Recursive Backtracking 2

Shreya Shankar

Stanford CS 106B

16 July 2018

Based on slides created by Ashley Taylor, Marty Stepp, Chris Gregg, Keith Schwarz, Julie Zelenski, Jerry Cain, Eric Roberts, Mehran Sahami, Stuart Reges, Cynthia Lee, and others.

Outline

Exhaustive Search

- **Exhaustive search**: exploring every possible combination from a set of choices or values, often implemented recursively

Exhaustive Search

- **Exhaustive search**: exploring every possible combination from a set of choices or values, often implemented recursively
- Often the search space consists of many **decisions**, each of which has several available **choices**

Exhaustive Search

- **Exhaustive search**: exploring every possible combination from a set of choices or values, often implemented recursively
- Often the search space consists of many **decisions**, each of which has several available **choices**
- Example: When enumerating all 5-letter strings, each of the 5 letters is a decision, and each of those decisions has 26 possible choices.

Backtracking

- **Backtracking**: finding solution(s) by trying partial solutions and then abandoning them if they are not suitable

Backtracking

- **Backtracking**: finding solution(s) by trying partial solutions and then abandoning them if they are not suitable
- Think of this as a "brute force" technique because it tries all paths or combinations

Backtracking

```
Explore (decisions):
    If there are no more decisions to make:
        Stop
    Else:
        // Handle one decision here, and do the rest by recursion
        For each available choice C for this decision:
            Choose C
            Explore the remaining decisions that could follow C
            Unchoose C // Backtrack
```

- Key tasks:

Backtracking

```
Explore (decisions):
    If there are no more decisions to make:
        Stop
    Else:
        // Handle one decision here, and do the rest by recursion
        For each available choice C for this decision:
            Choose C
            Explore the remaining decisions that could follow C
            Unchoose C // Backtrack
```

- Key tasks:
  - Figure out appropriate smallest unit of work (decision)

Backtracking

```
Explore (decisions):
    If there are no more decisions to make:
        Stop
    Else:
        // Handle one decision here, and do the rest by recursion
        For each available choice C for this decision:
            Choose C
            Explore the remaining decisions that could follow C
            Unchoose C // Backtrack
```

- Key tasks:
  - Figure out appropriate smallest unit of work (decision)
  - Figure out how to enumerate all possible choices/options for it

Mental Model

- Choosing

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
  2. What are the **choices** for each decision? For loop?

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
  2. What are the **choices** for each decision? For loop?
- Exploring

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
  2. What are the **choices** for each decision? For loop?
- Exploring
  1. How can we represent that choice?

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
  2. What are the **choices** for each decision? For loop?
- Exploring
  1. How can we represent that choice?
  2. Do we need to use a **wrapper** due to extra parameters?

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
  2. What are the **choices** for each decision? For loop?
- Exploring
  1. How can we represent that choice?
  2. Do we need to use a **wrapper** due to extra parameters?
  3. How should we use the return value of the recursive calls?

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
  2. What are the **choices** for each decision? For loop?
- Exploring
  1. How can we represent that choice?
  2. Do we need to use a **wrapper** due to extra parameters?
  3. How should we use the return value of the recursive calls?
- Un-choosing

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
  2. What are the **choices** for each decision? For loop?
- Exploring
  1. How can we represent that choice?
  2. Do we need to use a **wrapper** due to extra parameters?
  3. How should we use the return value of the recursive calls?
- Un-choosing
  1. How do we **un-modify** the parameters? Do we need to explicitly un-modify, or are they copied? Are they un-modified at the same level as they were modified?

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
  2. What are the **choices** for each decision? For loop?
- Exploring
  1. How can we represent that choice?
  2. Do we need to use a **wrapper** due to extra parameters?
  3. How should we use the return value of the recursive calls?
- Un-choosing
  1. How do we **un-modify** the parameters? Do we need to explicitly un-modify, or are they copied? Are they un-modified at the same level as they were modified?
- Base case

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
  2. What are the **choices** for each decision? For loop?
- Exploring
  1. How can we represent that choice?
  2. Do we need to use a **wrapper** due to extra parameters?
  3. How should we use the return value of the recursive calls?
- Un-choosing
  1. How do we **un-modify** the parameters? Do we need to explicitly un-modify, or are they copied? Are they un-modified at the same level as they were modified?
- Base case
  1. What should we do in the **base case** when we're out of decisions?

