CSE 143
Lecture 17
Recursive Backtracking

slides created by Marty Stepp
http://www.cs.washington.edu/143/
ideas and examples taken from Stanford University CS slides/lectures
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("MARTY")
  ```
  outputs the following sequence of lines:
Examining the problem

• Think of each permutation as 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

• We want to generate all possible sequences of decisions.
  for (each possible first letter):
    for (each possible second letter):
      for (each possible third letter):
        ...
        print!
  – This is called a **depth-first search**
Decision trees

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

MARTY

MARTY

MARTY

MARTY

MARTY

MARYT

MARYT

MARYT

MARYT

MARYT
• **backtracking**: A general algorithm for finding solution(s) to a computational problem by trying partial solutions and then abandoning them ("backtracking") if they are not suitable.
  
  – a "brute force" algorithmic technique (tries all paths; not clever)
  – often (but not always) implemented recursively

Applications:
  
  – producing all permutations of a set of values
  – parsing languages
  – games: anagrams, crosswords, word jumbles, 8 queens
  – combinatorics and logic programming
A general pseudo-code algorithm for backtracking problems:

explore(choices):

– if there are no more choices to make: stop.

– else:

  • Make a single choice C from the set of choices.
    – Remove C from the set of choices.

  • explore the remaining choices.

  • Un-make choice C.
    – Backtrack!
• 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 the rest, what should I do?
  – How do I "un-make" a choice?
Permutations revisited

• 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("MARTY")
    ```
    outputs the following sequence of lines:
// 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
// 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
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.
```java
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

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);
            }
        }
    }
}