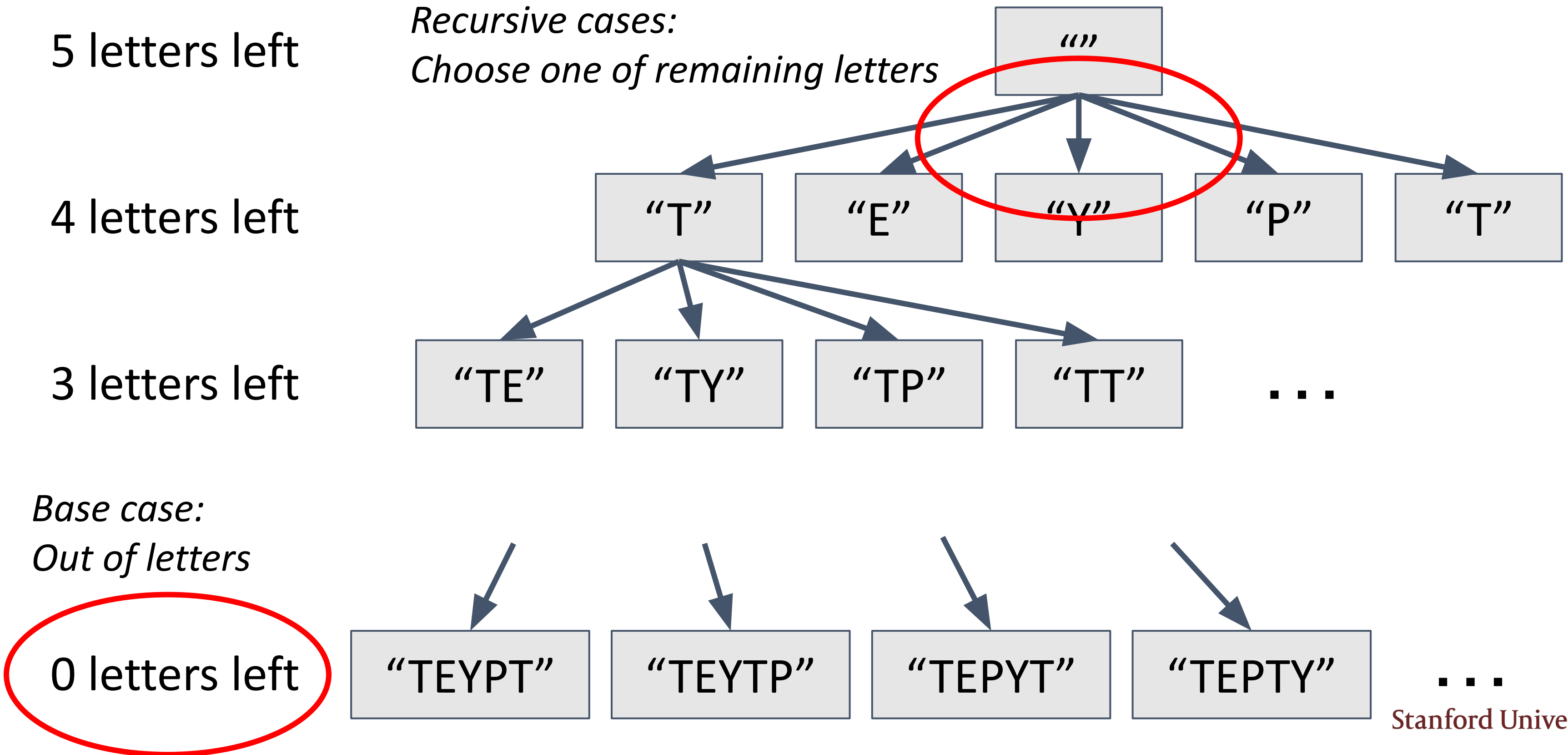
- We’d like to print every ordering of “TEYPT” to solve the puzzle
- This is much like coin sequences, but instead of choosing H or T, we are choosing a letter at each step



Word Jumble


5 letters left
4 letters left
3 letters left
...
0 letters left
“”
“T”
“E”
“Y”
“P”
“T”
“TE”
“TY”
“TP”
“TT”
...
“TEYPT”
“TEYTP”
“TEPYT”
“TEPTY”
...

