Recursive Backtracking

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
  - Make the choice
  - Recursively continue to make choices
  - Un-make the choice (if we got back here, it means we need to continue looking)

Words & Permutations

All Words

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

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).

All Words Solution

```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      } else {
7          // Try appending each possible choice to our partial word.
8          for (String choice : choices) {
9              choose(choice);  // Add the choice
10             words(length - 1);  // Recurse on the rest
11             unchoose();  // undo the choice
12          }
13      }
14  }
15 }```

private static void words(String acc, int length) {
    String[] choices = {"a", "b", "c", "d");
    // The empty string is the only word of length 0
    if (length == 0) {
        print();
    } else {
        for (String choice : choices) {
            acc += choice;
            words(acc, length - 1);
            acc = acc.substring(0, acc.length() - 1);
        }
    }
}

public boolean canSolveMaze(int x, int y) {
    // We found a path to the goal!
    if (p.isGoal()) {
        p.makeVisited(panel);
        return true;
    }
    else if (!p.isOOB() && p.isPassage(panel)) {
        p.makeVisited(panel);
        panel.sleep(120);
        if (solveMaze(p.getLeft()) ||
            solveMaze(p.getRight()) ||
            solveMaze(p.getAbove()) ||
            solveMaze(p.getBelow())) {
            return true;
        }
        panel.sleep(200);
        p.makeDeadEnd(panel); // Undo the choice
    }
    return false;
}

public static boolean solveMaze(Point p) {
    // We found a path to the goal!
    if (p.isGoal()) {
        p.makeVisited(panel);
        return true;
    }
    else if (inBounds(x, y) && isPassage(x, y)) {
        solveMaze(x + 1, y) ||
        solveMaze(x - 1, y) ||
        solveMaze(x + 1, y) ||
        solveMaze(x, y - 1);
    } else {
        panel.sleep(120);
        if (solveMaze(p.getRight()) ||
            solveMaze(p.getBelow())) {
            return true;
        }
    }
    return false;
}

public static boolean solveMaze(Point p) {
    // We found a path to the goal!
    if (p.isGoal()) {
        p.makeVisited(panel);
        return true;
    }
    else if (!p.isOOB() && p.isPassage(panel)) {
        p.makeVisited(panel);
        panel.sleep(120);
        if (solveMaze(p.getLeft()) ||
            solveMaze(p.getBelow())) {
            return true;
        }
        panel.sleep(200);
        p.makeDeadEnd(panel); // Undo the choice
    }
    return false;
}

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.