CS 106X, Lecture 11
Recursive Backtracking 2

reading:

Programming Abstractions in C++, Chapter 9

Plan For Today

- **Recap:** what is recursive backtracking?
- **Recap:** dice rolls
- **Recap:** subsets
- Selection Problems: Subsets, Combinations and Permutations
- **Break:** Announcements
- Practicing types of backtracking problems
  - *Find all solutions*
  - *Find any solution*
  - *How many solutions are there?*
  - *Does a solution exist?*
  - *Find the best solution*

Plan For Today

- **Recap:** what is recursive backtracking?
- **Recap:** dice rolls
- **Recap:** subsets
- Selection Problems: Subsets, Combinations and Permutations
- **Break:** Announcements
- Practicing types of backtracking problems
  - *Find all solutions*
  - *Find any solution*
  - *How many solutions are there?*
  - *Does a solution exist?*
  - *Find the best solution*

Recursive Backtracking

- **Recursive Backtracking:** using recursion to explore solutions to a problem and abandoning them if they are not suitable.
  - *Find all solutions*
  - *Find any solution*
  - *How many solutions are there?*
  - *Does a solution exist?*
  - *Find the best solution*
- Applications:
  - Puzzle solving (Sudoku, Crosswords, etc.)
  - Game playing (Chess, Solitaire, etc.)
  - Constraint satisfaction problems (scheduling, matching, etc.)

The Backtracking Checklist

- ❑ **Find what choice(s) we have at each step.** What different options are there for the next step?

For each valid choice:

- ❑ **Make it and explore recursively.** Pass the information for a choice to the next recursive call(s).
- ❑ **Undo it after exploring.** Restore everything to the way it was before making this choice.

- ❑ **Find our base case(s).** What should we do when we are out of decisions?

Plan For Today

- **Recap:** what is recursive backtracking?
- **Recap:** dice rolls
- **Recap:** subsets
- Selection Problems: Subsets, Combinations and Permutations
- **Break:** Announcements
- Practicing types of backtracking problems
  - *Find all solutions*
  - *Find any solution*
  - *How many solutions are there?*
  - *Does a solution exist?*
  - *Find the best solution*

Let's Roll the Dice

- Write a recursive function **`diceRoll`** that accepts an integer representing a number of 6-sided dice to roll, and output all possible combinations of values that could appear on the dice.

`diceRoll(2);`

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

`diceRoll(3);`

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

The Backtracking Checklist

- ❑ **Find what choice(s) we have at each step.** What different options are there for the next step?

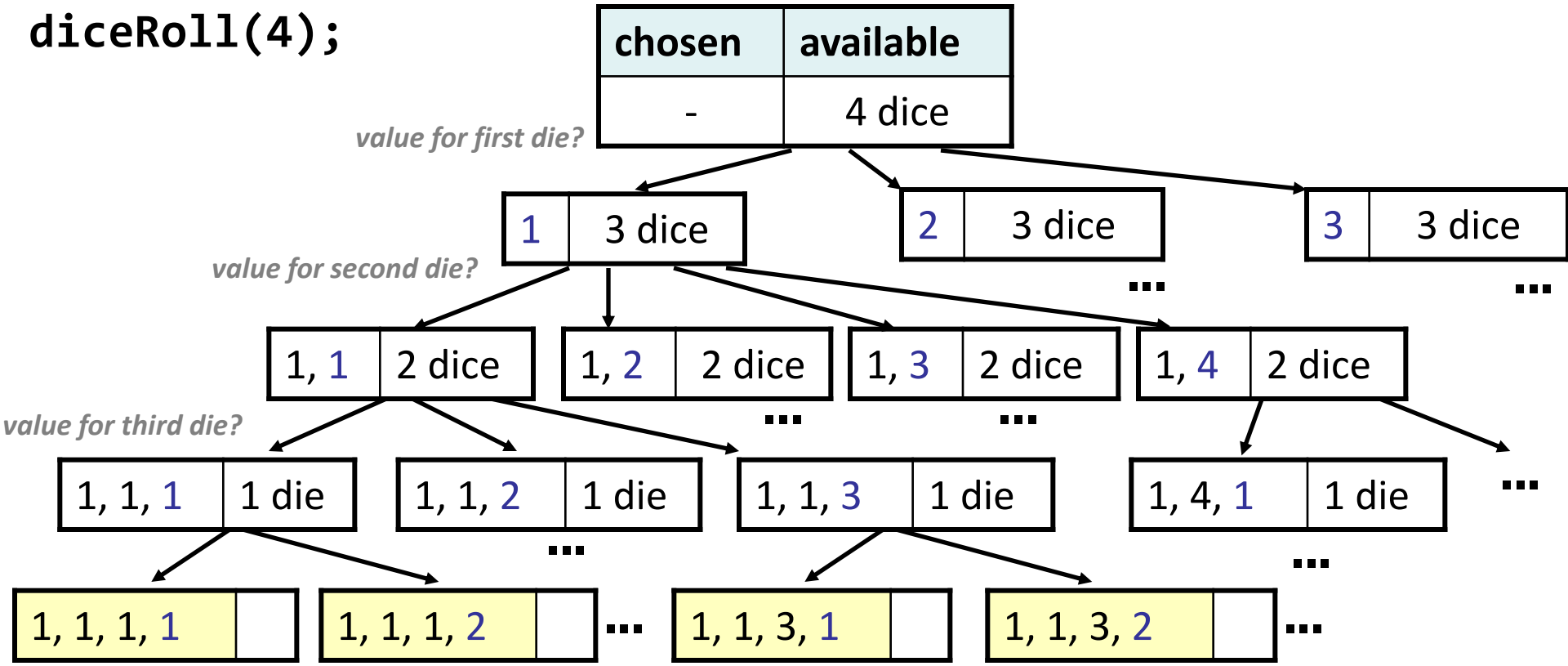
For each valid choice:

- ❑ **Make it and explore recursively.** Pass the information for a choice to the next recursive call(s).
- ❑ **Undo it after exploring.** Restore everything to the way it was before making this choice.

- ❑ **Find our base case(s).** What should we do when we are out of decisions?

Decision Tree

```
diceRoll(4);
```



diceRolls Pseudocode

```
function diceRolls(dice):
    if dice == 0:
        nothing to do.
    else:
        // handle all roll values for a single die; let recursion do the rest.
        for each die value i in range [1..6]:
            choose that the current die will have value i.
            diceRolls(dice-1).          // explore the remaining dice.
            un-choose the value i.
```

- How do we keep track of our choices?

diceRolls Solution

```
// Prints all possible outcomes of rolling the given
// number of six-sided dice in {#, #, #} format.
void diceRolls(int dice) {
    Vector<int> chosen;
    diceRollHelper(dice, chosen);
}

// private recursive helper to implement diceRolls logic
void diceRollHelper(int dice, Vector<int>& chosen) {
    if (dice == 0) {
        cout << chosen << endl;                 // base case
    } else {
        for (int i = 1; i <= 6; i++) {
            chosen.add(i);                      // choose
            diceRollHelper(dice - 1, chosen);   // explore
            chosen.remove(chosen.size() - 1);   // un-choose
        }
    }
}
```

Rolling the Dice, Round 2

- Write a function **`diceSum`** similar to `diceRoll`, but it also accepts a desired sum and prints only combinations that add up to exactly that sum.

