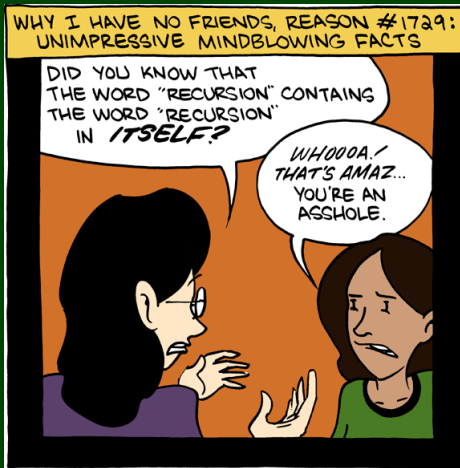


CSE 143

Computer Programming II

Recursive Backtracking



Outline

1 Solving Mazes

2 Words & Permutations

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!

```
1 public boolean canSolveMaze(int x, int y) {
2     if (isGoal(x, y)) {
3         return true;
4     }
5     else if (inBounds(x, y) && isPassage(x, y)) {
6         return solveMaze(x + 1, y) ||
7             solveMaze(x - 1, y) ||
8             solveMaze(x, y + 1) ||
9             solveMaze(x, y - 1);
10    }
11 }
```

```
1 public static boolean solveMaze(Point p) {
2     // We found a path to the goal!
3     if (p.isGoal()) {
4         p.makeVisited(panel);
5         return true;
6     }
7
8     // If the point is a valid part of a path to the solution...
9     if (!p.isOB() && p.isPassage(panel)) {
10        p.makeVisited(panel);           // Choose this point
11        panel.sleep(120);
12        if (solveMaze(p.getLeft()) || // Try each direction
13            solveMaze(p.getRight()) || // until we get a
14            solveMaze(p.getAbove()) || // solution.
15            solveMaze(p.getBelow())) {
16            return true;
17        }
18        panel.sleep(200);
19        p.makeDeadEnd(panel);           // Undo the choice
20    }
21    return false;
22 }
```

Definition (Recursive Backtracking)

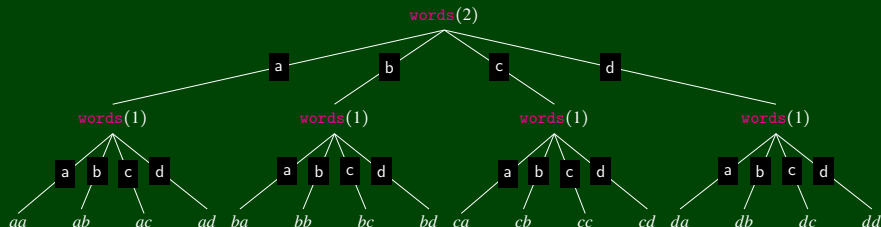
Recursive Backtracking is an attempt to find solution(s) by building up partial solutions and abandoning them if they don't work.

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 `part` that is initialized to `""`.)

- The only length 0 string is `""`; so, `part` is a solution.
- Otherwise, the four choices are a , b , c , and d :
 - To make the choice `letter`, we set `part += letter`.
 - Then, we need to find all solutions with one fewer letter recursively.
 - Now, we unmake the choice (to continue looking) by removing `letter` from `part`.


```
1 String part = "";
2 private static void words(int length) {
3     String[] choices = {"a", "b", "c", "d"};
4     // The empty string is the only word of length 0
5     if (length == 0) {
6         System.out.println(part);
7     }
8     else {
9         // Try appending each possible choice to our partial word.
10        for (String choice : choices) {
11            part += choice;           // Add the choice
12            words(length - 1);       // Recurse on the rest
13            int size = part.length()
14            part = part.substring(0, size - 1); // Undo the choice
15        }
16    }
17 }
```

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.

```
1 String part = "";
2 private static void permutations(int length) {
3     String[] choices = {"a", "b", "c", "d"};
4     // If we have a repeat letter, the solution is invalid.
5     if (hasRepeats(part)) {
6         return;
7     }
8     else if (length == 0) {
9         System.out.println(part);
10    }
11    else {
12        for (String choice : choices) {
13            part += choice;
14            permutations(length - 1);
15            int size = part.length()
16            part = part.substring(0, size - 1);
17        }
18    }
19 }
```

- 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.