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
  2. What are the **choices** for each decision? For loop?
- Exploring
  1. How can we represent that choice?
  2. Do we need to use a **wrapper** due to extra parameters?
  3. How should we use the return value of the recursive calls?
- Un-choosing
  1. How do we **un-modify** the parameters? Do we need to explicitly un-modify, or are they copied? Are they un-modified at the same level as they were modified?
- Base case
  1. What should we do in the **base case** when we're out of decisions?
  2. Is there a case for when there **aren't any valid choices left**?

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
  2. What are the **choices** for each decision? For loop?
- Exploring
  1. How can we represent that choice?
  2. Do we need to use a **wrapper** due to extra parameters?
  3. How should we use the return value of the recursive calls?
- Un-choosing
  1. How do we **un-modify** the parameters? Do we need to explicitly un-modify, or are they copied? Are they un-modified at the same level as they were modified?
- Base case
  1. What should we do in the **base case** when we're out of decisions?
  2. Is there a case for when there **aren't any valid choices left**?
  3. Are we avoiding **arms-length** recursion?

Today's lecture

- Continued `diceSum` example from last week

Today's lecture

- Continued `diceSum` example from last week
- More recursive backtracking practice

Today's lecture

- Continued `diceSum` example from last week
- More recursive backtracking practice
- Why "arms-length" recursion is not good, especially in backtracking

Today's lecture

- Continued `diceSum` example from last week
- More recursive backtracking practice
- Why "arms-length" recursion is not good, especially in backtracking
- Backtracking application: permutations

diceSum

diceSum Problem

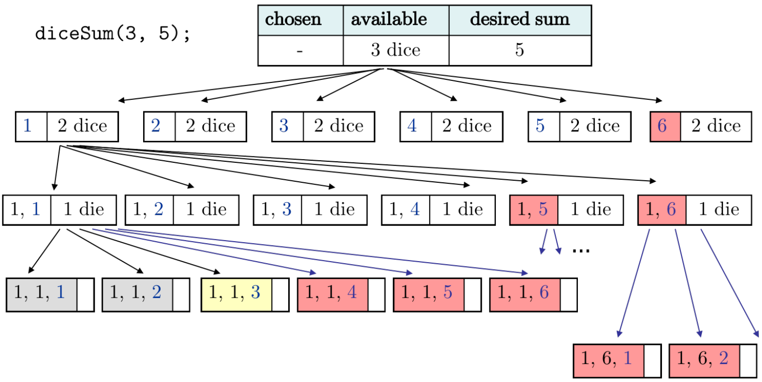
Write a function diceSum that accepts two integer parameters: a number of dice to roll, and a desired sum of all die values. Output all combinations of die values that add up to exactly that sum.

```
void diceSum(int dice, int desiredSum) {
    Vector<int> chosen;
    diceSumHelper(dice, desiredSum, chosen);
}
```

Initial Solution

```
void diceSumHelper(int dice, int desiredSum, Vector<int>&
    chosen) {
    if (dice == 0) {
        if (sumAll(chosen) == desiredSum) {
            cout << chosen << endl; // base case
        }
    } else {
        for (int i = 1; i <= 6; i++) {
            chosen.add(i);                 // choose
            diceSumHelper(dice - 1, desiredSum, chosen);
                // explore
            chosen.remove(chosen.size() - 1); // un-choose
        }
    }
}
```

Wasteful Recursion



Optimizations

- We need not visit every branch of the decision tree

Optimizations

- We need not visit every branch of the decision tree
  - Some branches are clearly not going to lead to success

Optimizations

- We need not visit every branch of the decision tree
  - Some branches are clearly not going to lead to success
  - We can preemptively stop, or **prune**, these branches

Optimizations

- We need not visit every branch of the decision tree
  - Some branches are clearly not going to lead to success
  - We can preemptively stop, or **prune**, these branches
- Inefficiencies in our dice sum algorithm:

Optimizations

- We need not visit every branch of the decision tree
  - Some branches are clearly not going to lead to success
  - We can preemptively stop, or **prune**, these branches
- Inefficiencies in our dice sum algorithm:
  - Sometimes the current sum is already **too high** – even rolling 1 for all remaining dice would exceed the desired sum

Optimizations

- We need not visit every branch of the decision tree
  - Some branches are clearly not going to lead to success
  - We can preemptively stop, or **prune**, these branches
- Inefficiencies in our dice sum algorithm:
  - Sometimes the current sum is already **too high** – even rolling 1 for all remaining dice would exceed the desired sum
  - Sometimes the current sum is already **too low** – even rolling 6 for all remaining dice would exceed the desired sum

