Recursive Backtracking: Enumeration

What is a game that would be easy to play if you had the ability to quickly think of all possible moves or plays?

(please put your answers in the chat)
Roadmap

C++ basics

User/client implementation

Recursive problem-solving

Core tools

Algorithmic analysis

Life after CS106B

Arrays
dynamic memory management
linked data structures
real-world algorithms

Diagnostic

Object-Oriented Programming

Arrays
dynamic memory management
linked data structures
real-world algorithms

Recursive problem-solving

Object-Oriented Programming

Vectors + grids
Stacks + queues
Sets + maps

Core tools

Algorithmic analysis

Life after CS106B

Recursive problem-solving

Object-Oriented Programming

Vectors + grids
Stacks + queues
Sets + maps

Core tools

Algorithmic analysis

Life after CS106B

Recursive problem-solving
How can we leverage backtracking recursion to solve interesting problems?
Today’s topics

1. Review
2. Shrinkable Words
3. Generating Subsets
4. Selecting Unbiased Juries
Celebrate Struggle
Review

(advanced recursion patterns and introduction to recursive backtracking)
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
Elegance (Towers of Hanoi)

void findSolution(int n, char source, char dest, char aux) {
    if (n == 1) {
        moveSingleDisk(source, dest);
    } else {
        findSolution(n - 1, source, aux, dest);
        moveSingleDisk(source, dest);
        findSolution(n - 1, aux, dest, source);
    }
}

void findSolutionIterative(int n, char source, char dest, char aux) {

    int numMoves = pow(2, n) - 1; // total number of moves necessary

    // if number of disks is even, swap dest and aux posts
    if (n % 2 == 0) {
        char temp = dest;
        dest = aux;
        aux = temp;
    }

    Stack<int> srcStack;
    for (int i = n; i > 0; i--) {
        srcStack.push(i);
    }
    cout << srcStack << endl;
    Stack<int> destStack;
    Stack<int> auxStack;

    // Determine next move based on how many moves have been made so far
    for (int i = 1; i <= numMoves; i++) {
        switch (i % 3) {
        case 1:
            if (srcStack.isEmpty() ||
                (!destStack.isEmpty() && srcStack.peek() > destStack.peek())) {
                srcStack.push(destStack.pop());
            } else {
                destStack.push(srcStack.pop());
            }
            moveSingleDisk(dest, source);
            break;
        case 2:
            if (srcStack.isEmpty() ||
                (!auxStack.isEmpty() && srcStack.peek() > auxStack.peek())) {
                srcStack.push(auxStack.pop());
            } else {
                auxStack.push(srcStack.pop());
            }
            moveSingleDisk(source, dest);
            break;
        case 0:
            if (destStack.isEmpty() ||
                (!auxStack.isEmpty() && destStack.peek() > auxStack.peek())) {
                destStack.push(auxStack.pop());
            } else {
                auxStack.push(destStack.pop());
            }
            moveSingleDisk(dest, aux);
            break;
        }
    }
}
Efficiency (Binary Search)

- Leverage the structure in sorted data to eliminate half of the search space every time when searching for an element
  - Only do a direct comparison with the middle element in the list
  - Recursively search the left half if the element is less than the middle
  - Recursively search the right half if the element is greater than the middle

- Binary search has logarithmic Big-O: \( O(\log N) \)
  - Enables efficient performance of sets and maps

<table>
<thead>
<tr>
<th>Input Size</th>
<th>Runtime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000000</td>
<td>0.064</td>
</tr>
<tr>
<td>2000000</td>
<td>0.072</td>
</tr>
<tr>
<td>4000000</td>
<td>0.082</td>
</tr>
<tr>
<td>8000000</td>
<td>0.097</td>
</tr>
<tr>
<td>16000000</td>
<td>0.111</td>
</tr>
<tr>
<td>32000000</td>
<td>0.121</td>
</tr>
<tr>
<td>64000000</td>
<td>0.14</td>
</tr>
</tbody>
</table>

Binary Search

<table>
<thead>
<tr>
<th>Input Size</th>
<th>Runtime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>0.096</td>
</tr>
<tr>
<td>20000</td>
<td>0.189</td>
</tr>
<tr>
<td>40000</td>
<td>0.368</td>
</tr>
<tr>
<td>8000000</td>
<td>0.767</td>
</tr>
<tr>
<td>160000</td>
<td>1.387</td>
</tr>
<tr>
<td>320000</td>
<td>2.746</td>
</tr>
<tr>
<td>640000</td>
<td>6.154</td>
</tr>
</tbody>
</table>

Linear Search

Input Size RunTime (s)
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
Dynamic (Coin Sequences + Decision Trees)

- 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 at each decision**.

