Building Java Programs

Appendix R

Recursive backtracking

Backtracking

- Useful to solve problems that require making decisions
  - Insufficient information to make a thoughtful choice
  - Each decision leads to new choices
  - Some sequence of choices will be a solution

- Backtracking involves trying out sequences of decisions until one that works is found

- 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)

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?

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

```
TEAM  |  ATEM
TEMA  |  ATME
TAEM  |  AETM
TAME  |  AEMT
TMEA  |  AMTE
TMAE  |  AMET
ETAM  |  MTEA
ETMA  |  MTAE
EATM  |  META
EAMT  |  MEAT
EMTA  |  MATE
EMAT  |  MAET
```

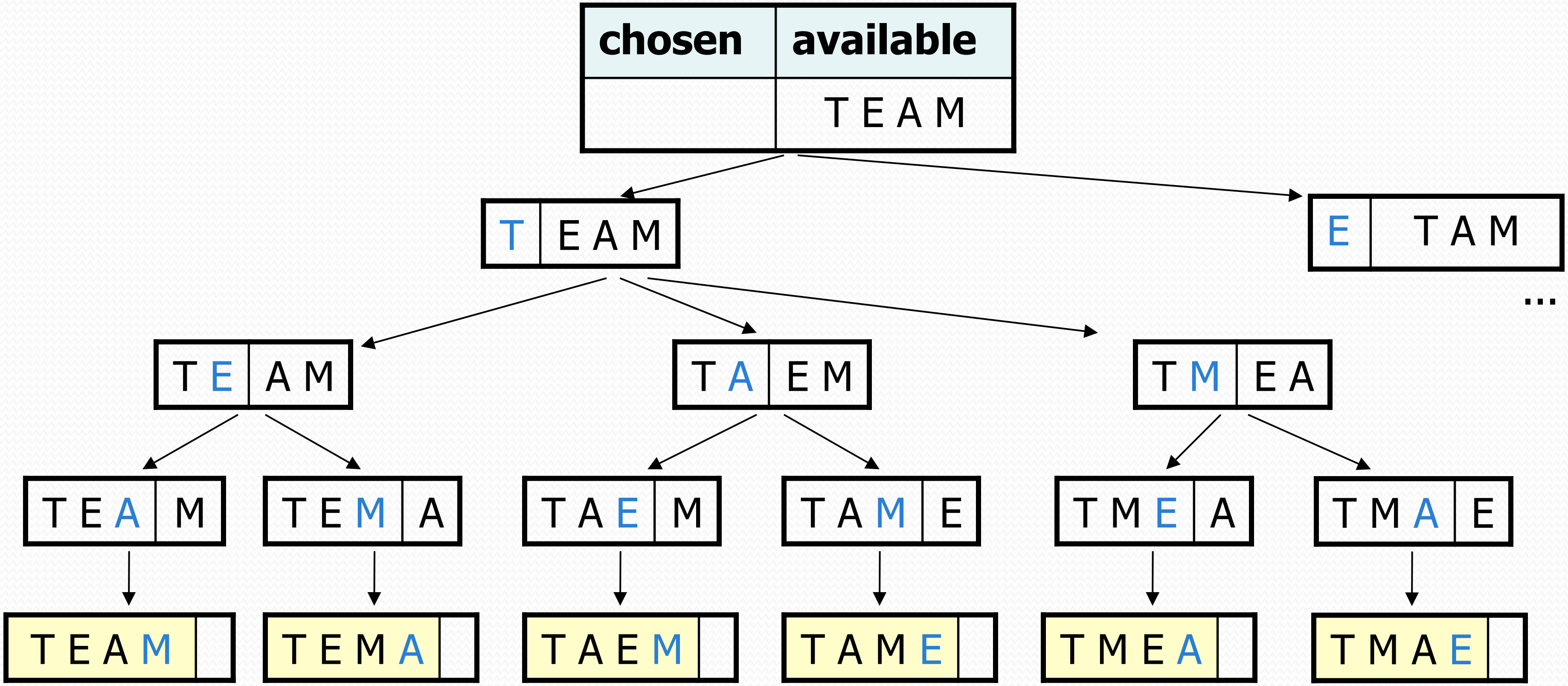
Examining the problem

- 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



chosen available

TEAM

T EAM

E TAM

...

TE AM

TA EM

TM EA

TEA M

TEM A

TAE M

TAM E

TME A

TMA E

TEAM

TEMA

TAEM

TAME

TMEA

TMAE

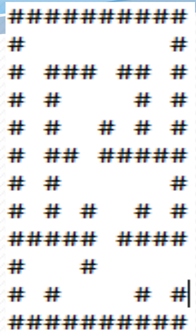
Exercise solution

```
// 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);
            chosen += c;

            permute(s, chosen);                 // explore

            s = s.substring(0, i) + c + s.substring(i);
            chosen = chosen.substring(0, chosen.length() - 1);
        }                                       // 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) +   // remove
                          s.substring(i + 1);
            permute(rest, chosen + ch);         // explore
        }
    }       // (don't need to "un-choose" because
}           //  we used temp variables)
```

Maze class



```
##########
#        #
# ### ## #
# #    # #
# #  # # #
# ## #####
# #      #
# # #  # #
##### ####
#   #
# #    # #
##########
```

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

Method/Constructor	Description
`public` **`Maze`**`(String text)`	construct a given maze
`public int` **`getHeight`**`(),` **`getWidth`**`()`	get maze dimensions

`public boolean` **`isExplored`**`(int r, int c)`
`public void` **`setExplored`**`(int r, int c)`
	get/set whether you have visited a location

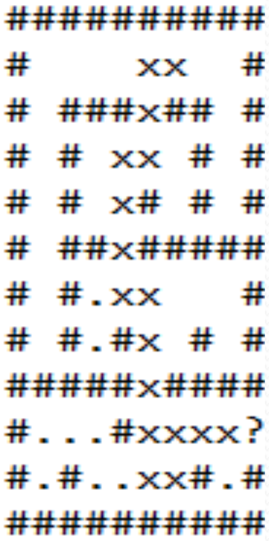
`public void` **`isWall`**`(int r, int c)`	whether given location is blocked by a wall

`public void` **`mark`**`(int r, int c)`
`public void` **`isMarked`**`(int r, int c)`
	whether given location is marked in a path

`public String` **`toString`**`()`	text display of maze

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.)



```
##########
#    xx  #
# ###x## #
# # xx # #
# # x# # #
# ##x#####
# #.xx   #
# #.#x # #
#####x####
#...#xxxx?
#.#..xx#.#
##########
```

Recall: Backtracking

A general pseudo-code algorithm for backtracking problems:

Explore(**choices**):

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

What are the choices in this problem?