Optimizations

- We need not visit every branch of the decision tree
  - Some branches are clearly not going to lead to success
  - We can preemptively stop, or **prune**, these branches
- Inefficiencies in our dice sum algorithm:
  - Sometimes the current sum is already **too high** – even rolling 1 for all remaining dice would exceed the desired sum
  - Sometimes the current sum is already **too low** – even rolling 6 for all remaining dice would exceed the desired sum
  - The code must **re-compute** the sum many times

Optimized diceSum

```
void diceSumHelper(int dice, int sum, int desiredSum,
    Vector<int>& chosen) {
    if (dice == 0) {
        if (sum == desiredSum) {
            cout << chosen << endl; // base case
        }
    } else if (sum + 1*dice <= desiredSum
            && sum + 6*dice >= desiredSum) {
        for (int i = 1; i <= 6; i++) {
            chosen.add(i);  // choose
            diceSumHelper(dice - 1, sum + i, desiredSum,
                chosen); // explore
            chosen.remove(chosen.size() - 1); // un-choose
        }
    }
}
```

Maze

"Escape Maze" exercise

Write a function `escapeMaze(maze, row, col)` that searches for a path out of a given 2-dimensional maze.

- Return `true` if able to escape, or `false` if not

"Escape Maze" exercise

Write a function `escapeMaze(maze, row, col)` that searches for a path out of a given 2-dimensional maze.

- Return `true` if able to escape, or `false` if not
- "Escaping" means exiting the maze boundaries

"Escape Maze" exercise

Write a function `escapeMaze(maze, row, col)` that searches for a path out of a given 2-dimensional maze.

- Return `true` if able to escape, or `false` if not
- "Escaping" means exiting the maze boundaries
- You can move 1 square at a time in any of the 4 directions

"Escape Maze" exercise

Write a function `escapeMaze(maze, row, col)` that searches for a path out of a given 2-dimensional maze.

- Return `true` if able to escape, or `false` if not
- "Escaping" means exiting the maze boundaries
- You can move 1 square at a time in any of the 4 directions
- "Mark" your path along the way

"Escape Maze" exercise

Write a function `escapeMaze(maze, row, col)` that searches for a path out of a given 2-dimensional maze.

- Return `true` if able to escape, or `false` if not
- "Escaping" means exiting the maze boundaries
- You can move 1 square at a time in any of the 4 directions
- "Mark" your path along the way
- "Taint" bad paths that do not work

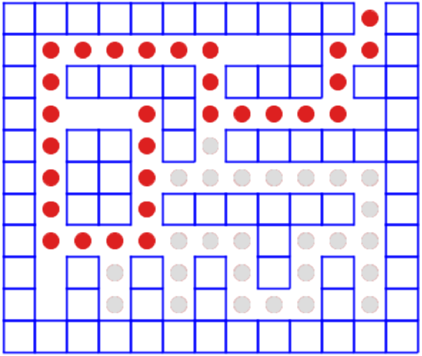
"Escape Maze" exercise

Write a function `escapeMaze(maze, row, col)` that searches for a path out of a given 2-dimensional maze.

- Return `true` if able to escape, or `false` if not
- "Escaping" means exiting the maze boundaries
- You can move 1 square at a time in any of the 4 directions
- "Mark" your path along the way
- "Taint" bad paths that do not work
- Do not explore the same path twice

"Escape Maze" exercise

Write a function `escapeMaze(maze, row, col)` that searches for a path out of a given 2-dimensional maze.



Maze class

```
#include "Maze.h"
```

Member name	Description
`m.inBounds(row, col)`	true if within maze boundaries
`m.isMarked(row, col)`	true if given cell is marked
`m.isOpen(row, col)`	true if given cell is empty (no wall or mark)
`m.isTainted(row, col)`	true if given cell has been tainted
`m.isWall(row, col)`	true if given cell contains a wall
`m.mark(row, col);`	sets given cell to be marked
`m.numRows(), m.numCols()`	returns dimensions of maze
`m.taint(row, col);`	sets given cell to be tainted
`m.unmark(row, col);`	sets given cell to be not marked if marked
`m.untaint(row, col);`	sets given cell to be not tainted if tainted

Mental Model

- Choosing

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
     - *Iterating over possible directions*

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
     - *Iterating over possible directions*
  2. What are the **choices** for each decision? Do we need a loop?