Word Jumble



5 letters left

Recursive cases:
Choose one of remaining letters

“”

4 letters left

“T” “E” “Y” “P” “T”

3 letters left

“TE” “TY” “TP” “TT” . . .

Base case:
Out of letters

0 letters left

“TEYPT” “TEYTP” “TEPYT” “TEPTY” . . .

From Coins Flips…

```cpp
void generateSequenceHelper(int flipsRemaining, string sequence) {
    // Base case: flipsRemaining = 0, no more flips
    if (flipsRemaining == 0) {
        cout << sequence << endl;
    } else {
        // Two recursive cases (when flipsRemaining > 0)
        generateSequenceHelper(flipsRemaining - 1, sequence + 'H'); // Add H to the sequence
        generateSequenceHelper(flipsRemaining - 1, sequence + 'T'); // OR add T to the sequence
    }
}

void generateSequences(int numCoins) {
    generateSequenceHelper(numCoins, "");
}
```

… To Permutations

```
void generatePermutationsHelper(string lettersRemaining, string sequence) {
    // Base case: lettersRemaining = 0, no more letters to choose from
    if (lettersRemaining.length() == 0) {
        cout << sequence << endl;
    } else {
        // Many recursive cases (when lettersRemaining > 0)
        for (int i = 0; i < lettersRemaining.length(); i++) {
            char letter = lettersRemaining[i];    // choose one of our remaining letters to build onto sequence
            generatePermutationsHelper(lettersRemaining.substr(0, i) + lettersRemaining.substr(i + 1), sequence + letter);
        }
    }
}

void generatePermutations(string word) {
    generatePermutationsHelper(word, "");
}
```

Let’s Check Out the Code!

Takeaways

- "Choose / explore / unchoose" pattern in backtracking

```
for (int i = 0; i < lettersRemaining.length(); i++) {
    // choose a letter
    char letter = lettersRemaining[i];

    // explore this choice by making a recursive call
    generatePermutationsHelper(lettersRemaining.substr(0, i) +
                               lettersRemaining.substr(i + 1), sequence + letter)

    // unchoose this letter by not including it in our sequence next loop
}
```

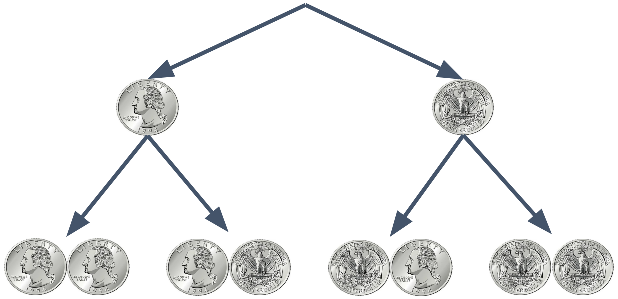
Takeaways

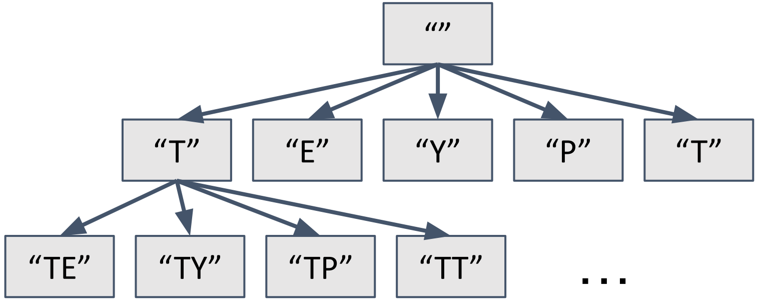
- "Choose / explore / unchoose" pattern in backtracking
- It is important to keep track of the decisions we've made so far and the decisions we have left to make

```
void generatePermutationsHelper(string lettersRemaining, string sequence) {
```