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

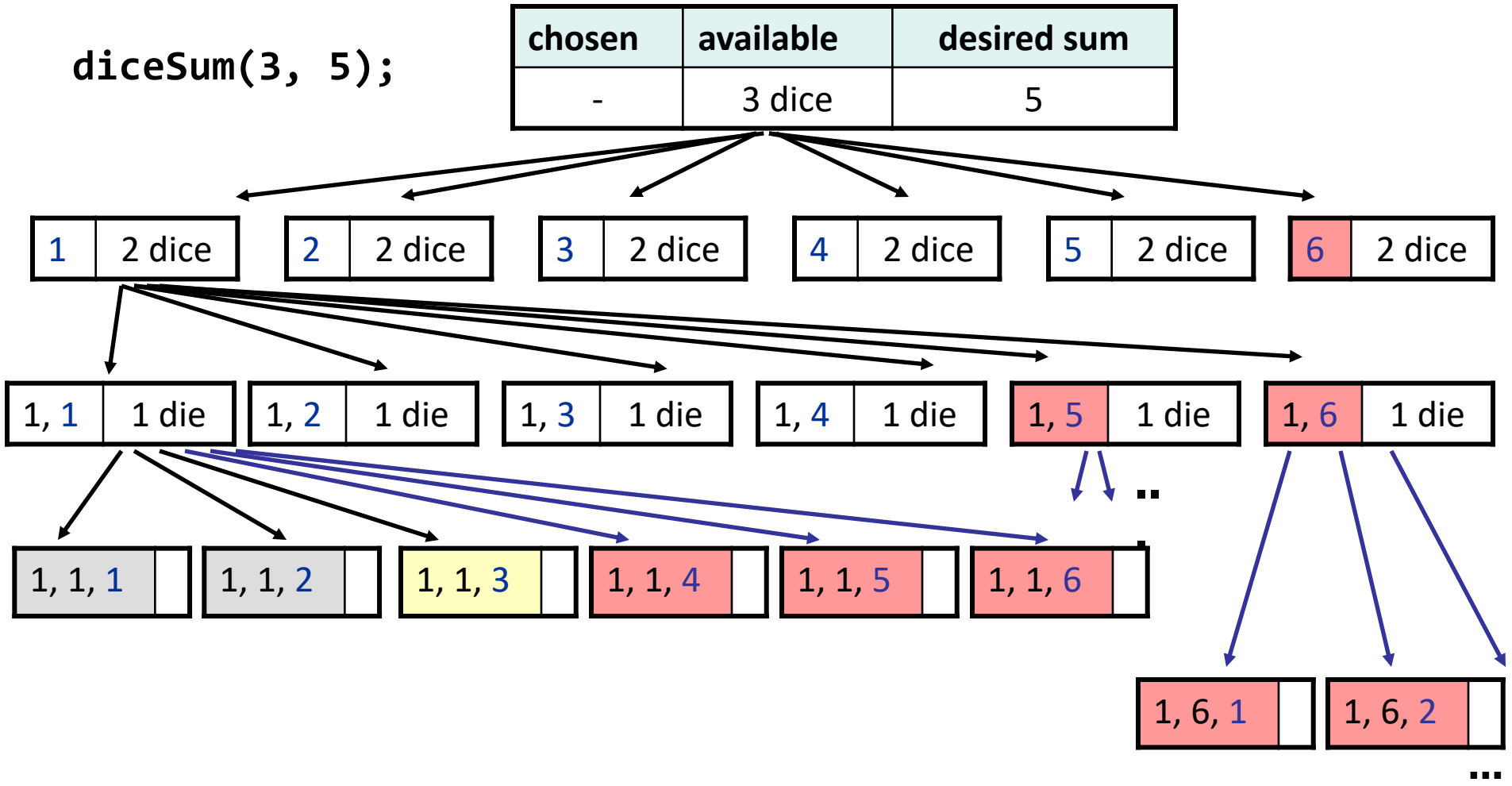
```
{1, 6}
{2, 5}
{3, 4}
{4, 3}
{5, 2}
{6, 1}
```

```
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}
```

Pruning the Tree

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



diceSum Solution

```
void diceSum(int dice, int desiredSum) {
    Vector<int> chosen;
    diceSumHelper(dice, 0, desiredSum, chosen);
}

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
        }
    }
}
```

Plan For Today

- **Recap:** what is recursive backtracking?
- **Recap:** dice rolls
- **Recap:** subsets
- Selection Problems: Subsets, Combinations and Permutations
- **Break:** Announcements
- Practicing types of backtracking problems
  - *Find all solutions*
  - *Find any solution*
  - *How many solutions are there?*
  - *Does a solution exist?*
  - *Find the best solution*

Exercise: Subsets

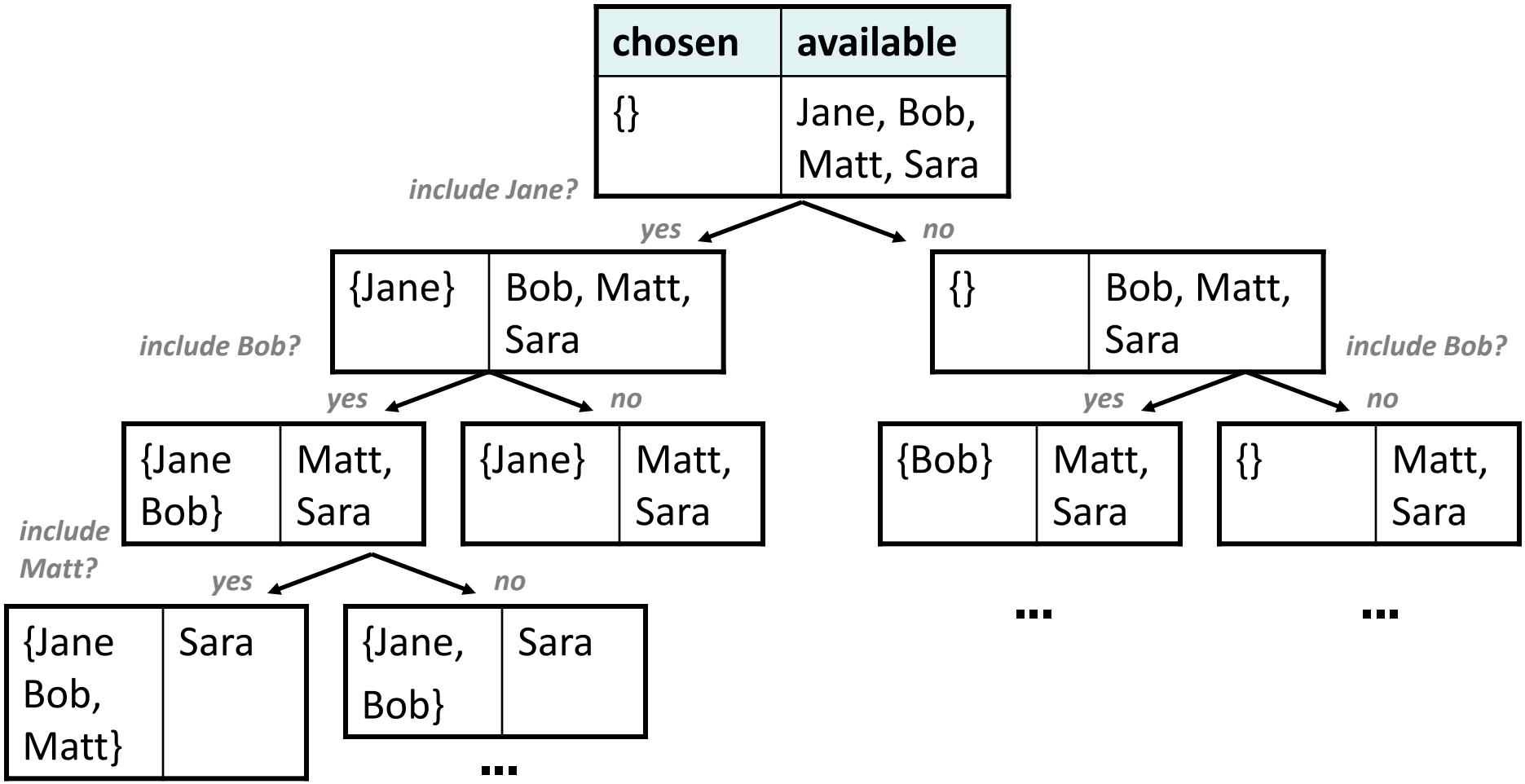
printSubVectors

- Write a function **`subsets`** that finds every possible sub-list of a given set. A subset of a set *V* contains ≥ 0 of *V*'s elements.
  - Example: if *V* is `{Jane, Bob, Matt, Sara}`, then the call of **`subsets`**`(V);` prints:

```
{Jane, Bob, Matt, Sara}
{Jane, Bob, Matt}
{Jane, Bob, Sara}
{Jane, Bob}
{Jane, Matt, Sara}
{Jane, Matt}
{Jane, Sara}
{Jane}
{Bob, Matt, Sara}
{Bob, Matt}
{Bob, Sara}
{Bob}
{Matt, Sara}
{Matt}
{Sara}
{}
```

  - You can print the subsets out in any order, one per line.

Decision Tree



- Each decision is: "Include Jane or not?" ... "Include Bob or not?" ...
  - The **order** of people chosen does not matter; only the **membership**.

The Backtracking Checklist

- ❑ **Find what choice(s) we have at each step.** What different options are there for the next step?

For each valid choice:

- ❑ **Make it and explore recursively.** Pass the information for a choice to the next recursive call(s).
- ❑ **Undo it after exploring.** Restore everything to the way it was before making this choice.

- ❑ **Find our base case(s).** What should we do when we are out of decisions?

subsets Solution

```
void subsets(const Set<string>& masterSet) {
    Set<string> chosen;
    listSubsetsRec(masterSet, chosen);
}

void listSubsetsRec(const Set<string>& masterSet, const Set<string>& used) {
    if (masterSet.isEmpty()) {
        cout << used << endl;
    } else {
        string element = masterSet.first();

        listSubsetsRec(masterSet - element, used);           // Without
        listSubsetsRec(masterSet - element, used + element); // With
    }
}
```

Types of Selection Problems

- **Subsets** are zero or more elements from a group of elements.
- **Combinations** are ways you can choose exactly K elements from a group of elements.
- **Permutations** are ways you can order a group of elements.

Finding Subsets

```
void subSets(const Set<string>& masterSet) {
    Set<string> chosen;
    listSubsetsRec(masterSet, chosen);
}

void listSubsetsRec(const Set<string>& masterSet, const Set<string>& used) {
    if (masterSet.isEmpty()) {
        cout << used << endl;
    } else {
        string element = masterSet.first();

        listSubsetsRec(masterSet - element, used);           // Without
        listSubsetsRec(masterSet - element, used + element); // With
    }
}
```

How could we modify this to find all combinations of size K?