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
     - *Iterating over possible directions*
  2. What are the **choices** for each decision? Do we need a loop?
     - *North, south, west, east*

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
     - *Iterating over possible directions*
  2. What are the **choices** for each decision? Do we need a loop?
     - *North, south, west, east*
- Exploring

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
     - *Iterating over possible directions*
  2. What are the **choices** for each decision? Do we need a loop?
     - *North, south, west, east*
- Exploring
  1. How can we represent that choice? How should we **modify the parameters**?

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
     - *Iterating over possible directions*
  2. What are the **choices** for each decision? Do we need a loop?
     - *North, south, west, east*
- Exploring
  1. How can we represent that choice? How should we **modify the parameters**?
     - *Each direction leads us to a new row, col pair*

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
     - *Iterating over possible directions*
  2. What are the **choices** for each decision? Do we need a loop?
     - *North, south, west, east*
- Exploring
  1. How can we represent that choice? How should we **modify the parameters**?
     - *Each direction leads us to a new row, col pair*
  2. Do we need to use a **wrapper** due to extra parameters?

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
     - *Iterating over possible directions*
  2. What are the **choices** for each decision? Do we need a loop?
     - *North, south, west, east*
- Exploring
  1. How can we represent that choice? How should we **modify the parameters**?
     - *Each direction leads us to a new row, col pair*
  2. Do we need to use a **wrapper** due to extra parameters?
     - *Nope*

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
     - *Iterating over possible directions*
  2. What are the **choices** for each decision? Do we need a loop?
     - *North, south, west, east*
- Exploring
  1. How can we represent that choice? How should we **modify the parameters**?
     - *Each direction leads us to a new row, col pair*
  2. Do we need to use a **wrapper** due to extra parameters?
     - *Nope*
- Base case

Mental Model

- Choosing
    1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
        - *Iterating over possible directions*
    2. What are the **choices** for each decision? Do we need a loop?
        - *North, south, west, east*
- Exploring
    1. How can we represent that choice? How should we **modify the parameters**?
        - *Each direction leads us to a new row, col pair*
    2. Do we need to use a **wrapper** due to extra parameters?
        - *Nope*
- Base case
    1. What should we do in the **base case** when we're out of decisions?

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
     - *Iterating over possible directions*
  2. What are the **choices** for each decision? Do we need a loop?
     - *North, south, west, east*
- Exploring
  1. How can we represent that choice? How should we **modify the parameters**?
     - *Each direction leads us to a new row, col pair*
  2. Do we need to use a **wrapper** due to extra parameters?
     - *Nope*
- Base case
  1. What should we do in the **base case** when we're out of decisions?
     - *Return true or false*

Escape Maze solution

```
bool escapeMaze(Maze& maze, int row, int col) {
    if (!maze.inBounds(row, col)) return true;
    else if (!maze.isOpen(row, col)) return false;
    else {
        // recursive case: try to escape in 4 directions
        maze.mark(row, col);
        if (escapeMaze(maze, row - 1, col)
                || escapeMaze(maze, row + 1, col)
                || escapeMaze(maze, row, col - 1)
                || escapeMaze(maze, row, col + 1)) {
            return true;   // one of the paths worked!
        } else {
            maze.taint(row, col);
            return false; // all 4 paths failed; taint
        }
    }
}
```

"Arms-length" Recursion

"Arms-length" Recursion

- **Arm's length recursion**: a poor style where unnecessary tests are performed before performing recursive calls

"Arms-length" Recursion

- **Arm's length recursion**: a poor style where unnecessary tests are performed before performing recursive calls
- Typically, the tests try to avoid making a call into what would otherwise be a base case

"Arms-length" Recursion

- **Arm's length recursion**: a poor style where unnecessary tests are performed before performing recursive calls
- Typically, the tests try to avoid making a call into what would otherwise be a base case
- Example: `escapeMaze` – our code recursively tries to explore up, down, left, and right. Some of those directions may lead to walls or off the board. Shouldn't we test before making calls in those directions?