- To exhaustively explore the entire search space, we must **try every possible option for every possible decision**.
Using backtracking recursion

- There are 3 main categories of problems that we can solve by using backtracking recursion:
Using backtracking recursion

- There are 3 main categories of problems that we can solve by using backtracking recursion:
  - We can generate all possible solutions to a problem or count the total number of possible solutions to a problem
  - We can find one specific solution to a problem or prove that one exists
  - We can find the best possible solution to a given problem
Using backtracking recursion

- There are 3 main categories of problems that we can solve by using backtracking recursion:
  - We can generate all possible solutions to a problem or count the total number of possible solutions to a problem
  - We can find one specific solution to a problem or prove that one exists
  - We can find the best possible solution to a given problem
Using backtracking recursion

- There are 3 main categories of problems that we can solve by using backtracking recursion:
  - We can generate all possible solutions to a problem or count the total number of possible solutions to a problem
  - We can find one specific solution to a problem or prove that one exists
  - We can find the best possible solution to a given problem
Using backtracking recursion

- There are 3 main categories of problems that we can solve by using backtracking recursion:
  - We can generate all possible solutions to a problem or count the total number of possible solutions to a problem
  - We can find one specific solution to a problem or prove that one exists
  - We can find the best possible solution to a given problem

- There are many, many examples of specific problems that we can solve, including
  - Generating permutations
  - Generating subsets
  - Generating combinations
  - And many, many more
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!
Recursive case: For every letter remaining, add that letter to the current permutation and recurse!
Word scramble 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);
        }
    }
}
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);
        }
    }
}
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);
        }
    }
}
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);
        }
    }
}
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);
        }
    }
}
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
How can we leverage backtracking recursion to solve interesting problems?
● A Little Word Puzzle
“What nine-letter word can be reduced to a single-letter word one letter at a time by removing letters, leaving it a legal word at each step?”
The Startling Truth?
The Startling Truth?
The Startling Truth?

STARING
The Startling Truth?
The Startling Truth?
The Startling Truth?

S I N G
The Startling Truth?
The Startling Truth?
The Startling Truth?
Is there really just one nine-letter word with this property?
How can we determine if a word is shrinkable?

- A **shrinkable word** is a word that can be reduced down to one letter by removing one character at a time, leaving a word at each step.

- Idea: Let’s use a decision tree to remove letters and determine **shrinkability**!
What defines our shrinkable decision tree?

- **Decision** at each step (each level of the tree):
  - What letter are going to remove?

- **Options** at each decision (branches from each node):
  - The remaining letters in the string

- Information we need to store along the way:
  - The shrinking string
What defines our shrinkable decision tree?

Examples from Chris Gregg and Keith
What defines our shrinkable decision tree?

"Cart" is shrinkable...

...because "art" is shrinkable....

...because "at" is shrinkable....

...because "a" is a single-letter word.
What defines our shrinkable decision tree?

We can find a path through the tree in two different ways!

Examples from Chris Gregg and Keith
What defines our shrinkable decision tree?

We can find a path through the tree in two different ways!

Examples from Chris Gregg and Keith
What defines our shrinkable decision tree?

We can find a path through the tree in two different ways!

Examples from Chris Gregg and Keith
Non-shrinkability

Examples from Chris Gregg and Keith
Non-shrinksibility

"Up" is not shrinkable... because neither "P" nor "U" are words.
Non-shrinkability

“Cup” is not shrinkable...

...because none of these are shrinkable words.
"Cusp" is not shrinkable...

...because none of these are shrinkable words.

Examples from Chris Gregg and Keith...
How can we determine if a word is shrinkable?

- **Base cases:**
  - A string that is not a word is not a shrinkable word.
  - Any single-letter word is shrinkable (A, I, and O).

- **Recursive cases:**
  - A multi-letter word is shrinkable if you can remove a letter to form a shrinkable word.
  - A multi-letter word is not shrinkable if no matter what letter you remove, it’s not shrinkable.
Lexicon

- Lexicon is a helpful ADT provided by the Stanford C++ libraries (in `lexicon.h`) that is used specifically for storing many words that make up a dictionary.

