CS 106B, Lecture 12
Backtracking

Plan for Today

- More practice!
  - Exhaustive Search
  - Backtracking

Exhaustive search

- **Exhaustive search**: Exploring every possible combination from a set of choices. Often implemented recursively.

Exhaustive search

A general pseudo-code algorithm for exhaustive search:

Explore(*decisions*):

- if there are no more decisions to make: Stop.
- else, let's handle one decision ourselves, and the rest by recursion.
  for each available choice *C* for this decision:
  - **Choose** *C* by modifying parameters.
  - **Explore** the remaining decisions that could follow *C*.
  - **Un-choose** *C* by returning parameters to original state (if necessary).

Backtracking

- **Backtracking**: Finding solution(s) by trying all possible paths and then abandoning them if they are not suitable.
  - Idea: it's exhaustive search **with conditions**

Mental Model

- **Choose**: What are the choices for each decision? Do we need a for loop?
- **Explore**: How do we make a choice? How are the parameters changed? Do we need a wrapper function to add more parameters? How should we use the return value of the recursive function?
- **Un-Choose**: How do we revert our choice? Do we need to explicitly change parameters back to original state?
- **Base Case**: What should we do when we are out of decisions to make?

Exercise: Dice roll sum

- 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.

```
diceSum(2, 7);
```

```
{1, 6}
{2, 5}
{3, 4}
{4, 3}
{5, 2}
{6, 1}
```

```
diceSum(3, 7);
```

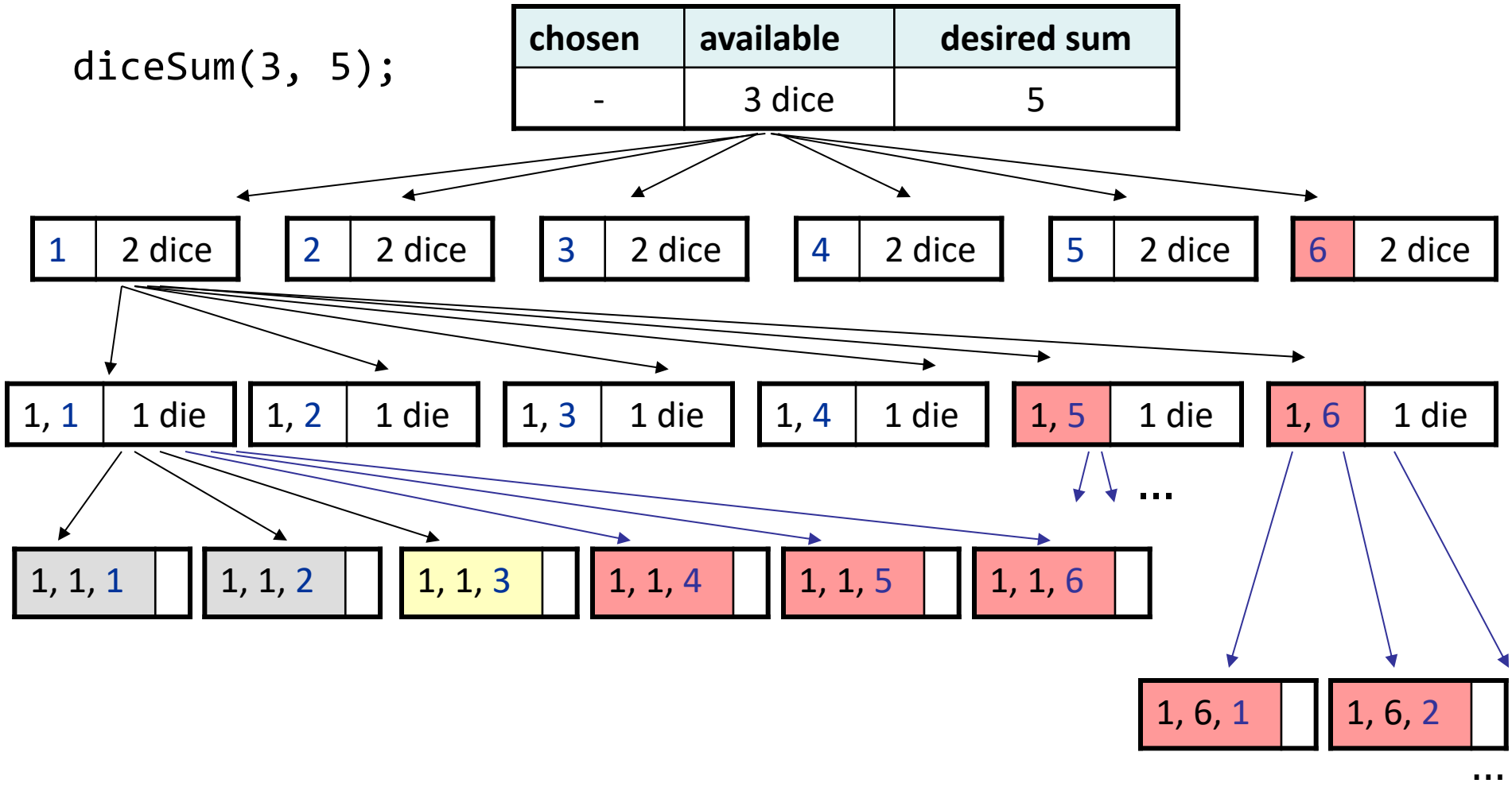
```
{1, 1, 5}
{1, 2, 4}
{1, 3, 3}
{1, 4, 2}
{1, 5, 1}
{2, 1, 4}
{2, 2, 3}
{2, 3, 2}
{2, 4, 1}
{3, 1, 3}
{3, 2, 2}
{3, 3, 1}
{4, 1, 2}
{4, 2, 1}
{5, 1, 1}
```