"Arms-length" escapeMaze

```
// This code is bad. It uses arm's length recursion.
bool escapeMaze(Maze& maze, int r, int c) {
    maze.mark(row, col);
    // recursive case: check each one by arm's length
    if (maze.inBounds(r-1,c) && maze.isOpen(r-1, c))
        if (escapeMaze(r-1,c)) {return true; }
    if (maze.inBounds(r+1,c) && maze.isOpen(r+1, c))
        if (escapeMaze(r+1,c)) {return true; }
    if (maze.inBounds(r,c-1) && maze.isOpen(r,c-1))
        if (escapeMaze(r,c-1)) {return true; }
    if (maze.inBounds(r,c+1) && maze.isOpen(r,c+1))
        if (escapeMaze(r,c+1)) {return true; }
    maze.taint(row, col);
    return false; // all 4 paths failed; taint
}
```

escapeMaze better solution

```
bool escapeMaze(Maze& maze, int row, int col) {
    if (!maze.inBounds(row, col)) return true;
    else if (!maze.isOpen(row, col)) return false;
    else {
        // recursive case: try to escape in 4 directions
        maze.mark(row, col);
        if (escapeMaze(maze, row - 1, col)
               || escapeMaze(maze, row + 1, col)
               || escapeMaze(maze, row, col - 1)
               || escapeMaze(maze, row, col + 1)) {
            return true;   // one of the paths worked!
        } else {
            maze.taint(row, col);
            return false; // all 4 paths failed; taint
        }
    }
}
```

Permutations

"Permute Vector" exercise

- Write a function `permute` that accepts a Vector of strings as a parameter and outputs all possible rearrangements of the strings in that vector. The arrangements may be output in any order.

{a, b, c, d}	{b, a, c, d}	{c, a, b, d}	{d, a, b, c}
{a, b, d, c}	{b, a, d, c}	{c, a, d, b}	{d, a, c, b}
{a, c, b, d}	{b, c, a, d}	{c, b, a, d}	{d, b, a, c}
{a, c, d, b}	{b, c, d, a}	{c, b, d, a}	{d, b, c, a}
{a, d, b, c}	{b, d, a, c}	{c, d, a, b}	{d, c, a, b}
{a, d, c, b}	{b, d, c, a}	{c, d, b, a}	{d, c, b, a}

"Permute Vector" exercise

- Write a function `permute` that accepts a Vector of strings as a parameter and outputs all possible rearrangements of the strings in that vector. The arrangements may be output in any order.
- Example: if v contains {`"a", "b", "c", "d"`}, your function outputs these permutations:

{a, b, c, d}	{b, a, c, d}	{c, a, b, d}	{d, a, b, c}
{a, b, d, c}	{b, a, d, c}	{c, a, d, b}	{d, a, c, b}
{a, c, b, d}	{b, c, a, d}	{c, b, a, d}	{d, b, a, c}
{a, c, d, b}	{b, c, d, a}	{c, b, d, a}	{d, b, c, a}
{a, d, b, c}	{b, d, a, c}	{c, d, a, b}	{d, c, a, b}
{a, d, c, b}	{b, d, c, a}	{c, d, b, a}	{d, c, b, a}

Examining the problem

- Think of each permutation as a set of choices or **decisions**

Examining the problem

- Think of each permutation as a set of choices or **decisions**
  - Which character do I want to place first?

Examining the problem

- Think of each permutation as a set of choices or **decisions**
  - Which character do I want to place first?
  - Which character do I want to place second?

Examining the problem

- Think of each permutation as a set of choices or **decisions**
  - Which character do I want to place first?
  - Which character do I want to place second?
  - **Solution space**: set of all possible sets of decisions to explore

Examining the problem

- Think of each permutation as a set of choices or **decisions**
  - Which character do I want to place first?
  - Which character do I want to place second?
  - **Solution space**: set of all possible sets of decisions to explore
- We want to generate all possible sequences of decisions

```
for (each possible first letter):
    for (each possible second letter):
        for (each possible third letter):
            ...
                print!
```

Examining the problem

- Think of each permutation as a set of choices or **decisions**
  - Which character do I want to place first?
  - Which character do I want to place second?
  - **Solution space**: set of all possible sets of decisions to explore
- We want to generate all possible sequences of decisions

```
for (each possible first letter):
    for (each possible second letter):
        for (each possible third letter):
            ...
                print!
```

- This is called a **depth-first search**

Mental Model

- Choosing

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
     - *Iterating over strings left in the vector*

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
     - *Iterating over strings left in the vector*
  2. What are the **choices** for each decision? Do we need a loop?

