Recursive Backtracking

Why I have no friends, reason #1729: Unimpressive mindblowing facts

Did you know that the word “recursion” contains the word “recursion” in itself?

Whoa, that’s amaz... you’re an asshole.
Outline

1. Solving Mazes
2. Words & Permutations
Recursion Reminder

Solving Recursion Problems

- Figure out what the pieces of the problem are.
- What is the base case? (the smallest possible piece of the problem)
- Solve one piece of the problem and recurse on the rest.

paintbucket Review

- A piece of the problem is **one surrounding set of squares**
- The base case is **we hit a non-white cell**
- To solve one piece of the problem, we **color the cell** and **go left, right, up, and down**
Solving a maze is a lot like paintbucket. What is the difference?

**Instead of filling everything in, we want to stop at dead ends!**

If you were in a maze, how would you solve it?

- Try a direction.
- Every time you go in a direction, draw an X on the ground.
- If you hit a dead end, go back until you can go in another direction.

**This is recursive backtracking!**

```java
public boolean canSolveMaze(int x, int y) {
    if (isGoal(x, y)) {
        return true;
    } else if (inBounds(x, y) && isPassage(x, y)) {
        return solveMaze(x + 1, y) ||
                solveMaze(x - 1, y) ||
                solveMaze(x, y + 1) ||
                solveMaze(x, y - 1);
    }
}
```
public static boolean solveMaze(Point p) {
    // We found a path to the goal!
    if (p.isGoal()) {
        p.makeVisited(panel);
        return true;
    }

    // If the point is a valid part of a path to the solution...
    if (!p.isOOB() && p.isPassage(panel)) {
        p.makeVisited(panel); // Choose this point
        panel.sleep(120);
        if (solveMaze(p.getLeft()) || // Try each direction
            solveMaze(p.getRight()) || // until we get a
            solveMaze(p.getAbove()) || // solution.
            solveMaze(p.getBelow()))) {
            return true;
        }
    }
    panel.sleep(200);
    p.makeDeadEnd(panel); // Undo the choice
    return false;
}
Definition (Recursive Backtracking)

Recursive Backtracking is an attempt to find solution(s) by building up partial solutions and abandoning them if they don’t work.

Recursive Backtracking Strategy

- If we found a solution, stop looking (e.g. return)
- Otherwise for each possible choice $c$...
  - Make the choice $c$
  - Recursively continue to make choices
  - Un-make the choice $c$ (if we got back here, it means we need to continue looking)
All Words

Find all length \( n \) strings made up of \( a \)'s, \( b \)'s, \( c \)'s, and \( d \)'s.

To do this, we build up partial solutions as follows:

(Assume there is a variable \( \text{part} \) that is initialized to "").

- The only length 0 string is ""; so, \( \text{part} \) is a solution.
- Otherwise, the four choices are \( a \), \( b \), \( c \), and \( d \):
  - To make the choice \( \text{letter} \), we set \( \text{part} += \text{letter} \).
  - Then, we need to find all solutions with one fewer letter recursively.
  - Now, we unmake the choice (to continue looking) by removing \( \text{letter} \) from \( \text{part} \).
```java
String part = "";
private static void words(int length) {
    String[] choices = {"a", "b", "c", "d"};
    // The empty string is the only word of length 0
    if (length == 0) {
        System.out.println(part);
    } else {
        // Try appending each possible choice to our partial word.
        for (String choice : choices) {
            part += choice; // Add the choice
            words(length - 1); // Recurse on the rest
            int size = part.length(); // Undo the choice
            part = part.substring(0, size - 1);
        }
    }
}
```

**Permutations**

How do we change `words` to only print out words that have each character exactly once?
Idea: When a solution becomes "bad" (it has multiple of the same letter), stop trying that branch.

```java
String part = "";
private static void permutations(int length) {
    String[] choices = {"a", "b", "c", "d");
    // If we have a repeat letter, the solution is invalid.
    if (hasRepeats(part)) {
        return;
    }
    else if (length == 0) {
        System.out.println(part);
    }
    else {
        for (String choice : choices) {
            part += choice;
            permutations(length - 1);
            int size = part.length();
            part = part.substring(0, size - 1);
        }
    }
}
```
Recursive Backtracking Tips!

- The most important part is figuring out what the choices are.

- It can help to draw out a tree of choices.

- Make sure to undo your choices after the recursive call.

- You will still always have a base case.