Finding Combinations

```
void subSets(const Set<string>& masterSet, int size) {
    Set<string> chosen;
    listSubsetsRec(masterSet, size, chosen);
}

void listSubsetsRec(const Set<string>& masterSet, int size, const
  Set<string>& used) {
    if (size == 0) {
        cout << used << endl;
    } else if (masterSet.size() > 0) {
        string element = masterSet.first();

        listSubsetsRec(masterSet - element, size, used);          // Without
        listSubsetsRec(masterSet - element, size-1, used + element); // With
    }
}
```

Plan For Today

- **Recap:** what is recursive backtracking?
- **Recap:** dice rolls
- **Recap:** subsets
- Selection Problems: Subsets, Combinations and Permutations
- **Break:** Announcements
- Practicing types of backtracking problems
  - *Find all solutions*
  - *Find any solution*
  - *How many solutions are there?*
  - *Does a solution exist?*
  - *Find the best solution*

Types of Selection Problems

- **Subsets** are zero or more elements from a group of elements.
- **Combinations** are ways you can choose exactly K elements from a group of elements.
- **Permutations** are ways you can order a group of elements.

Grab the Popcorn 

- Write a function **`printMovieOrders`** that accepts a `Vector` of movie names (Strings) as a parameter and outputs all possible orders in which we could watch those movies. The arrangements may be output in any order.
  - Example: if v contains `{"Up", "Argo", "Black Panther", "Inside Out"}`, your function outputs permutations like:

{“Up”, “Argo”, “Black Panther”, “Inside Out”}	“Inside Out”, “Argo”, “Black Panther”, “Up”}
{“Up”, “Argo”, “Inside Out”, “Black Panther”}	“Inside Out”, “Argo”, “Up”, “Black Panther”}
{“Up”, “Black Panther”, “Argo”, “Inside Out”}	“Inside Out”, “Black Panther”, “Up”, “Argo”}
{“Up”, “Black Panther”, “Inside Out”, “Argo”}	“Inside Out”, “Black Panther”, “Argo”, “Up”}
...	...

The Backtracking Checklist

- ❑ **Find what choice(s) we have at each step.** What different options are there for the next step?

For each valid choice:

- ❑ **Make it and explore recursively.** Pass the information for a choice to the next recursive call(s).
- ❑ **Undo it after exploring.** Restore everything to the way it was before making this choice.

- ❑ **Find our base case(s).** What should we do when we are out of decisions?

Movie Permutations

```
void printMovieOrders(Vector<string>& allMovies,
                      Vector<string>& chosen) {
    if (allMovies.isEmpty()) {
        cout << chosen << endl;   // base case
    } else {
        for (int i = 0; i < allMovies.size(); i++) {
            string currMovie = allMovies[i];
            allMovies.remove(i);
            chosen.add(currMovie);                  // choose
            printMovieOrders(allMovies, chosen);    // explore
            chosen.remove(chosen.size() - 1);     // un-choose
            allMovies.insert(i, currMovie);
        }
    }
}
```

Movie Permutations

```
// Outputs all permutations of the given list of movies.
void printMovieOrders(const Vector<string>& allMovies) {
    Vector<string> chosen;
    Vector<string> moviesCopy = allMovies;
    printMovieOrders(moviesCopy, chosen);
}
```

Find/Print All Solutions

- **Base case:** is it a valid solution? If so, print it (or add to set of found solutions). If not valid, don't.
- **Recursive step:** Make all recursive calls. If you are returning a collection, add each recursive result to the collection.

Plan For Today

- **Recap:** what is recursive backtracking?
- **Recap:** dice rolls
- **Recap:** subsets
- Selection Problems: Subsets, Combinations and Permutations
- **Break:** Announcements
- Practicing types of backtracking problems
  - *Find all solutions*
  - *Find any solution*
  - *How many solutions are there?*
  - *Does a solution exist?*
  - *Find the best solution*

CS + Social Good

Plan For Today

- **Recap:** what is recursive backtracking?
- **Recap:** dice rolls
- **Recap:** subsets
- Selection Problems: Subsets, Combinations and Permutations
- **Break:** Announcements
- Practicing types of backtracking problems
  - *Find all solutions*
  - *Find any solution*
  - *How many solutions are there?*
  - *Does a solution exist?*
  - *Find the best solution*

Plan For Today

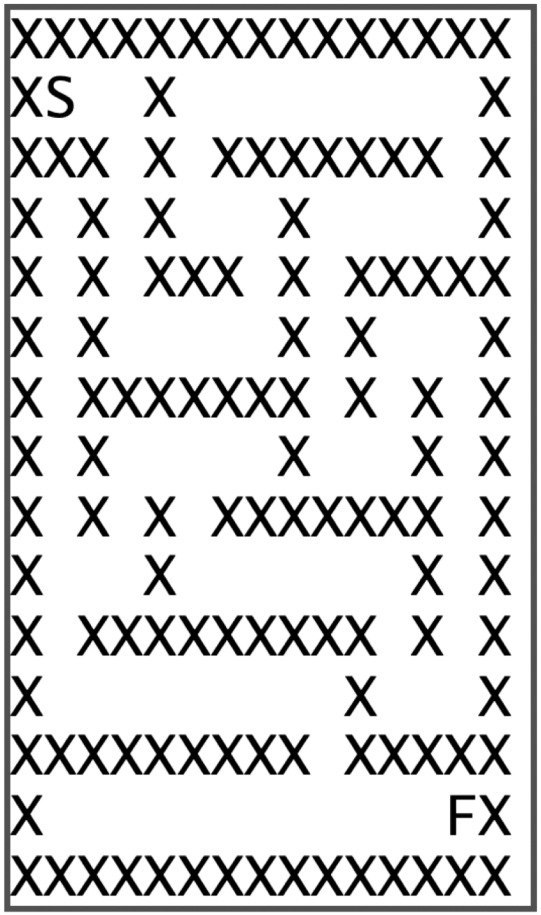
- **Recap:** what is recursive backtracking?
- **Recap:** dice rolls
- **Recap:** subsets
- Selection Problems: Subsets, Combinations and Permutations
- Announcements
- Practicing types of backtracking problems
  - *Find all solutions*
  - *Find any solution*
  - *How many solutions are there?*
  - *Does a solution exist?*
  - *Find the best solution*

Maze Solving

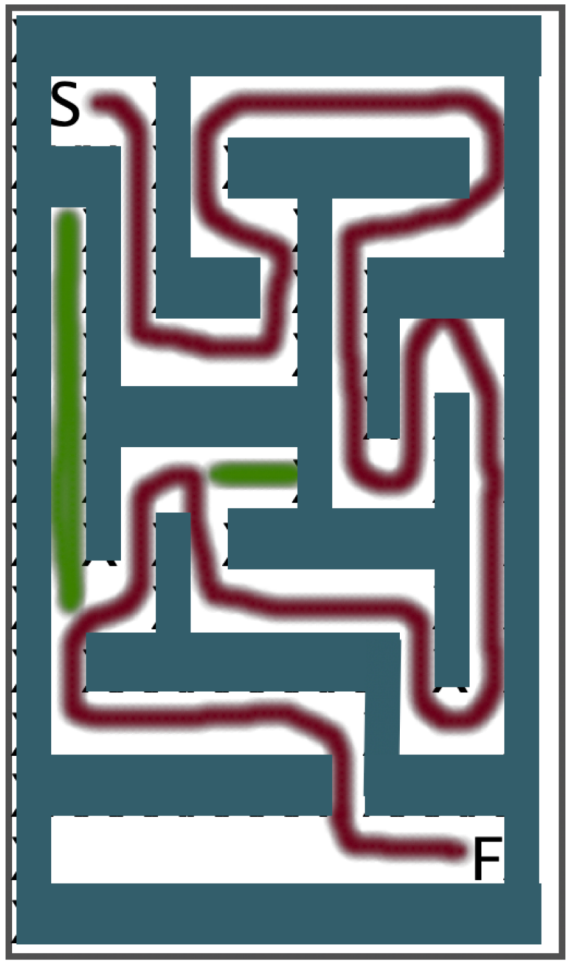
https://www.reddit.com/r/funny/comments/lejyk/billy_mays_maize_maze/

Maze Solving

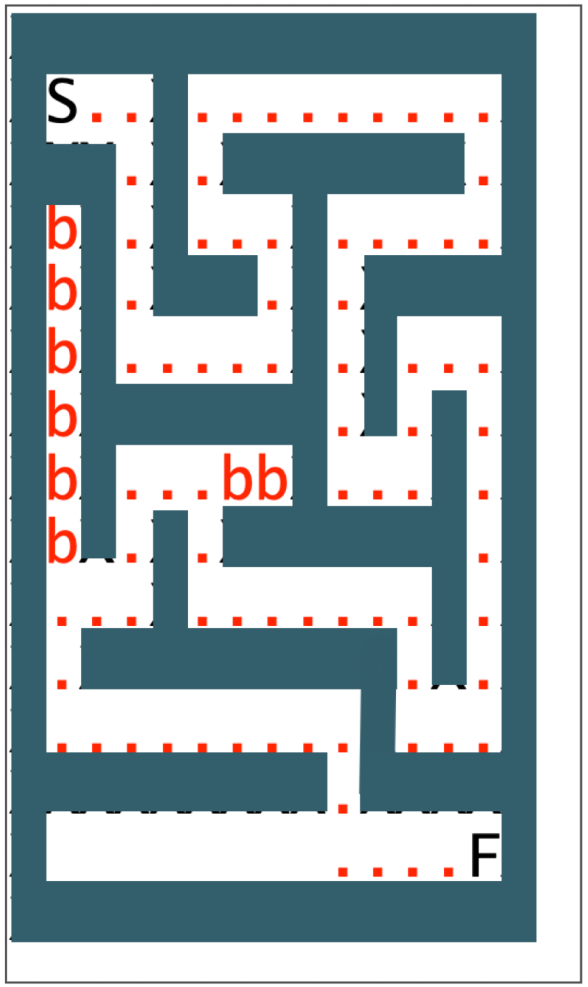
- The code for today's class includes a text-based recursive maze creator and solver.
- The mazes look like the one to the right
  - There is a Start (marked with an "S") and a Finish (marked with an "F").
  - The Xs represent walls, and the spaces represent paths to walk through the maze.