Initial solution

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

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
        }
    }
}

int sumAll(const Vector<int>& v) {   // adds the values in given vector
    int sum = 0;
    for (int k : v) { sum += k; }
    return sum;
}
```

Wasteful decision tree

```
diceSum(3, 5);
```



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.
    - (1+1+1 = …, 1+1+2 = …, 1+1+3 = …, 1+1+4 = …, …)

diceSum solution

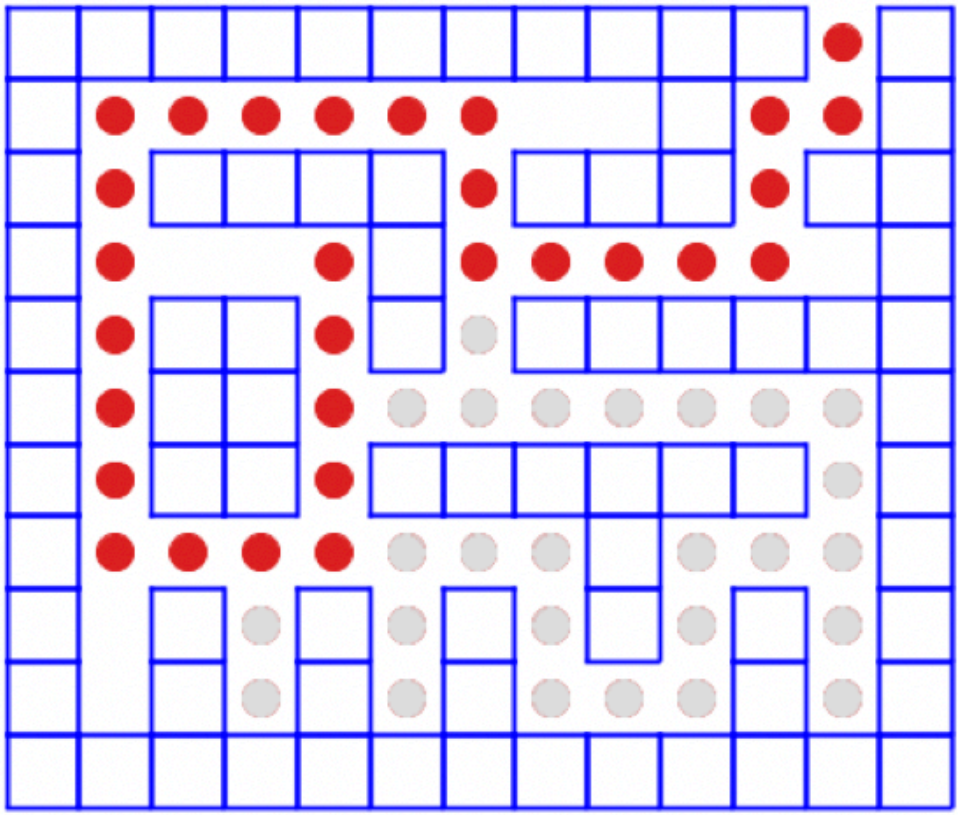
```
void diceSum(int dice, int desiredSum) {
    Vector<int> chosen;
    diceSumHelper(dice, desiredSum, chosen);
}