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
     - *Iterating over strings left in the vector*
  2. What are the **choices** for each decision? Do we need a loop?
     - *Strings in the vector; use a for loop*

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
     - *Iterating over strings left in the vector*
  2. What are the **choices** for each decision? Do we need a loop?
     - *Strings in the vector; use a for loop*
- Exploring

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
     - *Iterating over strings left in the vector*
  2. What are the **choices** for each decision? Do we need a loop?
     - *Strings in the vector; use a for loop*
- Exploring
  1. How can we represent that choice? How should we **modify the parameters**?

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
     - *Iterating over strings left in the vector*
  2. What are the **choices** for each decision? Do we need a loop?
     - *Strings in the vector; use a for loop*
- Exploring
  1. How can we represent that choice? How should we **modify the parameters**?
     - *Build up a vector of strings used in our current permutation*

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
     - *Iterating over strings left in the vector*
  2. What are the **choices** for each decision? Do we need a loop?
     - *Strings in the vector; use a for loop*
- Exploring
  1. How can we represent that choice? How should we **modify the parameters**?
     - *Build up a vector of strings used in our current permutation*
  2. Do we need to use a **wrapper** due to extra parameters?

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
     - *Iterating over strings left in the vector*
  2. What are the **choices** for each decision? Do we need a loop?
     - *Strings in the vector; use a for loop*
- Exploring
  1. How can we represent that choice? How should we **modify the parameters**?
     - *Build up a vector of strings used in our current permutation*
  2. Do we need to use a **wrapper** due to extra parameters?
     - *Yes: vector of strings already a part of our permutation*

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
     - *Iterating over strings left in the vector*
  2. What are the **choices** for each decision? Do we need a loop?
     - *Strings in the vector; use a for loop*
- Exploring
  1. How can we represent that choice? How should we **modify the parameters**?
     - *Build up a vector of strings used in our current permutation*
  2. Do we need to use a **wrapper** due to extra parameters?
     - *Yes: vector of strings already a part of our permutation*
- Base case

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
     - *Iterating over strings left in the vector*
  2. What are the **choices** for each decision? Do we need a loop?
     - *Strings in the vector; use a for loop*
- Exploring
  1. How can we represent that choice? How should we **modify the parameters**?
     - *Build up a vector of strings used in our current permutation*
  2. Do we need to use a **wrapper** due to extra parameters?
     - *Yes: vector of strings already a part of our permutation*
- Base case
  1. What should we do in the **base case** when we're out of decisions?

Mental Model

- Choosing
  1. We generally iterate over decisions. What are we iterating over here? The iteration will be done by **recursion**.
     - *Iterating over strings left in the vector*
  2. What are the **choices** for each decision? Do we need a loop?
     - *Strings in the vector; use a for loop*
- Exploring
  1. How can we represent that choice? How should we **modify the parameters**?
     - *Build up a vector of strings used in our current permutation*
  2. Do we need to use a **wrapper** due to extra parameters?
     - *Yes: vector of strings already a part of our permutation*
- Base case
  1. What should we do in the **base case** when we're out of decisions?
     - *Print out permutation*

"Permute Vector" solution

```
// Outputs all permutations of the given vector.
void permute(Vector<string>& v) {
    Vector<string> chosen; permuteHelper(v, chosen);
}
void permuteHelper(Vector<string>& v, Vector<string>&
    chosen) {
    if (v.isEmpty()) cout << chosen << endl;
    else {
        for (int i = 0; i < v.size(); i++) {
            string s = v[i]; v.remove(i);
            chosen.add(s);                     // choose
            permuteHelper(v, chosen);          // explore
            chosen.remove(chosen.size() - 1); // un-choose
            v.insert(i, s);
        }
    }
}
```

Permute a string

- Write a function `permute` that accepts a string as a parameter and outputs all possible rearrangements of the characters in that string. The arrangements may be output in any order.

Permute a string

- Write a function `permute` that accepts a string as a parameter and outputs all possible rearrangements of the characters in that string. The arrangements may be output in any order.
- Example: there are 6 permutations of "cat"

Permute a string – solution

```
// Outputs all permutations of the given string.
void permute(string s) {
    permute(s, "");
}
void permuteHelper(string s, string chosen = "") {
    if (s == "") {
        cout << chosen << endl; // base case: nothing left
    } else {
        // recursive case: choose each possible next letter
        for (int i = 0; i < s.length(); i++) {
            string rest = s.substr(0, i) + s.substr(i + 1);
            permuteHelper(rest, chosen + s[i]); //
                choose/explore
        }
    }
}
```