CSE 143

Recursive backtracking
Exercise: Dice rolls

- Write a method `diceRoll` that accepts an integer parameter representing a number of 6-sided dice to roll, and output all possible arrangements of values that could appear on the dice.

```java
diceRoll(2);
```

```plaintext
[1, 1]  [3, 1]  [5, 1] 
[1, 2]  [3, 2]  [5, 2] 
[1, 3]  [3, 3]  [5, 3] 
[1, 4]  [3, 4]  [5, 4] 
[1, 5]  [3, 5]  [5, 5] 
[1, 6]  [3, 6]  [5, 6] 
[2, 1]  [4, 1]  [6, 1] 
[2, 2]  [4, 2]  [6, 2] 
[2, 3]  [4, 3]  [6, 3] 
[2, 4]  [4, 4]  [6, 4] 
[2, 5]  [4, 5]  [6, 5] 
[2, 6]  [4, 6]  [6, 6] 
```

```java
diceRoll(3);
```

```plaintext
[1, 1, 1]  [1, 1, 2]  [1, 1, 3]  [1, 1, 4]  [1, 1, 5]  [1, 1, 6]  [1, 2, 1]  [1, 2, 2]  ...  [6, 6, 4]  [6, 6, 5]  [6, 6, 6]
```
Examining the problem

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

- This is called a **depth-first search**

- How can we completely explore such a large search space?
A decision tree

<table>
<thead>
<tr>
<th>chosen</th>
<th>available</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>4 dice</td>
</tr>
</tbody>
</table>

1 3 dice

1, 1 2 dice
1, 2 2 dice
1, 3 2 dice
1, 4 2 dice

1, 1, 1 1 die
1, 1, 2 1 die
1, 1, 3 1 die
1, 4, 1 1 die

1, 1, 1, 1
1, 1, 1, 2
... 1, 1, 3, 1
... 1, 1, 3, 2
Exercise: Dice roll sum

- Write a method `diceSum` similar to `diceRoll`, but it also accepts a desired sum and prints only arrangements that add up to exactly that sum.

```java
diceSum(2, 7);
diceSum(3, 7);
```

- Sample arrangements:

  ```
  [1, 1, 5]
  [1, 2, 4]
  [1, 3, 3]
  [1, 4, 2]
  [1, 5, 1]
  [2, 1, 4]
  [2, 2, 3]
  [2, 3, 2]
  [2, 4, 1]
  [2, 5, 1]
  [3, 1, 3]
  [3, 2, 2]
  [3, 3, 1]
  [3, 4, 1]
  [3, 5, 1]
  [4, 1, 2]
  [4, 2, 1]
  [4, 3, 1]
  [5, 1, 1]
  ```
Consider all paths?

<table>
<thead>
<tr>
<th>chosen</th>
<th>available</th>
<th>desired sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>3 dice</td>
<td>5</td>
</tr>
</tbody>
</table>

```
1, 1  1 die
1, 2  1 die
1, 3  1 die
1, 4  1 die
1, 5  1 die
1, 6  1 die
1, 1, 1
1, 1, 2
1, 1, 3
1, 1, 4
1, 1, 5
1, 1, 6
1, 6, 1
1, 6, 2
...```
New decision tree

<table>
<thead>
<tr>
<th>chosen</th>
<th>available</th>
<th>desired sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>3 dice</td>
<td>5</td>
</tr>
</tbody>
</table>

...
Backtracking

- **backtracking**: Finding solution(s) by trying partial solutions and then abandoning them if they are not suitable.
  - a "brute force" algorithmic technique (tries all paths)
  - often implemented recursively

Applications:
- producing all permutations of a set of values
- parsing languages
- games: anagrams, crosswords, word jumbles, 8 queens
- combinatorics and logic programming
Backtracking algorithms

A general pseudo-code algorithm for backtracking problems:

Explore(\texttt{choices}):  
\begin{itemize} 
    \item if there are no more \texttt{choices} to make: stop. 
    \item else: 
        \begin{itemize} 
            \item Make a single choice \texttt{C}. 
            \item Explore the remaining \texttt{choices}. 
            \item Un-make choice \texttt{C}, if necessary. (backtrack!) 
        \end{itemize} 
\end{itemize}
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: Combinations

• Write a method `combinations` that accepts a string `s` and an integer `k` as parameters and outputs all possible `k`-letter words that can be formed from unique letters in that string. The arrangements may be output in any order.

• Example:
  `combinations("GOOGLE", 3)` outputs the sequence of lines at right.

• To simplify the problem, you may assume that the string `s` contains at least `k` unique characters.
Initial attempt

public static void combinations(String s, int length) {
    combinations(s, "", length);
}

private static void combinations(String s, String chosen, int length) {
    if (length == 0) {
        System.out.println(chosen); // base case: no choices left
    } else {
        for (int i = 0; i < s.length(); i++) {
            String ch = s.substring(i, i + 1);
            if (!chosen.contains(ch)) {
                String rest = s.substring(0, i) + s.substring(i + 1);
                combinations(rest, chosen + ch, length - 1);
            }
        }
    }
}

• Problem: Prints same string multiple times.
Exercise solution

```java
public static void combinations(String s, int length) {
    Set<String> all = new TreeSet<String>();
    combinations(s, "", all, length);
    for (String comb : all) {
        System.out.println(comb);
    }
}

private static void combinations(String s, String chosen,
                                   Set<String> all, int length) {
    if (length == 0) {
        all.add(chosen);  // base case: no choices left
    } else {
        for (int i = 0; i < s.length(); i++) {
            String ch = s.substring(i, i + 1);
            if (!chosen.contains(ch)) {
                String rest = s.substring(0, i) + s.substring(i + 1);
                combinations(rest, chosen + ch, all, length - 1);
            }
        }
    }
}
```