Maze Solving



- The program will put dots in the correct positions.
- But, it will also put lowercase b's when it goes in the wrong direction and has to backtrack.



The Backtracking Checklist

- ❑ **Find what choice(s) we have at each step.** What different options are there for the next step?

For each valid choice:

- ❑ **Make it and explore recursively.** Pass the information for a choice to the next recursive call(s).
- ❑ **Undo it after exploring.** Restore everything to the way it was before making this choice.

- ❑ **Find our base case(s).** What should we do when we are out of decisions?

Maze Solving

```
bool solveMazeRecursive(int row, int col, Grid<int>& maze) {
    if (maze[row][col] == 'X') return false;
    if (maze[row][col] == '.') return false;
    if (maze[row][col] == 'F') return true;

    maze[row][col] = '.';

    // north
    if (solveMazeRecursive(row-1,col,maze)) return true;
    // east
    if (solveMazeRecursive(row,col+1,maze)) return true;
    // south
    if (solveMazeRecursive(row+1,col,maze)) return true;
    // west
    if (solveMazeRecursive(row,col-1,maze)) return true;

    maze[row][col] = 'b';
    return false;
}
```

Maze Solving

```
bool solveMazeRecursive(int row, int col, Grid<int>& maze) {
    if (maze[row][col] == 'X') return false;
    if (maze[row][col] == '.') return false;
    if (maze[row][col] == 'F') return true;

    maze[row][col] = '.';

    // north
    return solveMazeRecursive(row-1,col,maze)); // bad idea ☹
    // east
    if (solveMazeRecursive(row,col+1,maze)) return true;
    // south
    if (solveMazeRecursive(row+1,col,maze)) return true;
    // west
    if (solveMazeRecursive(row,col-1,maze)) return true;

    maze[row][col] = 'b';
    return false;
}
```

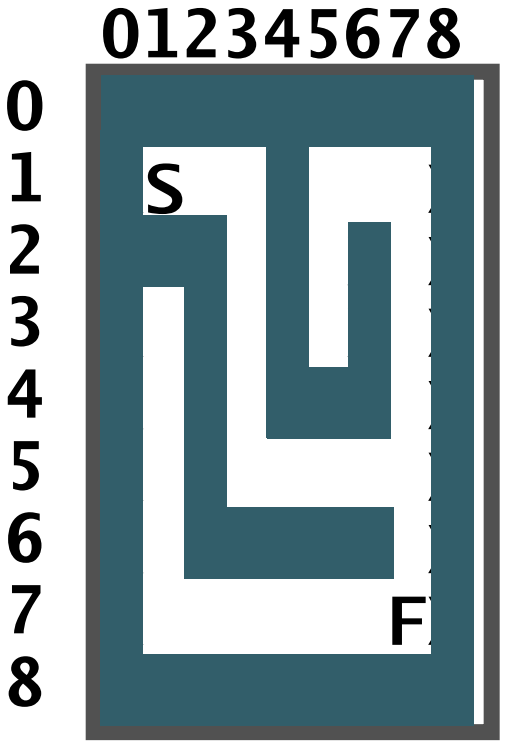
Maze Solving

```
bool solveMazeRecursive(int row, int col, Grid<int>& maze) {
    if (maze[row][col] == 'X') return false;
    if (maze[row][col] == '.') return false;
    if (maze[row][col] == 'F') return true;

    maze[row][col] = '.';

    // north
    return solveMazeRecursive(row-1,col,maze)); // bad idea ☹
    // east
    if (solveMazeRecursive(row,col+1,maze)) return true;
    // south
    if (solveMazeRecursive(row+1,col,maze)) return true;
    //
    if

    ma
    re
}
```

Just because one recursive exploration fails doesn't mean we should give up hope! Have faith that another exploration might find a solution, and only return false if you've tried everything.

Maze Solving

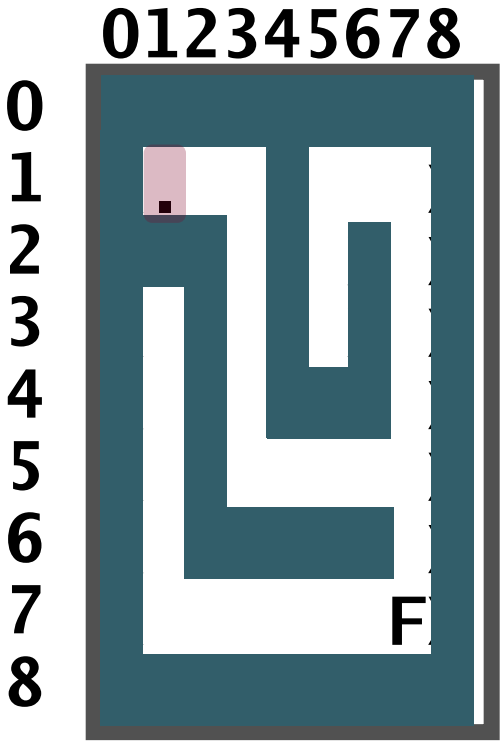
Let's walk through what our backtracking algorithm is doing.



Maze Solving

Let's walk through what our backtracking algorithm is doing.

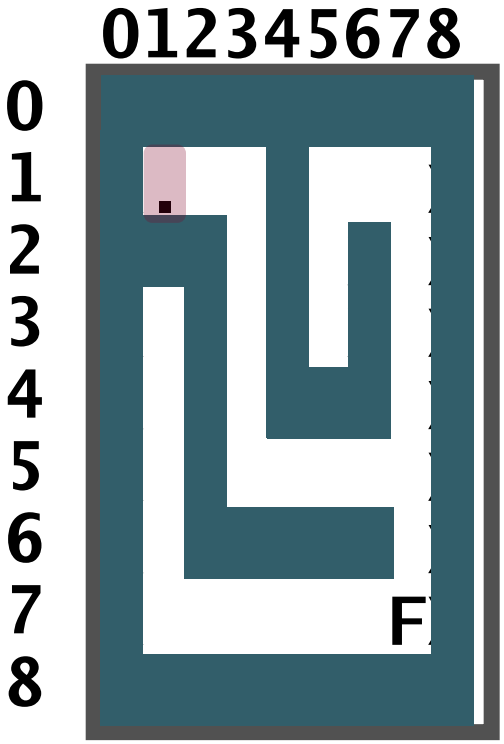
- Start: **row=1, col=1**. Marking with period (.)



Maze Solving

Let's walk through what our backtracking algorithm is doing.

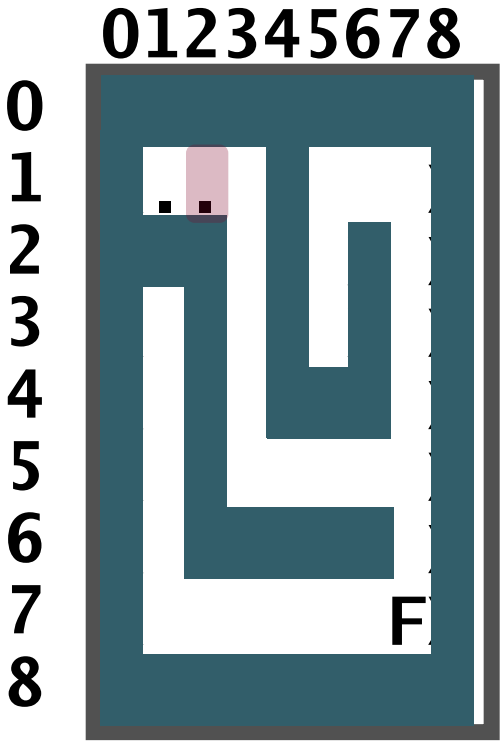
- Start: **row=1, col=1**. Marking with period (.)
- We have to try all paths, N/E/S/W.
- Trying north, row=0 and col=1. Wall! Back to row=1, col=1.



Maze Solving

Let's walk through what our backtracking algorithm is doing.