- Generally, Lexicons offer faster lookup than normal Sets, which is why we choose to use them when dealing with words and large dictionaries.

```cpp
Lexicon lex("res/EnglishWords.txt"); // create from file
lex.contains("koala"); // returns true
lex.contains("zzzzz"); // returns false
lex.containsPrefix("fi"); // returns true if there are any words starting with "fi" in the dictionary
```
Let’s code it!
Takeaways

- This is another example of **copy-edit-recurse** to choose, explore, and then implicitly unchoose!

- In this problem, we’re using backtracking to **find if a solution exists**.
  - Notice the way the recursive case is structured:

    ```
    for all options at each decision point:
    if recursive call returns true:
      return true;
    return false if all options are exhausted;
    ```
Announcements
Announcements

• Assignment 3 was released last Thursday.
  • The Assignment 3 YEAH session slides and recording have been posted.
  • This assignment is challenging and quite long; please come to LaIR or office hours to check in with the teaching team if you need anything!

• We’ve released practice problems and information about the diagnostic. You’ll be able to take the diagnostic over the weekend.

• Please make sure to read the instructions on the diagnostic page of the website, and verify that you can access your Gradescope account before the weekend!
Subsets
Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:
Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:
Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:
Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

\[
\{\}
\]

Even though we may not care about this “team,” the empty set is a subset of our original set!
Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

\{
\}

As humans, it might be easiest to think about all teams (subsets) of a particular size.
Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

{}  
{“Nick”}  
{“Kylie”}  
{“Trip”}

As humans, it might be easiest to think about all teams (subsets) of a particular size.
Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

\[
\begin{align*}
\{
\} \\
\{\text{“Nick”}\} \\
\{\text{“Kylie”}\} \\
\{\text{“Trip”}\} \\
\{\text{“Nick”, “Kylie”}\} \\
\{\text{“Nick”, “Trip”}\} \\
\{\text{“Kylie”, “Trip”}\}
\end{align*}
\]

As humans, it might be easiest to think about all teams (subsets) of a particular size.
Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

{}  
{“Nick”}  
{“Kylie”}  
{“Trip”}  
{“Nick”, “Kylie”}  
{“Nick”, “Trip”}  
{“Kylie”, “Trip”}  
{“Nick”, “Kylie”, “Trip”}

As humans, it might be easiest to think about all teams (subsets) of a particular size.
Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

{}  
{“Nick”}  
{“Kylie”}  
{“Trip”}  
{“Nick”, “Kylie”}  
{“Nick”, “Trip”}  
{“Kylie”, “Trip”}  
{“Nick”, “Kylie”, “Trip”}

Another case of “generate/count all solutions” using recursion and backtracking!
Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

{}  
{"Nick"}  
{"Kylie"}  
{"Trip"}  
{"Nick", "Kylie"}  
{"Nick", "Trip"}  
{"Kylie", "Trip"}  
{"Nick", "Kylie", "Trip"}

For computers generating subsets (and thinking about decisions), there’s another pattern we might notice.
Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

\[
\begin{align*}
&\{\} \\
&\{\text{“Nick”}\} \\
&\{\text{“Kylie”}\} \\
&\{\text{“Trip”}\} \\
&\{\text{“Nick”, “Kylie”}\} \\
&\{\text{“Nick”, “Trip”}\} \\
&\{\text{“Kylie”, “Trip”}\} \\
&\{\text{“Nick”, “Kylie”, “Trip”}\}
\end{align*}
\]

Half the subsets contain “Nick”
Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

{ }
{“Nick”}
{“Kylie”}
{“Trip”}
{“Nick”, “Kylie”}
{“Nick”, “Trip”}
{“Kylie”, “Trip”}
{“Nick”, “Kylie”, “Trip”}

Half the subsets contain “Kylie”
Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

{}  
{"Nick"}  
{"Kylie"}  
{"Trip"}  
{"Nick", "Kylie"}  
{"Nick", "Trip"}  
{"Kylie", "Trip"}  
{"Nick", "Kylie", "Trip"}

Half the subsets contain "Trip"
Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

- \{
- \{“Nick”\}
- \{“Kylie”\}
- \{“Trip”\}
- \{“Nick”, “Kylie”\}
- \{“Nick”, “Trip”\}
- \{“Kylie”, “Trip”\}
- \{“Nick”, “Kylie”, “Trip”\}

Half the subsets that contain “Trip” also contain “Nick”
Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

```javascript
{}
{“Nick”}
{“Kylie”}
{“Trip”}
{“Nick”, “Kylie”}
{“Nick”, “Trip”}
{“Kylie”, “Trip”}
{“Nick”, “Kylie”, “Trip”}
```

Half the subsets that contain both “Trip” and “Nick” contain “Kylie”
Subsets

Given a group of people, suppose we wanted to generate all possible teams, or subsets, of those people:

- {}
- {“Nick”}
- {“Kylie”}
- {“Trip”}
- {“Nick”, “Kylie”}
- {“Nick”, “Trip”}
- {“Kylie”, “Trip”}
- {“Nick”, “Kylie”, “Trip”}
It’s time to draw a tree!!!
What defines our subsets decision tree?