Takeaways

- "Choose / explore / unchoose" pattern in backtracking
- 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





Shrinkable Words

Find a solution

“What nine-letter word can be reduced to a single-letter word one letter at a time, leaving it a legal word at each step?”

startling → starling → staring →

👥 *Finish the sequence!*

startling ⟶ starling ⟶ staring ⟶ string ⟶ sting ⟶ sing ⟶ sin ⟶ in ⟶ i

Shrinkable Words

- A *shrinkable word* is a word that can be reduced down to one letter by removing one letter at a time, leaving a valid word at each step
- Idea: Let’s use a decision tree to remove letters and determine shrinkability!

Shrinkable Words 🤔

Take a few minutes to think:

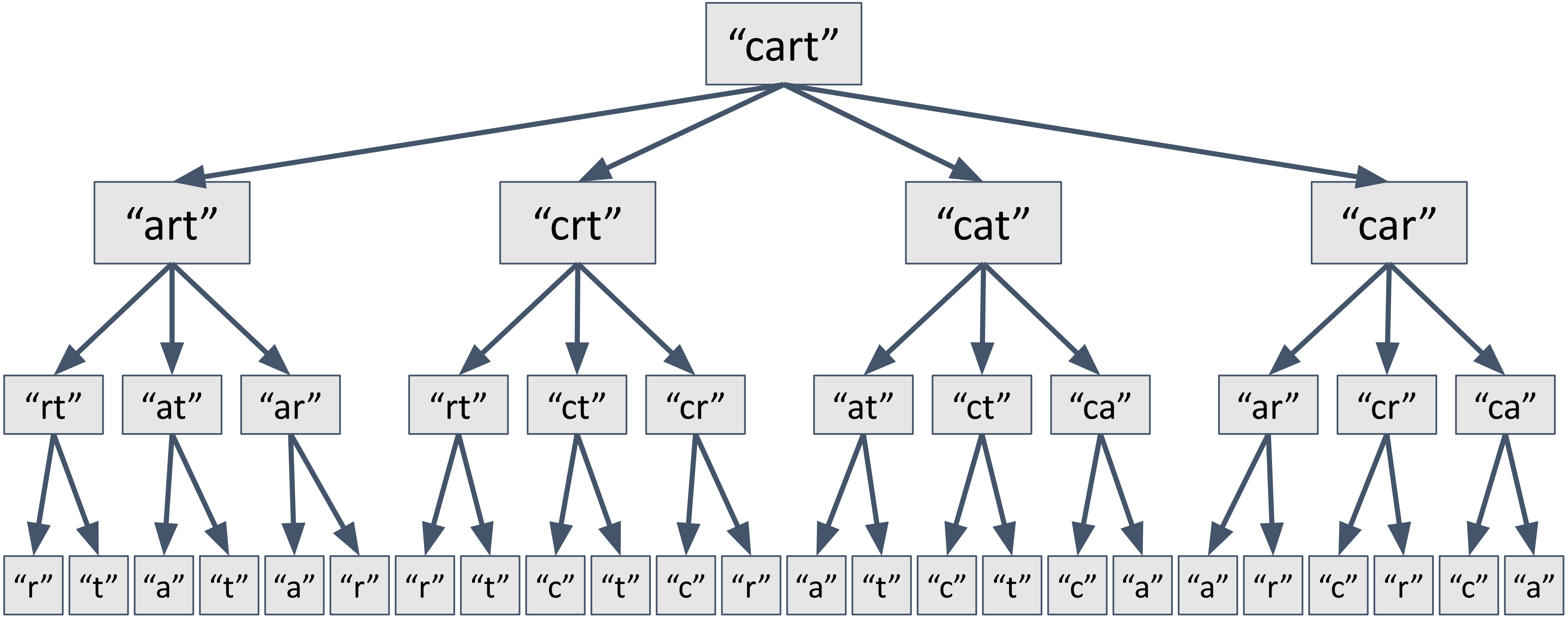
- What are we choosing at each step?
- What is our base case?
- Information to store along the way?