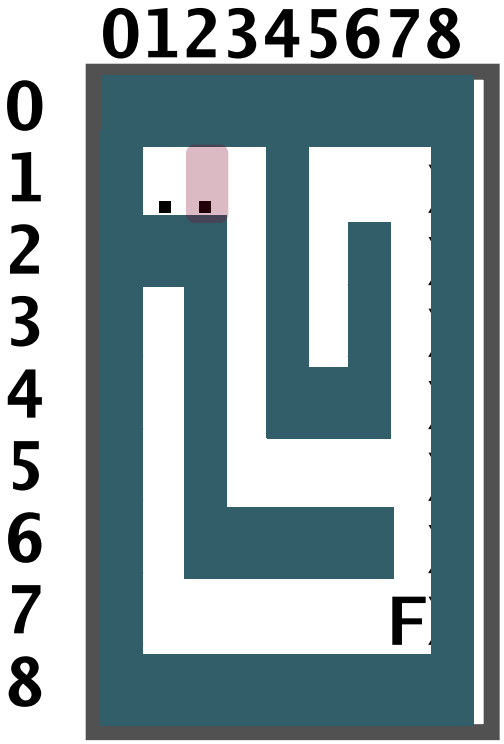
- Start: **row=1, col=1**. Marking with period (.)
- We have to try all paths, N/E/S/W.
- Trying north, row=0 and col=1. Wall! Back to row=1, col=1.
- Trying east, row=1 and col=2. Marking with period (.)



Maze Solving

Let's walk through what our backtracking algorithm is doing.

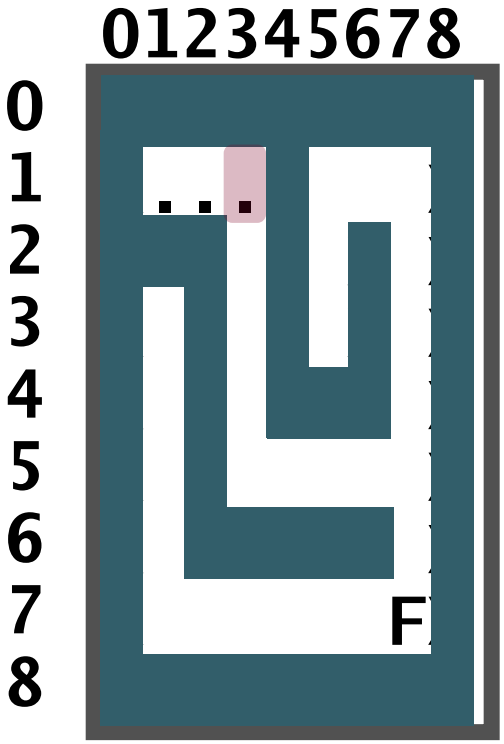
- Start: **row=1, col=1**. Marking with period (.)
- We have to try all paths, N/E/S/W.
- Trying north, row=0 and col=1. Wall! Back to row=1, col=1.
- Trying east, row=1 and col=2. Marking with period (.)
- Trying north, row=0 and col=2. Wall! Back to row=1, col=2.



Maze Solving

Let's walk through what our backtracking algorithm is doing.

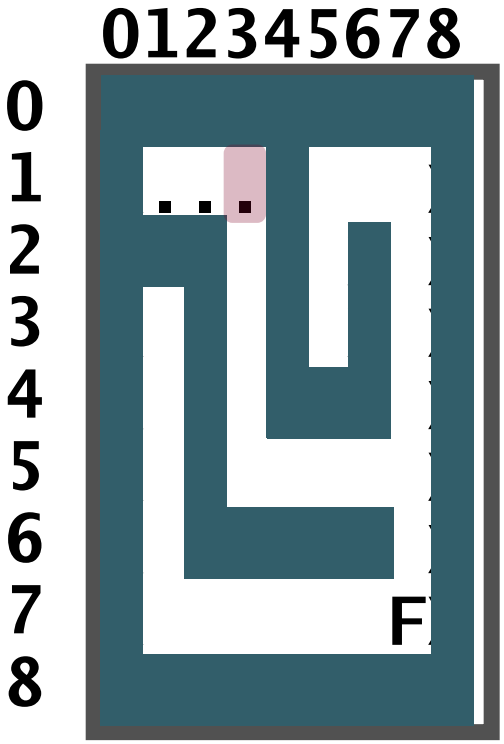
- Start: **row=1, col=1**. Marking with period (.)
- We have to try all paths, N/E/S/W.
- Trying north, row=0 and col=1. Wall! Back to row=1, col=1.
- Trying east, row=1 and col=2. Marking with period (.)
- Trying north, row=0 and col=2. Wall! Back to row=1, col=2.
- Trying east, row=1 and col=3. Marking with period (.)



Maze Solving

Let's walk through what our backtracking algorithm is doing.

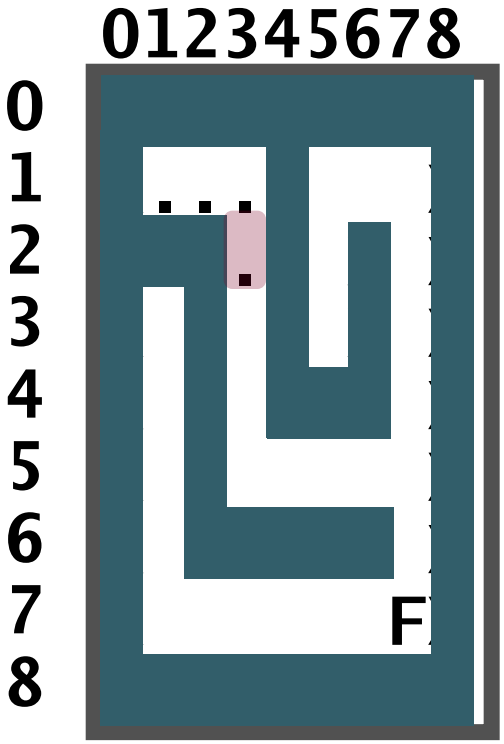
- Start: **row=1, col=1**. Marking with period (.)
- …
- Trying east, row=1 and col=3. Marking with period (.)
- Trying north, row=0 and col=3. Wall! Back to row=1, col=3.
- Trying east, row=1 and col=4. Wall! Back to row=1, col=3.



Maze Solving

Let's walk through what our backtracking algorithm is doing.

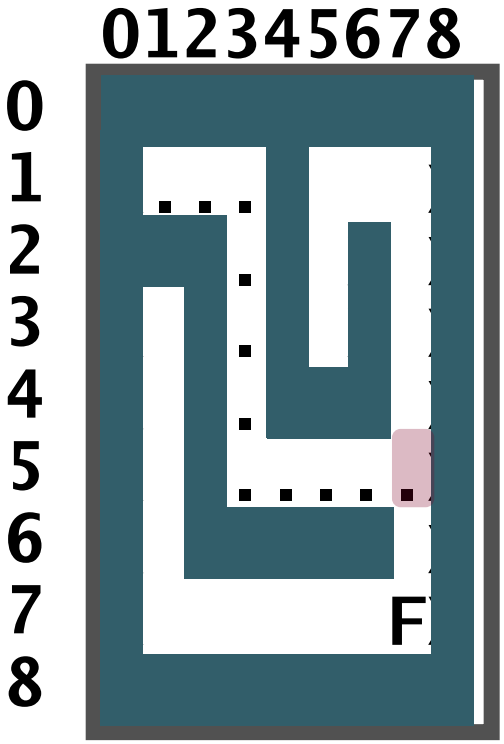
- Start: **row=1, col=1**. Marking with period (.)
- …
- Trying east, row=1 and col=3. Marking with period (.)
- Trying north, row=0 and col=3. Wall! Back to row=1, col=3.
- Trying east, row=1 and col=4. Wall! Back to row=1, col=3.
- Trying south, row=2 and col=3. Marking with period (.)



Maze Solving

Let's walk through what our backtracking algorithm is doing.

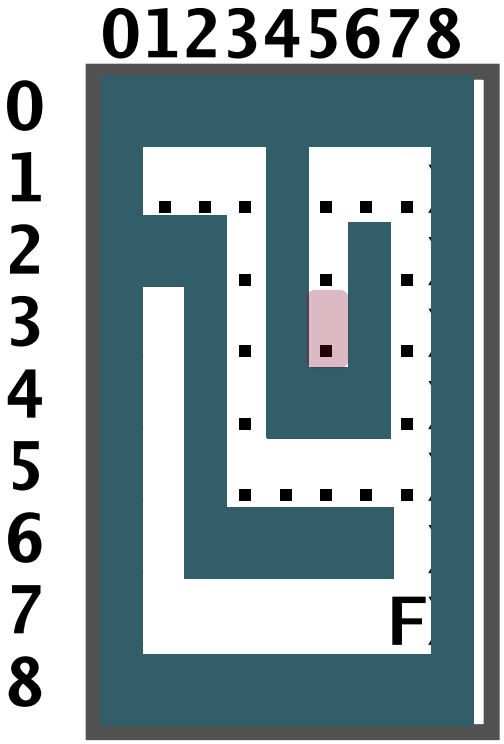
- Start: **row=1, col=1**. Marking with period (.)
- **… (continues). Now what happens? We check north first (bummer).**



Maze Solving

Now what?