- **Decision** at each step (each level of the tree):
  - Are we going to include a given element in our subset?
What defines our subsets \textit{decision tree}?

- **Decision** at each step (each level of the tree):
  - Are we going to include a given element in our subset?

- **Options** at each decision (branches from each node):
  - Include element
  - Don’t include element
What defines our subsets decision tree?

- **Decision** at each step (each level of the tree):
  - Are we going to include a given element in our subset?

- **Options** at each decision (branches from each node):
  - Include element
  - Don’t include element

- Information we need to store along the way:
  - The set you’ve built so far
  - The remaining elements in the original set
Decision tree

Don’t include Nick  Empty set  Include Nick
Decision tree

Don’t include Nick

Empty set

Include Nick

No Kylie

Kylie
Decision tree

Don’t include Nick

Kylie

No Kylie

No Trip

Trip

Include Nick

No Kylie

Kylie

No Trip

Trip

No Kylie

Kylie

No Trip

Trip

No Kylie

Kylie

No Trip

Trip

No Kylie

Kylie

No Trip

Trip

No Kylie

Kylie

No Trip

Trip
What defines our subsets decision tree?

- **Decision** at each step (each level of the tree):
  - Are we going to include a given element in our subset?

- **Options** at each decision (branches from each node):
  - Include element
  - Don’t include element

- Information we need to store along the way:
  - The set you’ve built so far
  - **The remaining elements in the original set**
Decision tree

Remaining: {'Nick', 'Kylie', 'Trip'}
Decision tree

Remaining: {“Nick”, “Kylie”, “Trip”}

Remaining: {“Kylie”, “Trip”}
Decision tree

Remaining: {"Nick", "Kylie", "Trip"}

Remaining: {"Kylie", "Trip"}

Remaining: {"Trip"}

Remaining: {""}
Base case: No people remaining to choose from!
Recursive case: Pick someone in the set. Choose to include or not include them.
How do recursive backtracking solutions look different when data structures are involved?
Let’s code it!
Takeaways

- This is our first time seeing an explicit “unchoose” step
  - This is necessary because we’re passing sets by reference and editing them!
Takeaways

- This is our first time seeing an explicit “unchoose” step
  - This is necessary because we’re passing sets by reference and editing them!

```java
string elem = remaining.first();
// remove this element from possible choices
remaining = remaining - elem;
listSubsetsHelper(remaining, chosen); // do not add elem to chosen
chosen = chosen + elem;
listSubsetsHelper(remaining, chosen); // add elem to chosen
chosen = chosen - elem;
// add this element back to possible choices
remaining = remaining + elem;
```
Takeaways

- This is our first time seeing an explicit “unchoose” step
  - This is necessary because we’re passing sets by reference and editing them!

```java
string elem = remaining.first();
// remove this element from possible choices
remaining = remaining - elem;
listSubsetsHelper(remaining, chosen); // do not add elem to chosen
chosen = chosen + elem;
listSubsetsHelper(remaining, chosen); // add elem to chosen
chosen = chosen - elem;
// add this element back to possible choices
remaining = remaining + elem;
```
Takeaways

- This is our first time seeing an explicit “unchoose” step
  - This is necessary because we’re passing sets by reference and editing them!

```java
string elem = remaining.first();
// remove this element from possible choices
remaining = remaining - elem;
listSubsetsHelper(remaining, chosen); // do not add elem to chosen
chosen = chosen + elem;
listSubsetsHelper(remaining, chosen); // add elem to chosen
chosen = chosen - elem;
// add this element back to possible choices
remaining = remaining + elem;
```

Explore (part 1)
Takeaways

- This is our first time seeing an explicit “unchoose” step
  - This is necessary because we’re passing sets by reference and editing them!

```java
string elem = remaining.first();
// remove this element from possible choices
remaining = remaining - elem;
listSubsetsHelper(remaining, chosen); // do not add elem to chosen
chosen = chosen + elem;
listSubsetsHelper(remaining, chosen); // add elem to chosen
chosen = chosen - elem;
// add this element back to possible choices
remaining = remaining + elem;
```

