Building Java Programs

read: 12.5
Recursive backtracking
Exhaustive Search

- Iterate through all elements of a search space

- Useful to solve problems that require making decisions
  - Each decision leads to new choices
  - Insufficient information to make a thoughtful choice

- Depth first search: we go deep down one path rather than broad

- Natural to implement recursively: call stack keeps track of decision points in right order (opposite from visited)
Exercise: Permutations

- Write a method `permute` that accepts a string as a parameter and outputs all possible rearrangements of the letters in that string. The arrangements may be output in any order.

  Example:
  ```java
  permute("TEAM")
  ```
  outputs the following sequence of lines:

```
TEAM
TEMA
TAME
TMEA
TMAE
ETAM
ETMA
EMTA
EMAT
EMTA
EAMT
EMAT
ATEM
ATEM
TAEM
AETM
AEMT
AMTE
AMET
MTEA
MTAE
META
MEAT
MATE
MAET
```
Examining the problem

- We want to generate all possible sequences of letters.
  for (each possible first letter):
    for (each possible second letter):
      for (each possible third letter):
        ...
        print!

- Each permutation is a set of choices or **decisions**:
  - Which character do I want to place first?
  - Which character do I want to place second?
  - ...
  - **solution space**: set of all possible sets of decisions to explore
Decision tree

<table>
<thead>
<tr>
<th>chosen</th>
<th>available</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEAM</td>
<td></td>
</tr>
</tbody>
</table>

TEAM

TEAM

TEAM

TEAM

TEAM

TEAM

TEAM

TEAM

TEAM

TEAM

TEAM
// Outputs all permutations of the given string.
public static void permute(String s) {
    permute(s, "");
}

private static void permute(String s, String chosen) {
    if (s.length() == 0) {
        // base case: no choices left to be made
        System.out.println(chosen);
    } else {
        // recursive case: choose each possible next letter
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i); // choose
            s = s.substring(0, i) + s.substring(i + 1); // explore
            chosen += c;
            permute(s, chosen); // explore
            chosen = chosen.substring(0, chosen.length() - 1); // un-choose
            s = s.substring(0, i) + c + s.substring(i); // un-choose
        }
    }
}
Exercise solution 2

// Outputs all permutations of the given string.
public static void permute(String s) {
    permute(s, "");
}

private static void permute(String s, String chosen) {
    if (s.length() == 0) {
        // base case: no choices left to be made
        System.out.println(chosen);
    } else {
        // recursive case: choose each possible next letter
        for (int i = 0; i < s.length(); i++) {
            String ch = s.substring(i, i + 1); // choose
            String rest = s.substring(0, i) + 
                          s.substring(i + 1); // remove
            permute(rest, chosen + ch); // explore
        }
        // (don't need to "un-choose" because
        // we used temp variables)
Backtracking

- Useful to solve problems that require making decisions
  - Each decision leads to new choices
  - Some (but not all!) sequence(s) of choices will be a solution
  - Insufficient information to make a thoughtful choice

- Systematically prune out infeasible solutions
Backtracking strategies

- When solving a backtracking problem, ask these questions:
  - What are the "choices" in this problem?
    - What is the "base case"? (How do I know when I'm out of choices?)
  - How do I "make" a choice?
    - Do I need to create additional variables to remember my choices?
    - Do I need to modify the values of existing variables?
  - How do I explore the rest of the choices?
    - Do I need to remove the made choice from the list of choices?
  - Once I'm done exploring, what should I do?
  - How do I "un-make" a choice?
Maze class

- Suppose we have a `Maze` class with these methods:

<table>
<thead>
<tr>
<th>Method/Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public <code>Maze(String text)</code></td>
<td>construct a given maze</td>
</tr>
<tr>
<td>public int <code>getHeight()</code>, <code>getWidth()</code></td>
<td>get maze dimensions</td>
</tr>
<tr>
<td>public boolean <code>isExplored(int r, int c)</code></td>
<td>get/set whether you have visited a location</td>
</tr>
<tr>
<td>public void <code>setExplored(int r, int c)</code></td>
<td>get/set whether you have visited a location</td>
</tr>
<tr>
<td>public void <code>isWall(int r, int c)</code></td>
<td>whether given location is blocked by a wall</td>
</tr>
<tr>
<td>public void <code>mark(int r, int c)</code></td>
<td>whether given location is marked in a path</td>
</tr>
<tr>
<td>public void <code>isMarked(int r, int c)</code></td>
<td>whether given location is marked in a path</td>
</tr>
<tr>
<td>public String <code>toString()</code></td>
<td>text display of maze</td>
</tr>
</tbody>
</table>
Exercise: solve maze

- Write a method `solveMaze` that accepts a `Maze` and a starting row/column as parameters and tries to find a path out of the maze starting from that position.

  - If you find a solution:
    - Your code should **stop** exploring.
    - You should **mark** the path out of the maze on your way back out of the recursion, using backtracking.

  - (As you explore the maze, squares you set as 'explored' will be printed with a dot, and squares you 'mark' will display an X.)
Decision tree

position (row 1, col 7)

choices (these never change)

- (1, 6)
  - (1, 5) wall
  - (0, 6) wall
  - (2, 6) visited
- (0, 7)
- (2, 7) wall
- (1, 8) wall
Recall: Backtracking

A general pseudo-code algorithm for backtracking problems:

Explore(\textit{choices}):

- if there are no more \textit{choices} to make: stop.
- else, for each available choice \textbf{C}:
  - Choose \textbf{C}.
  - Explore the remaining \textit{choices}.
  - Un-choose \textbf{C}, if necessary. (backtrack!)

What are the choices in this problem?