Shrinkable Words

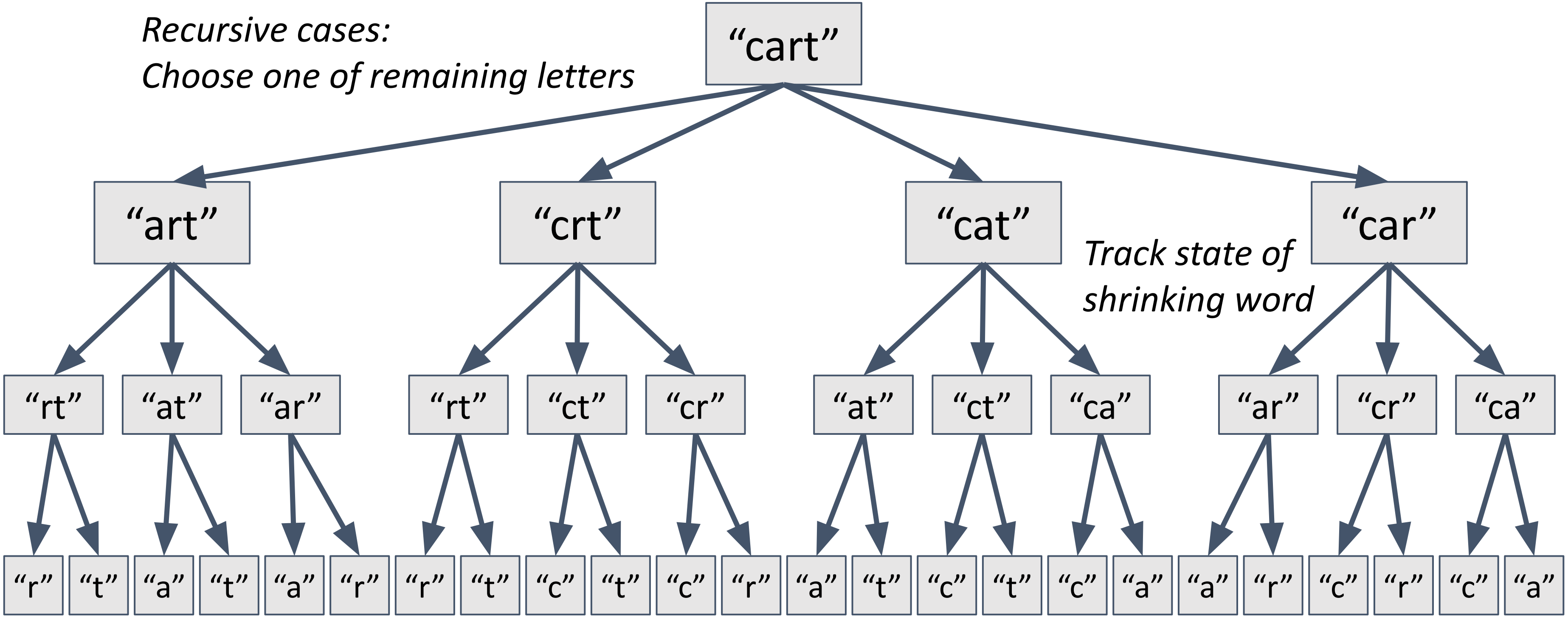
Take a few minutes to think:

- What are we choosing at each step?
    - Which remaining letter to remove from our word
- What is our base case?
    - When we get down to 1 (or 0?) letters
- Information to store along the way?
    - Current state of our shrinking string (letters remaining)