Explore (part 2)
Takeaways

- This is our first time seeing an explicit “unchoose” step
  - This is necessary because we’re passing sets by reference and editing them!

```java
string elem = remaining.first();
// remove this element from possible choices
remaining = remaining - elem;
listSubsetsHelper(remaining, chosen); // do not add elem to chosen
chosen = chosen + elem;
listSubsetsHelper(remaining, chosen); // add elem to chosen
chosen = chosen - elem;
// add this element back to possible choices
remaining = remaining + elem;
```
Decision tree

Remaining: {'Nick', 'Kylie', 'Trip'}

Remaining: {'Kylie', 'Trip'}

Remaining: {'Trip'}

Remaining: {}
Takeaways

- This is our first time seeing an explicit “unchoose” step
  - This is necessary because we’re passing sets by reference and editing them!

```java
string elem = remaining.first(); // remove this element from possible choices
remaining = remaining - elem;
listSubsetsHelper(remaining, chosen); // do not add elem to chosen
chosen = chosen + elem;
listSubsetsHelper(remaining, chosen); // add elem to chosen
chosen = chosen - elem; // add this element back to possible choices
remaining = remaining + elem;
```

Without this step, we could not explore the other side of the tree
Takeaways

- This is our first time seeing an explicit “unchoose” step
  - This is necessary because we’re passing sets by reference and editing them!

- It’s important to consider not only decisions and options at each decision, but also to keep in mind what information you have to keep track of with each recursive call. This might help you define your base case.
Takeaways

- This is our first time seeing an explicit “unchoose” step
  - This is necessary because we’re passing sets by reference and editing them!

- It’s important to consider not only decisions and options at each decision, but also to keep in mind what information you have to keep track of with each recursive call. This might help you define your base case.

- The subset problem contains themes we’ve seen in backtracking recursion:
  - Building up solutions as we go down the decision tree
  - Using a helper function to abstract away implementation details
Application: Choosing an Unbiased Jury
Jury Selection

- The process of jury selection involves processing a large pool of candidates that have been called for jury duty, and selecting some small subset of those candidates to serve on the jury.
Photo from 2007 TV show, the Verdict
Jury Selection

- The process of jury selection involves processing a large pool of candidates that have been called for jury duty, and selecting some small subset of those candidates to serve on the jury.

- When selecting members of a jury, each individual person will come in with their own biases that might sway the case.
Jury Selection

● The process of jury selection involves processing a large pool of candidates that have been called for jury duty, and selecting some small subset of those candidates to serve on the jury.

● When selecting members of a jury, each individual person will come in with their own biases that might sway the case.

● Ideally, we would like to select a jury that is unbiased (sum of all biases is 0)
Jury Selection

- The process of jury selection involves processing a large pool of candidates that have been called for jury duty, and selecting some small subset of those candidates to serve on the jury.

- When selecting members of a jury, each individual person will come in with their own biases that might sway the case.

- Ideally, we would like to select a jury that is unbiased (sum of all biases is 0)

- **An unbiased jury is just a subset with a special property** – let's apply the code that we just wrote!
What defines our subsets decision tree?

- **Decision** at each step (each level of the tree):
  - Are we going to include a given element in our subset?

- **Options** at each decision (branches from each node):
  - Include element
  - Don’t include element

- Information we need to store along the way:
  - The set you’ve built so far
  - The remaining elements in the original set
What defines our jury selection decision tree?

- **Decision** at each step (each level of the tree):
  - Are we going to include a given candidate in our jury?

- **Options** at each decision (branches from each node):
  - Include candidate
  - Don’t include candidate

- Information we need to store along the way:
  - The collection of candidates making up our jury so far
  - The remaining candidates to consider
  - The sum total bias of the current jury so far
Jury Selection Pseudocode

- Problem Setup
  - Assume that we have defined a custom `juror` struct, which packages up important information about a juror (their name and their bias, represented as an `int`
Jury Selection Pseudocode

- Problem Setup
  - Assume that we have defined a custom `juror` struct, which packages up important information about a juror (their `name` and their `bias`, represented as an `int`)
  - Given a `Vector<juror>` (there may be duplicate name/bias pairs among candidates), we want to print out all possible unbiased juries that can be formed
Jury Selection Pseudocode

● Problem Setup
  ○ Assume that we have defined a custom juror struct, which packages up important information about a juror (their name and their bias, represented as an int)
  ○ Given a vector<juror> (there may be duplicate name/bias pairs among candidates), we want to print out all possible unbiased juries that can be formed

● Recursive Case
  ○ Select a candidate that hasn't been considered yet.
  ○ Try not including them in the jury, and recursively find all possible unbiased juries.
  ○ Try including them in the jury, and recursively find all possible unbiased juries.
Jury Selection Pseudocode

- Problem Setup
  - Assume that we have defined a custom juror struct, which packages up important information about a juror (their name and their bias, represented as an int)
  - Given a vector<juror> (there may be duplicate name/bias pairs among candidates), we want to print out all possible unbiased juries that can be formed

- Recursive Case
  - Select a candidate that hasn't been considered yet.
  - Try not including them in the jury, and recursively find all possible unbiased juries.
  - Try including them in the jury, and recursively find all possible unbiased juries.

