Slide 1

Adam Blank   **Lecture 18**   **Spring 2015**

CSE 143

Computer Programming II

Slide 2

CSE 143: Computer Programming II

Recursive Backtracking

Slide 3 — Outline

Outline

1. Words & Permutations

2. Solving Mazes

Slide 4 — Recursive Backtracking

Recursive Backtracking   1

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.

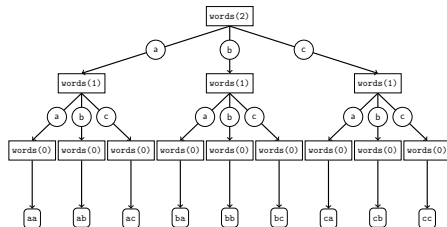**Recursive Backtracking Strategy**

- 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)

Slide 5 — Words & Permutations

Words & Permutations   2

All Words

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

```
                          words(2)
              a               b               c
          words(1)        words(1)        words(1)
        a    b    c      a    b    c      a    b    c
    words(0) words(0) words(0) words(0) words(0) words(0) words(0) words(0) words(0)
        aa   ab   ac     ba   bb   bc     ca   cb   cc
```

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

- The only length 0 string is ""; so, we're done.
- Otherwise, the three choices are a, b, and c:
  - Make the choice `letter`
  - Find all solutions with one fewer letter recursively.
  - Unmake the choice (to continue looking).

Slide 6 — All Words Solution

All Words Solution   3

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

```
1  private static void words(String acc, int length) {
2      String[] choices = {"a", "b", "c", "d"};
3      // The empty string is the only word of length 0
4      if (length == 0) {
5          print();
6      }
7      else {
8          for (String choice : choices) {
9              acc += choice;
10             words(acc, length - 1);
11             acc = acc.substring(0, acc.length() - 1);
12         }
13     }
14 }
```

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.isOOB() && 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 }
```

- 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.