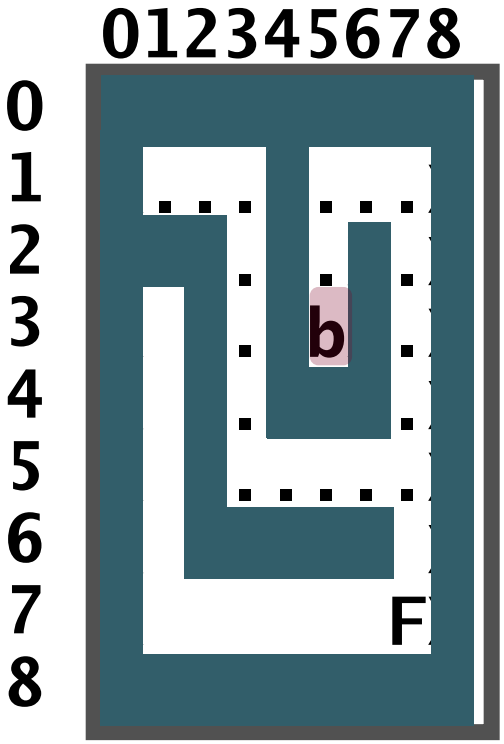
- Trying north, row=2,col=5. We were already there! Back to row=3, col=5.
- Trying east, row=3,col=6. Wall! Back to row=3, col=5.
- Trying south, row=4,col=5. Wall! Back to row=3, col=5.
- Trying west, row=3,col=4. Wall! Back to row=3, col=5.
- There are no solutions. Fail! Marking bad path with b. Back at row=2, col=5.



Maze Solving

Now what?

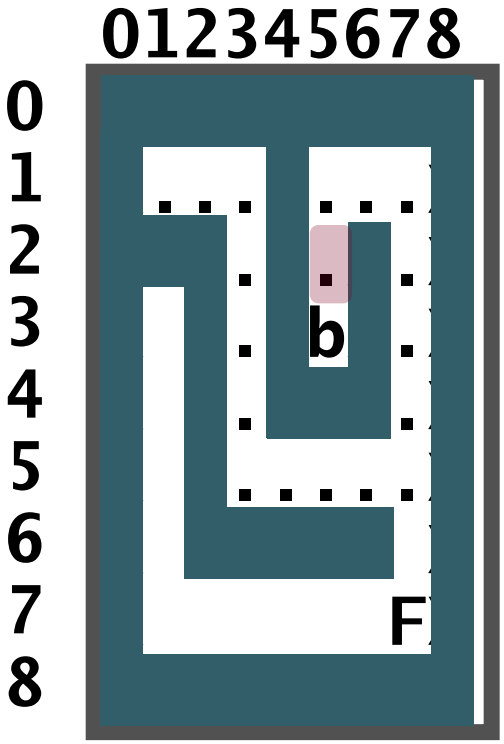
- Trying north, row=2,col=5. We were already there! Back to row=3, col=5.
- Trying east, row=3,col=6. Wall! Back to row=3, col=5.
- Trying south, row=4,col=5. Wall! Back to row=3, col=5.
- Trying west, row=3,col=4. Wall! Back to row=3, col=5.
- There are no solutions. Fail! Marking bad path with b. Back at row=2, col=5.



Maze Solving

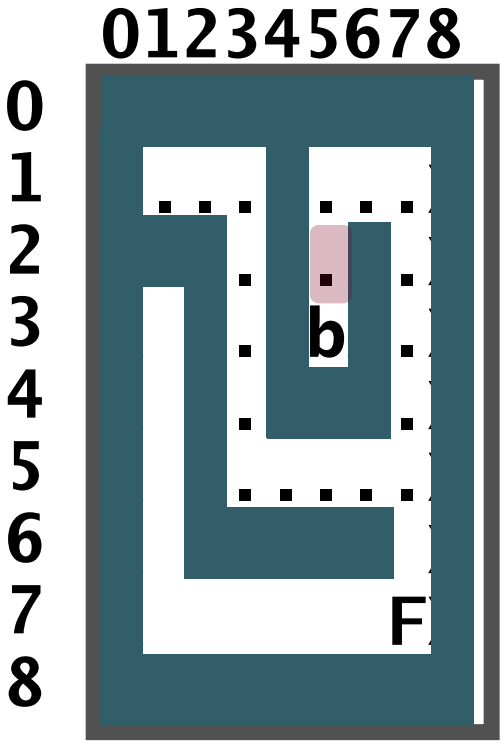
Now what?

- Trying north, row=2,col=5. We were already there! Back to row=3, col=5.
- Trying east, row=3,col=6. Wall! Back to row=3, col=5.
- Trying south, row=4,col=5. Wall! Back to row=3, col=5.
- Trying west, row=3,col=4. Wall! Back to row=3, col=5.
- There are no solutions. Fail! Marking bad path with b. Back at row=2, col=5.
- **Then what?** How did we get here? From the north! Meaning we checked <u>south</u> to get here. So now we check WEST.



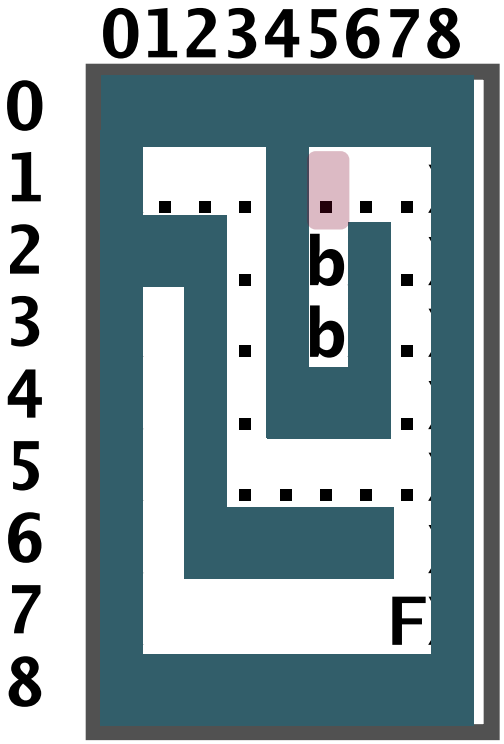
Maze Solving

- **Then what?** How did we get here? From the north! Meaning we checked south to get here. So now we check WEST.
- Trying west, row=2,col=4. Wall! Back at row=2, col=5. There are no solutions. Fail! Marking bad path with. B. Back at row=1, col=5.



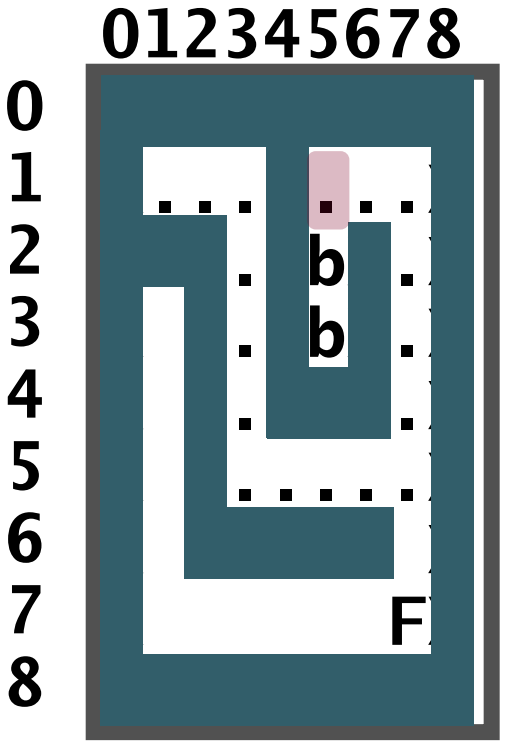
Maze Solving

- **Then what?** How did we get here? From the north! Meaning we checked south to get here. So now we check WEST.
- Trying west, row=2,col=4. Wall! Back at row=2, col=5. There are no solutions. Fail! Marking bad path with. B. Back at row=1, col=5.