- Base Case
  - Once we're out of candidates to consider, check the bias of the current jury. If 0, display them!
void findAllUnbiasedJuriesHelper(Vector<juror>& allCandidates, Vector<juror>& currentJury, int currentBias){
    if (allCandidates.isEmpty()){
        if (currentBias == 0){
            displayJury(currentJury);
        }
    } else {
        juror currentCandidate = allCandidates[0];
        allCandidates.remove(0);

        findAllUnbiasedJuriesHelper(allCandidates, currentJury, currentBias);
        currentJury.add(currentCandidate);
        findAllUnbiasedJuriesHelper(allCandidates, currentJury, currentBias + currentCandidate.bias);
        currentJury.remove(currentJury.size() - 1);
        allCandidates.insert(0, currentCandidate);
    }
}

void findAllUnbiasedJuries(Vector<juror>& allCandidates){
    Vector<juror> jury;
    findAllUnbiasedJuriesHelper(allCandidates, jury, 0);
}
void findAllUnbiasedJuriesHelper(Vector<juror>& allCandidates, Vector<juror>& currentJury, int currentBias)
{
    if (allCandidates.isEmpty()){
        if (currentBias == 0){
            displayJury(currentJury);
        }
    } else {
        juror currentCandidate = allCandidates[0];
        allCandidates.remove(0);

        findAllUnbiasedJuriesHelper(allCandidates, currentJury, currentBias);
        currentJury.add(currentCandidate);
        findAllUnbiasedJuriesHelper(allCandidates, currentJury, currentBias + currentCandidate.bias);
        currentJury.remove(currentJury.size() - 1);
        allCandidates.insert(0, currentCandidate);
    }
}

void findAllUnbiasedJuries(Vector<juror>& allCandidates){
    Vector<juror> jury;
    findAllUnbiasedJuriesHelper(allCandidates, jury, 0);
}

Helper function: Extra variable to keep track of total bias
void findAllUnbiasedJuriesHelper(Vector<juror>& allCandidates, Vector<juror>& currentJury, int currentBias){
    if (allCandidates.isEmpty()){
        if (currentBias == 0){
            displayJury(currentJury);
        }
    } else {
        juror currentCandidate = allCandidates[0];
        allCandidates.remove(0);

        findAllUnbiasedJuriesHelper(allCandidates, currentJury, currentBias);
        currentJury.add(currentCandidate);
        findAllUnbiasedJuriesHelper(allCandidates, currentJury, currentBias + currentCandidate.bias);
        currentJury.remove(currentJury.size() - 1);
        allCandidates.insert(0, currentCandidate);
    }
}

void findAllUnbiasedJuries(Vector<juror>& allCandidates){
    Vector<juror> jury;
    findAllUnbiasedJuriesHelper(allCandidates, jury, 0);
}

Base case: Only display juries with no total bias
void findAllUnbiasedJuriesHelper(Vector<juror>& allCandidates, Vector<juror>& currentJury, int currentBias) {
    if (allCandidates.isEmpty()) {
        if (currentBias == 0) {
            displayJury(currentJury);
        }
    } else {
        juror currentCandidate = allCandidates[0];
        allCandidates.remove(0);
        findAllUnbiasedJuriesHelper(allCandidates, currentJury, currentBias);
        currentJury.add(currentCandidate);
        findAllUnbiasedJuriesHelper(allCandidates, currentJury, currentBias + currentCandidate.bias);
        currentJury.remove(currentJury.size() - 1);
        allCandidates.insert(0, currentCandidate);
    }
}

void findAllUnbiasedJuries(Vector<juror>& allCandidates) {
    Vector<juror> jury;
    findAllUnbiasedJuriesHelper(allCandidates, jury, 0);
}