Shrinkable Words Decision Tree



Shrinkable Words Decision Tree

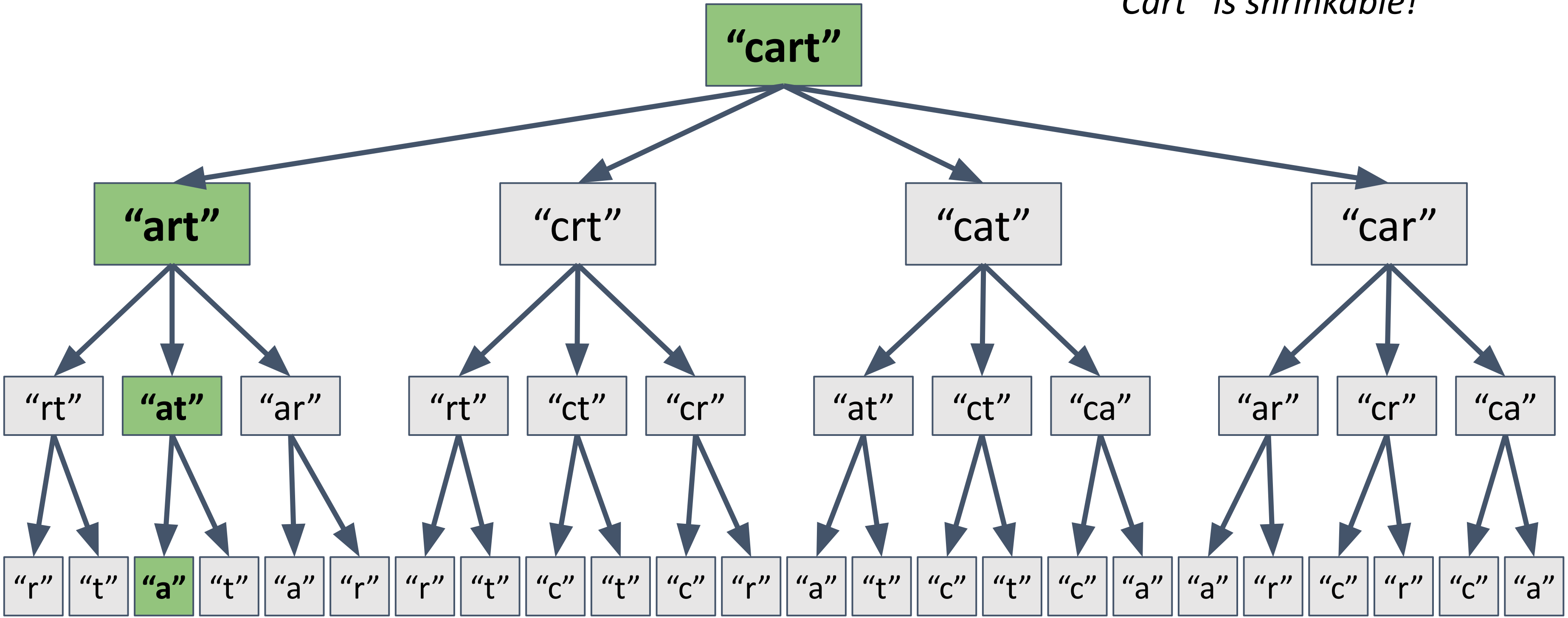


Recursive cases:
Choose one of remaining letters