void diceSumHelper(int dice, int desiredSum, Vector<int>& chosen) {
    if (dice == 0 && desiredSum == 0) {
        cout << chosen << endl;
    } else if (dice > 0 && (dice <= desiredSum && desiredSum <= dice*6)) {
        for (int i = 1; i <= 6; i++) {
            chosen.add(i);
            diceSum(dice - 1, desiredSum - i, chosen);
            chosen.removeBack();
        }
    }
}
```

Exercise: Escape Maze

- 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, false if not.
  - 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.

Exercise: Escape Maze

- 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, false if not.



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

- **Choose**: What decisions do we have to make? What are our choices?
- **Explore**: How should we modify our parameters after making a choice?
- **Un-Choose**: How do we revert our choice?
- **Base Case**: What should we do when we are out of decisions to make?

Mental Model

- **Choose**: What decisions do we have to make? What are our choices?
  - *Possible directions to take. North, south, west, east*
- **Explore**: How should we modify our parameters after making a choice?
  - *Each direction leads to a new (row, col)*
- **Un-Choose**: How do we revert our choice?
  - *(row, col) is copied, so nothing!*
- **Base Case**: What should we do when we are out of decisions to make?
  - *Return true or false*

escapeMaze 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
                }
        }
}
```

Announcements

- Homework 3 due on Wednesday at **5PM**

- **Midterm next Wednesday,** 7/24 7-9PM
  - Logistics are released on the website
    - http://web.stanford.edu/class/archive/cs/cs106b/cs106b.1198/exams/midterm.html
  - Practice midterm will be released soon

Arms-Length Recursion

- **Arm's Length Recursion:** Unnecessary tests are performed before performing recursive calls. Considered bad style.
- 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 arms-length recursion
bool escapeMaze(Maze& maze, int row, int col) {
        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
}
```

Permutations

- Write a function `permute` that accepts a `Vector` of strings as a parameter and outputs all possible rearrangements of the strings in that vector.
  - For example, if v contains {"a", "b", "c", "d"}, output the below 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}

Permutations

- Each permutation is a set of decisions.
  - Which character do I want to place first?
  - Which character do I want to place second?

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

Mental Model

- **Choose**: What decisions do we have to make? What are our choices?
- **Explore**: How should we modify our parameters after making a choice?
- **Un-Choose**: How do we revert our choice?
- **Base Case**: What should we do when we are out of decisions to make?

Mental Model

- **Choose**: What decisions do we have to make? What are our choices?
  - *Possible strings to use. Any of remaining strings in the vector*
- **Explore**: How should we modify our parameters after making a choice?
  - *Build up a vector of strings used so far. We will need a wrapper function for this extra parameter*
- **Un-Choose**: How do we revert our choice?
  - *Update the vector of strings used so far*
- **Base Case**: What should we do when we are out of decisions to make?
  - *Print out the permutation*

permute 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.

permute 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
        }
    }
}
```