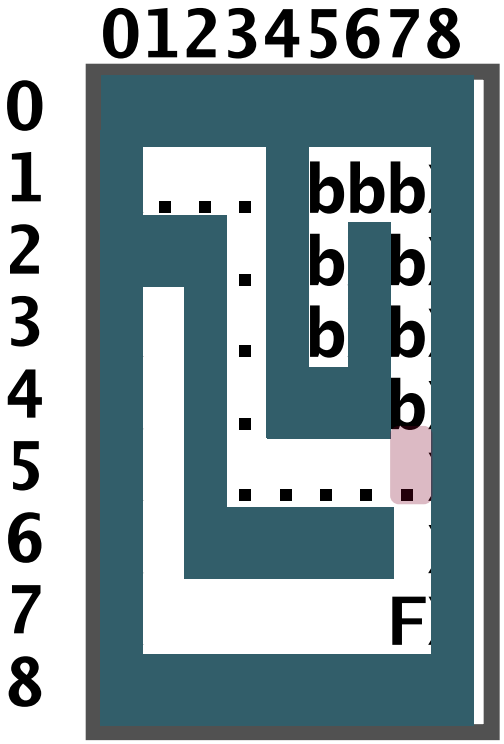
Maze Solving

- **Then what?** How did we get here? From the north! Meaning we checked south to get here. So now we check WEST.
- Trying west, row=2,col=4. Wall! Back at row=2, col=5. There are no solutions. Fail! Marking bad path with. B. Back at row=1, col=5.
- **The recursive calls "remember" where we have been!**



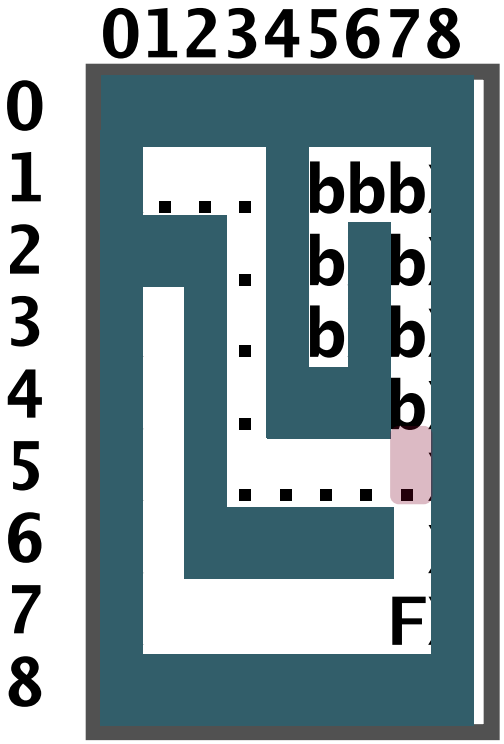
Maze Solving

- **Then what?** How did we get here? From the north! Meaning we checked south to get here. So now we check WEST.
- Trying west, row=2,col=4. Wall! Back at row=2, col=5. There are no solutions. Fail! Marking bad path with. B. Back at row=1, col=5.
- **The recursive calls "remember" where we have been!**
- **We will eventually arrive back at row=5, col=7.**



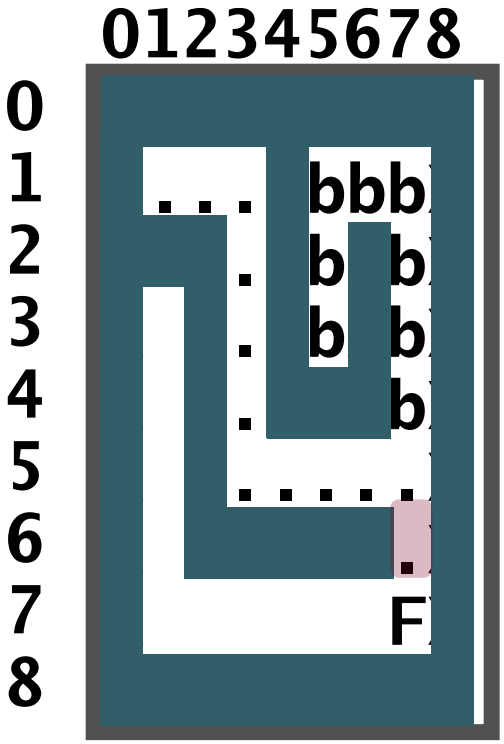
Maze Solving

- Trying east, row=5,col=8. Wall! Back at row=5, col=7.
- Trying south, row=6,col=7. Marking with period (.)



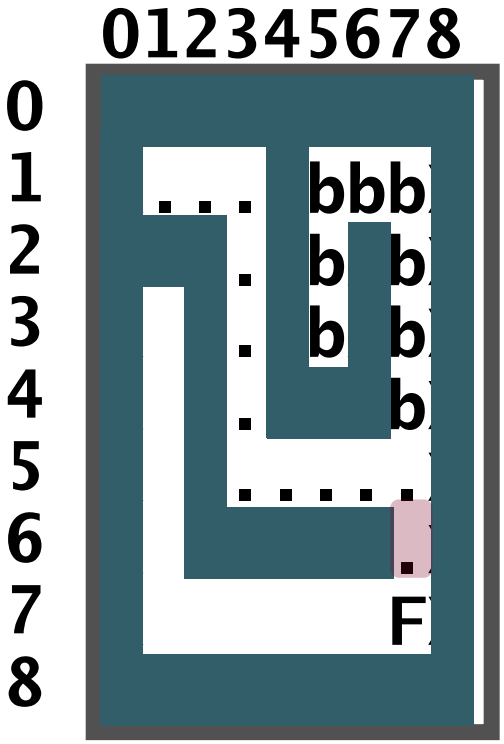
Maze Solving

- Trying east, row=5,col=8. Wall! Back at row=5, col=7.
- Trying south, row=6,col=7. Marking with period (.)



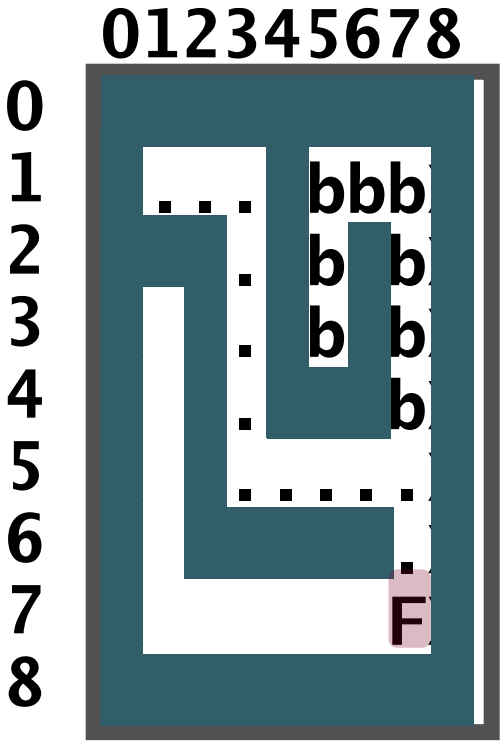
Maze Solving

- Trying east, row=5,col=8. Wall! Back at row=5, col=7.
- Trying south, row=6,col=7. Marking with period (.)
- Trying north, row=5,col=7. We were already there! Back to row=6,col=7.
- Trying east, row=6,col=8. Wall! Back to row=6, col=7.



Maze Solving

- Trying east, row=5,col=8. Wall! Back at row=5, col=7.
- Trying south, row=6,col=7. Marking with period (.)
- Trying north, row=5,col=7. We were already there! Back to row=6,col=7.
- Trying east, row=6,col=8. Wall! Back to row=6, col=7.
- Trying south, row=7, col=7. Found the finish!



Plan For Today

- **Recap:** what is recursive backtracking?
- **Recap:** dice rolls
- **Recap:** subsets
- Selection Problems: Subsets, Combinations and Permutations
- Announcements
- Practicing types of backtracking problems
  - *Find all solutions*
  - *Find any solution*
  - *How many solutions are there?*
  - *Does a solution exist?*
  - *Find the best solution*

Maze Counting

https://www.reddit.com/r/funny/comments/lejyk/billy_mays_maize_maze/

Count Solutions

Given a maze represented as a Grid<bool> (true if you can go somewhere, false if it's a wall), and a start and end location, count the number of unique paths from the start to the end. We assume the maze is bordered by walls. (This problem is useful for e.g. ensuring there is only 1 solution!)

```
int countMazeSolutionsHelper(Grid<bool>& maze,
                             int startRow,
                             int startCol, int endRow,
                             int endCol);
```

Count Solutions

```
int countMazeSolutionsHelper(Grid<bool>& maze, int startRow,
                             int startCol, int endRow, int endCol) {
    if (!maze[startRow][startCol]) {
        return 0;
    }
    if (startRow == endRow && startCol == endCol) {
        return 1; //reached our goal
    }

    maze[startRow][startCol] = false; // choose
    int numSolutions = countMazeSolutionsHelper(maze, startRow + 1,
        startCol, endRow, endCol);
    numSolutions += countMazeSolutionsHelper(maze, startRow - 1,
        startCol, endRow, endCol);
    numSolutions += countMazeSolutionsHelper(maze, startRow,
        startCol + 1, endRow, endCol);
    numSolutions += countMazeSolutionsHelper(maze, startRow,
        startCol - 1, endRow, endCol);
    maze[startRow][startCol] = true; // unchoose
    return numSolutions;
}
```

Count Solutions

- Base case: is it a valid solution? If so, return 1. Otherwise, return 0.
- Recursive step: return the sum of all the recursive calls.

This approach is useful because sometimes we want to make sure that there is *exactly* one solution. For instance, a maze!

Plan For Today

- **Recap:** what is recursive backtracking?
- **Recap:** dice rolls
- **Recap:** subsets
- Selection Problems: Subsets, Combinations and Permutations
- Practicing types of backtracking problems
  - *Find all solutions*
  - *Find any solution*
  - *How many solutions are there?*
  - *Does a solution exist?*
  - *Find the best solution*

A Startling Observation

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?