Track state of
shrinking word

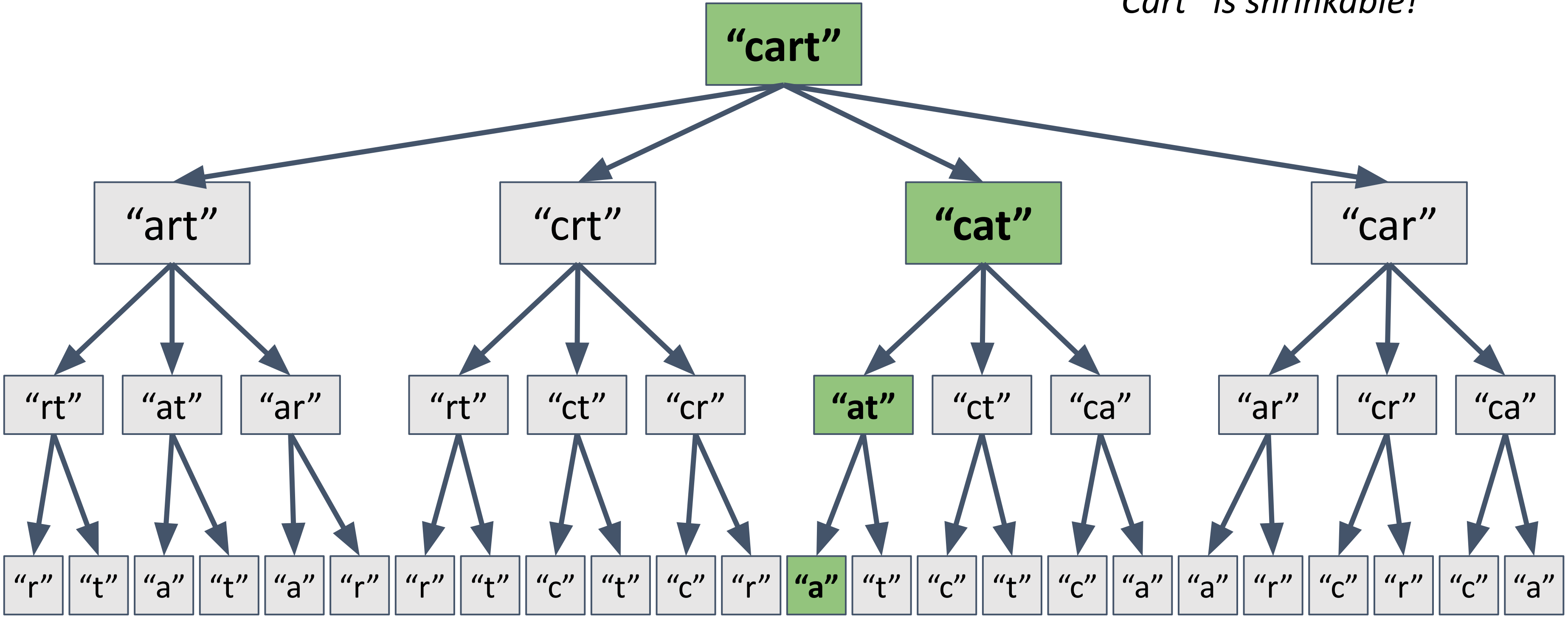
Shrinkable Words Decision Tree

“Cart” is shrinkable!