Recursive case: Consider juries both with and without this person
Jury Selection
Optimization
void findAllUnbiasedJuriesHelper(Vector<juror>& allCandidates, Vector<juror>& currentJury, int currentBias) {
    if (allCandidates.isEmpty()) {
        if (currentBias == 0) {
            displayJury(currentJury);
        }
    } else {
        juror currentCandidate = allCandidates[0];
        allCandidates.remove(0);

        findAllUnbiasedJuriesHelper(allCandidates, currentJury, currentBias);
        currentJury.add(currentCandidate);
        findAllUnbiasedJuriesHelper(allCandidates, currentJury, currentBias + currentCandidate.bias);
        currentJury.remove(currentJury.size() - 1);
        allCandidates.insert(0, currentCandidate);
    }
}

void findAllUnbiasedJuries(Vector<juror>& allCandidates) {
    Vector<juror> jury;
    findAllUnbiasedJuriesHelper(allCandidates, jury, 0);
}
void findAllUnbiasedJuriesHelper(Vector<juror>& allCandidates, Vector<juror>& currentJury, int currentBias)
{
    if (allCandidates.isEmpty()){
        if (currentBias == 0){
            displayJury(currentJury);
        }
    } else {
        juror currentCandidate = allCandidates[0];
        allCandidates.remove(0);

        findAllUnbiasedJuriesHelper(allCandidates, currentJury, currentBias);
        currentJury.add(currentCandidate);
        findAllUnbiasedJuriesHelper(allCandidates, currentJury, currentBias + currentCandidate.bias);
        currentJury.remove(currentJury.size() - 1);
        allCandidates.insert(0, currentCandidate);
    }
}

void findAllUnbiasedJuries(Vector<juror>& allCandidates){
    Vector<juror> jury;
    findAllUnbiasedJuriesHelper(allCandidates, jury, 0);
}

Vector addition/removal can be an expensive operation. Can we do better?
Optimizing Subset Creation

- The core component of subset generation includes visiting each element once, and making a decision about whether to include it or not.
Optimizing Subset Creation

- The core component of subset generation includes visiting each element once, and making a decision about whether to include it or not.

- Previously, we have done so by arbitrarily picking the "first" element in the collection as the one under consideration, and then removed it (expensive) from the collection for future recursive calls.
Optimizing Subset Creation

- The core component of subset generation includes visiting each element once, and making a decision about whether to include it or not.

- Previously, we have done so by arbitrarily picking the "first" element in the collection as the one under consideration, and then removed it (expensive) from the collection for future recursive calls.

- **Key Idea:** Instead of modifying the collection of elements, let's just keep track of our current place in the collection (**index of the element that is currently under consideration**).
void findAllUnbiasedJuriesHelper(Vector<juror>& allCandidates, Vector<juror>& currentJury, int currentBias){
    if (allCandidates.isEmpty()){
        if (currentBias == 0){
            displayJury(currentJury);
        }
    } else {
        juror currentCandidate = allCandidates[0];
        allCandidates.remove(0);

        findAllUnbiasedJuriesHelper(allCandidates, currentJury, currentBias);
        currentJury.add(currentCandidate);
        findAllUnbiasedJuriesHelper(allCandidates, currentJury, currentBias + currentCandidate.bias);
        currentJury.remove(currentJury.size() - 1);
        allCandidates.insert(0, currentCandidate);
    }
}

void findAllUnbiasedJuries(Vector<juror>& allCandidates){
    Vector<juror> jury;
    findAllUnbiasedJuriesHelper(allCandidates, jury, 0);
}
Jury Selection Code v2.0

```cpp
void findAllUnbiasedJuriesHelper(Vector<juror>& allCandidates, Vector<juror>& currentJury,
    int currentBias, int index)
{
    if (allCandidates.isEmpty()){
        if (currentBias == 0)
            displayJury(currentJury);
    } else {
        juror currentCandidate = allCandidates[0];
        allCandidates.remove(0);

        findAllUnbiasedJuriesHelper(allCandidates, currentJury, currentBias);
        currentJury.add(currentCandidate);
        findAllUnbiasedJuriesHelper(allCandidates, currentJury, currentBias + currentCandidate.bias);
        currentJury.remove(currentJury.size() - 1);
        allCandidates.insert(0, currentCandidate);
    }
}

void findAllUnbiasedJuries(Vector<juror>& allCandidates)
{
    Vector<juror> jury;
    findAllUnbiasedJuriesHelper(allCandidates, jury, 0, 0);
}
```
void findAllUnbiasedJuriesHelper(Vector<juror>& allCandidates, Vector<juror>& currentJury, int currentBias, int index) {
    if (index == allCandidates.size()) {
        if (currentBias == 0) {
            displayJury(currentJury);
        }
    } else {
        juror currentCandidate = allCandidates[0];
        allCandidates.remove(0);