A Startling Observation

S	T	A	R	T	L	I	N	G

A Startling Observation

S	T	A	R	T	I	N	G

A Startling Observation

S	T	A	R	I	N	G

A Startling Observation

S	T	R	I	N	G

A Startling Observation

S	T	I	N	G

A Startling Observation

S	I	N	G

A Startling Observation

S	I	N

A Startling Observation

I	N

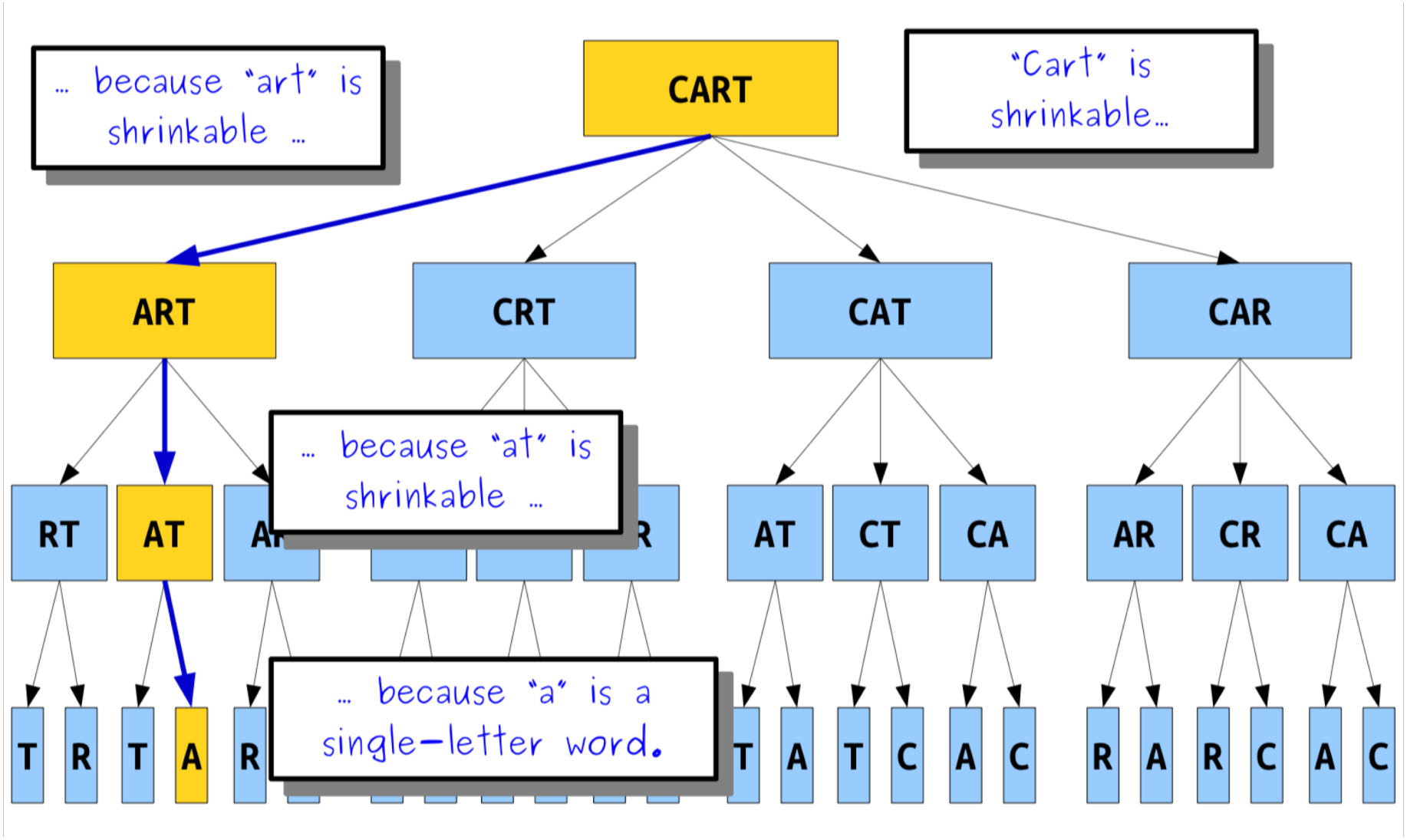
A Startling Observation

A Startling Observation

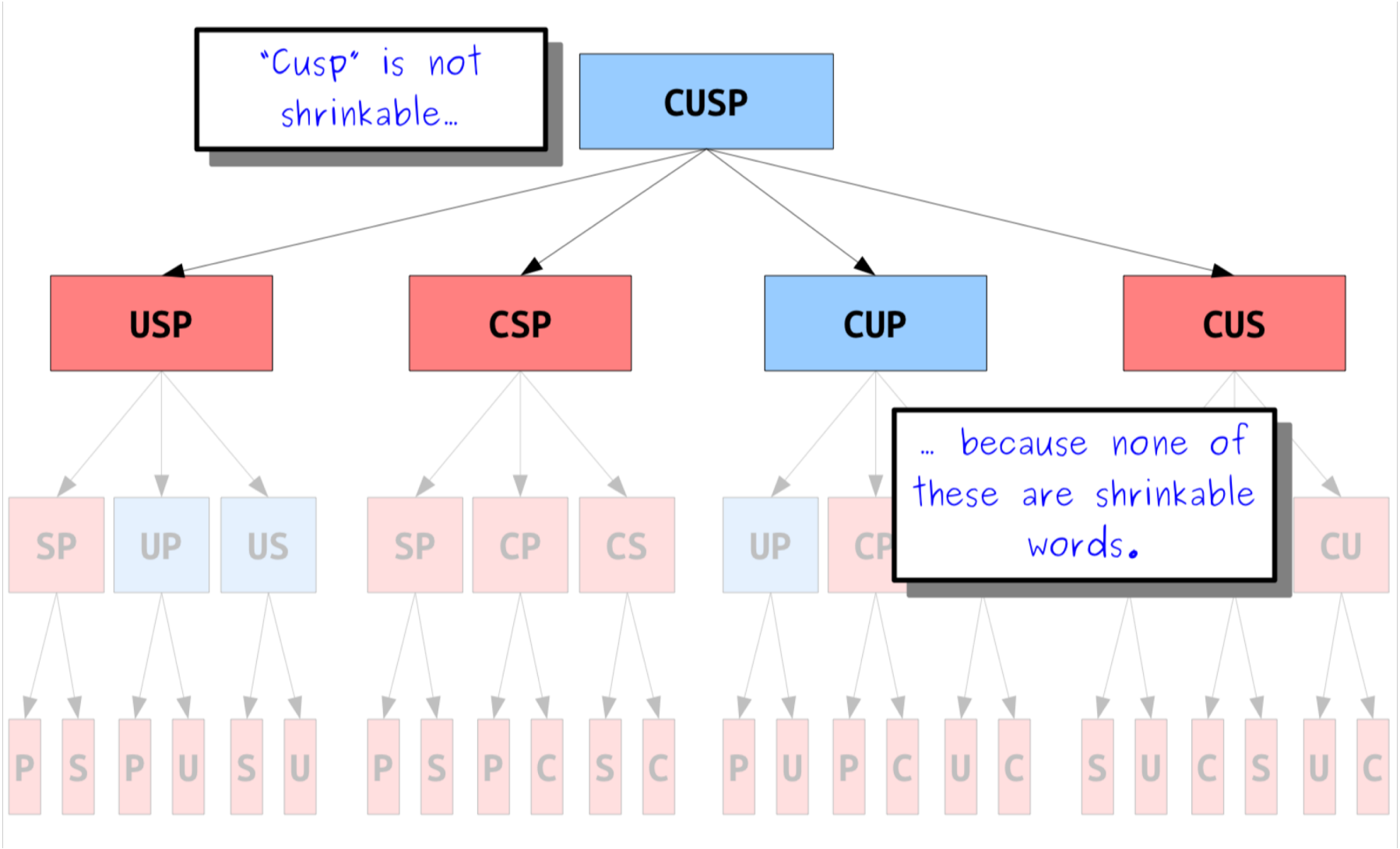
I

Are there other words with this property?

All Possible Paths



All Possible Paths



Shrinkable Words

- Let's define a **shrinkable word** as a word that can be reduced down to one letter by removing one character at a time, leaving a word at each step.
- Given an English dictionary, how can we determine whether a word is shrinkable?

The Backtracking Checklist

- ❑ **Find what choice(s) we have at each step.** What different options are there for the next step?

For each valid choice:

- ❑ **Make it and explore recursively.** Pass the information for a choice to the next recursive call(s).
- ❑ **Undo it after exploring.** Restore everything to the way it was before making this choice.

- ❑ **Find our base case(s).** What should we do when we are out of decisions?

isShrinkable

```
bool isShrinkable(const string& word, const Lexicon& english) {
    if (!english.contains(word)) return false;
    if (word.length() == 1) return true;

    for (int i = 0; i < word.length(); i++) {
        string shrunken = word.substr(0, i) + word.substr(i+1);
        if (isShrinkable(shrunken, english)) {
            return true;
        }
    }

    return false;
}
```

Next Time

- To be continued…