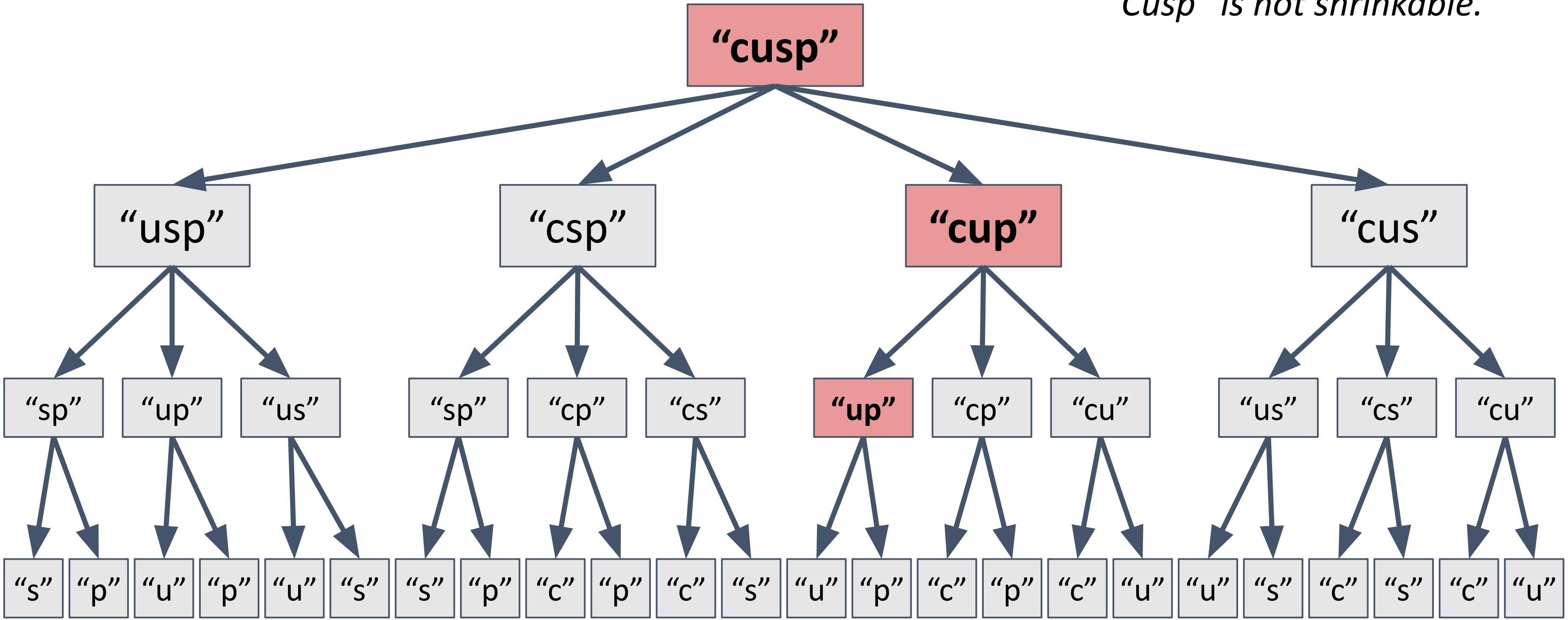
Shrinkable Words Decision Tree

"Cart" is shrinkable!



Shrinkable Words Decision Tree

"Cusp" is not shrinkable.



Shrinkable Words

Base cases:

- We reach an invalid word (failure)
- We get down to a single letter (success)

Shrinkable Words

Base cases:

- We reach an invalid word (failure)
- We get down to a single letter (success)

Recursive cases:

- The word is shrinkable if you can remove any letter and get a shrinkable word
- The word is not shrinkable if no matter what letter you remove, it’s not shrinkable

Lexicon

How do we check if a word is valid? We have an ADT for that:

- `#include “lexicon.h”` (documentation here)

```
Lexicon lex("res/EnglishWords.txt"); // create from file

lex.contains("koala"); // returns true

lex.contains("zzzzz"); // returns false

// returns true if there are any words starting with "fi" in the lexicon

lex.containsPrefix("fi");
```

Let’s Code it Up!

Solution

```
bool isShrinkable(Lexicon& lex, string word) {
    // base case 1) reach invalid word 2) reach final letter
    if (!lex.contains(word)) {
        return false;
    }
    if (word.length() == 1) {
        return true;
    }
    // recursive case: try removing every letter and if any succeeds, return true
    for (int i = 0; i < word.length(); i++) {
        string remainingWord = word.substr(0, i) + word.substr(i + 1);
        if (isShrinkable(lex, remainingWord)) {
            return true;
        }
    }
    return false;
}
```

Alternative Solution

```
bool isShrinkable(Lexicon& lex, string word) {
    // base case 1) run out of letters 2) reach invalid word
    if (word.length() == 0) {
        return true;
    }
    if (!lex.contains(word)) {
        return false;
    }
    // recursive case: try removing every letter and if any succeeds, return true
    for (int i = 0; i < word.length(); i++) {
        string remainingWord = word.substr(0, i) + word.substr(i + 1);
        if (isShrinkable(lex, remainingWord)) {
            return true;
        }
    }
    return false;
}
```

Takeaways

Notice the pattern we used to solve this problem:

```
for all options at each decision point {
    if (recursive call returns true) {
        return true;
    }
}

return false after all options are exhausted;
```

This pattern works well when we're checking if any solution exists.

Recap

- Generating coin sequences: our first backtracking program!
- Two types of recursion: basic vs. backtracking
    - Backtracking allows us to branch and explore many potential solutions
- Three main categories of backtracking
    - All possible solutions: Word Jumble, revisited
    - Find solution: Shrinkable Words
    - Find best solution: we'll explore in the future

See you 🔙 here tomorrow for more backtracking!