        findAllUnbiasedJuriesHelper(allCandidates, currentJury, currentBias);
        currentJury.add(currentCandidate);
        findAllUnbiasedJuriesHelper(allCandidates, currentJury, currentBias + currentCandidate.bias);
        currentJury.remove(currentJury.size() - 1);
        allCandidates.insert(0, currentCandidate);
    }
}

void findAllUnbiasedJuries(Vector<juror>& allCandidates) {
    Vector<juror> jury;
    findAllUnbiasedJuriesHelper(allCandidates, jury, 0, 0);
}
void findAllUnbiasedJuriesHelper(Vector<juror>& allCandidates, Vector<juror>& currentJury, int currentBias, int index) {
    if (index == allCandidates.size()) {
        if (currentBias == 0) {
            displayJury(currentJury);
        }
    } else {
        juror currentCandidate = allCandidates[index];

        findAllUnbiasedJuriesHelper(allCandidates, currentJury, currentBias, index + 1);
        currentJury.add(currentCandidate);
        findAllUnbiasedJuriesHelper(allCandidates, currentJury, currentBias + currentCandidate.bias, index + 1);
        currentJury.remove(currentJury.size() - 1);
    }
}

void findAllUnbiasedJuries(Vector<juror>& allCandidates) {
    Vector<juror> jury;
    findAllUnbiasedJuriesHelper(allCandidates, jury, 0, 0);
}

Jury Selection Code v2.0
Jury Selection Code v2.0

```cpp
void findAllUnbiasedJuriesHelper(Vector<juror>& allCandidates, Vector<juror>& currentJury, int currentBias, int index)
{
    if (index == allCandidates.size())
    {
        if (currentBias == 0)
        {
            displayJury(currentJury);
        }
    }
    else {
        juror currentCandidate = allCandidates[index];

        findAllUnbiasedJuriesHelper(allCandidates, currentJury, currentBias, index + 1);
        currentJury.add(currentCandidate);
        findAllUnbiasedJuriesHelper(allCandidates, currentJury, currentBias + currentCandidate.bias, index + 1);
        currentJury.remove(currentJury.size() - 1);
    }
}

void findAllUnbiasedJuries(Vector<juror>& allCandidates)
{
    Vector<juror> jury;
    findAllUnbiasedJuriesHelper(allCandidates, jury, 0, 0);
}
```

No more expensive addition/removal of possible candidates!
Takeaways

● Being able to enumerate all possible subsets and inspect subsets with certain constraints can be a powerful problem-solving tool.

● Maintaining an index of the current element under consideration for inclusion/exclusion in a collection is the most efficient way to keep track of the decision making process for subset generation.
Jury Selection Code v2.0

```cpp
void findAllUnbiasedJuriesHelper(Vector<juror>& allCandidates, Vector<juror>& currentJury, int currentBias, int index)
{
    if (index == allCandidates.size()){
        if (currentBias == 0){
            displayJury(currentJury);
        }
    } else {
        juror currentCandidate = allCandidates[index];

        findAllUnbiasedJuriesHelper(allCandidates, currentJury, currentBias, index + 1);
        currentJury.add(currentCandidate);
        findAllUnbiasedJuriesHelper(allCandidates, currentJury, currentBias + currentCandidate.bias, index + 1);
        currentJury.remove(currentJury.size() - 1);
    }
}

void findAllUnbiasedJuries(Vector<juror>& allCandidates){
    Vector<juror> jury;
    findAllUnbiasedJuriesHelper(allCandidates, jury, 0, 0);
}
```
Summary
Backtracking recursion:
Exploring many possible solutions
Overall paradigm: choose/explore/unchoose

Two ways of doing it

- **Choose explore undo**
  - Uses pass by reference; usually with large data structures
  - Explicit unchoose step by "undoing" prior modifications to structure
  - E.g. Generating subsets (one set passed around by reference to track subsets)

- **Copy edit explore**
  - Pass by value; usually when memory constraints aren’t an issue
  - Implicit unchoose step by virtue of making edits to copy
  - E.g. Building up a string over time

Three use cases for backtracking

1. Generate/count all solutions (enumeration)
2. Find one solution (or prove existence)
3. Pick one best solution

General examples of things you can do:
- Permutations
- Subsets
- Combinations
- etc.
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.
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.
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.
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.
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.
What’s next?
More Recursive Backtracking