CSE 143
Lecture 12

Recursive Backtracking

reading: "Appendix R" on course web site

slides adapted from Marty Stepp and Hélène Martin
http://www.cs.washington.edu/143/
ideas and examples taken from Stanford University CS slides/lectures
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 combinations of values that could appear on the dice.

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

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

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

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

```
diceRoll(3);
```

```
[1, 1, 1]  [1, 1, 2]  [1, 1, 3]  [1, 1, 4]  [1, 1, 5]  [1, 1, 6]  
[1, 2, 1]  [1, 2, 2]  [1, 2, 3]  [1, 2, 4]  [1, 2, 5]  [1, 2, 6]  
[1, 3, 1]  [1, 3, 2]  [1, 3, 3]  [1, 3, 4]  [1, 3, 5]  [1, 3, 6]  
[1, 4, 1]  [1, 4, 2]  [1, 4, 3]  [1, 4, 4]  [1, 4, 5]  [1, 4, 6]  
[1, 5, 1]  [1, 5, 2]  [1, 5, 3]  [1, 5, 4]  [1, 5, 5]  [1, 5, 6]  
[1, 6, 1]  [1, 6, 2]  [1, 6, 3]  [1, 6, 4]  [1, 6, 5]  [1, 6, 6]  
[2, 1, 1]  [2, 1, 2]  [2, 1, 3]  [2, 1, 4]  [2, 1, 5]  [2, 1, 6]  
[2, 2, 1]  [2, 2, 2]  [2, 2, 3]  [2, 2, 4]  [2, 2, 5]  [2, 2, 6]  
[2, 3, 1]  [2, 3, 2]  [2, 3, 3]  [2, 3, 4]  [2, 3, 5]  [2, 3, 6]  
[2, 4, 1]  [2, 4, 2]  [2, 4, 3]  [2, 4, 4]  [2, 4, 5]  [2, 4, 6]  
[2, 5, 1]  [2, 5, 2]  [2, 5, 3]  [2, 5, 4]  [2, 5, 5]  [2, 5, 6]  
[2, 6, 1]  [2, 6, 2]  [2, 6, 3]  [2, 6, 4]  [2, 6, 5]  [2, 6, 6]  
[3, 1, 1]  [3, 1, 2]  [3, 1, 3]  [3, 1, 4]  [3, 1, 5]  [3, 1, 6]  
[3, 2, 1]  [3, 2, 2]  [3, 2, 3]  [3, 2, 4]  [3, 2, 5]  [3, 2, 6]  
[3, 3, 1]  [3, 3, 2]  [3, 3, 3]  [3, 3, 4]  [3, 3, 5]  [3, 3, 6]  
[3, 4, 1]  [3, 4, 2]  [3, 4, 3]  [3, 4, 4]  [3, 4, 5]  [3, 4, 6]  
[3, 5, 1]  [3, 5, 2]  [3, 5, 3]  [3, 5, 4]  [3, 5, 5]  [3, 5, 6]  
[3, 6, 1]  [3, 6, 2]  [3, 6, 3]  [3, 6, 4]  [3, 6, 5]  [3, 6, 6]  
[4, 1, 1]  [4, 1, 2]  [4, 1, 3]  [4, 1, 4]  [4, 1, 5]  [4, 1, 6]  
[4, 2, 1]  [4, 2, 2]  [4, 2, 3]  [4, 2, 4]  [4, 2, 5]  [4, 2, 6]  
[4, 3, 1]  [4, 3, 2]  [4, 3, 3]  [4, 3, 4]  [4, 3, 5]  [4, 3, 6]  
[4, 4, 1]  [4, 4, 2]  [4, 4, 3]  [4, 4, 4]  [4, 4, 5]  [4, 4, 6]  
[4, 5, 1]  [4, 5, 2]  [4, 5, 3]  [4, 5, 4]  [4, 5, 5]  [4, 5, 6]  
[4, 6, 1]  [4, 6, 2]  [4, 6, 3]  [4, 6, 4]  [4, 6, 5]  [4, 6, 6]  
[5, 1, 1]  [5, 1, 2]  [5, 1, 3]  [5, 1, 4]  [5, 1, 5]  [5, 1, 6]  
[5, 2, 1]  [5, 2, 2]  [5, 2, 3]  [5, 2, 4]  [5, 2, 5]  [5, 2, 6]  
[5, 3, 1]  [5, 3, 2]  [5, 3, 3]  [5, 3, 4]  [5, 3, 5]  [5, 3, 6]  
[5, 4, 1]  [5, 4, 2]  [5, 4, 3]  [5, 4, 4]  [5, 4, 5]  [5, 4, 6]  
[5, 5, 1]  [5, 5, 2]  [5, 5, 3]  [5, 5, 4]  [5, 5, 5]  [5, 5, 6]  
[5, 6, 1]  [5, 6, 2]  [5, 6, 3]  [5, 6, 4]  [5, 6, 5]  [5, 6, 6]  
[6, 1, 1]  [6, 1, 2]  [6, 1, 3]  [6, 1, 4]  [6, 1, 5]  [6, 1, 6]  
[6, 2, 1]  [6, 2, 2]  [6, 2, 3]  [6, 2, 4]  [6, 2, 5]  [6, 2, 6]  
[6, 3, 1]  [6, 3, 2]  [6, 3, 3]  [6, 3, 4]  [6, 3, 5]  [6, 3, 6]  
[6, 4, 1]  [6, 4, 2]  [6, 4, 3]  [6, 4, 4]  [6, 4, 5]  [6, 4, 6]  
[6, 5, 1]  [6, 5, 2]  [6, 5, 3]  [6, 5, 4]  [6, 5, 5]  [6, 5, 6]  
[6, 6, 1]  [6, 6, 2]  [6, 6, 3]  [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?
• **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
A general pseudo-code algorithm for backtracking problems:

Explore(\texttt{choices}):

- if there are no more \texttt{choices} to make: stop.

- else:
  
  - Make a single choice \texttt{C}.
  
  - Explore the remaining \texttt{choices}.
  
  - Un-make choice \texttt{C}, if necessary. (backtrack!)
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, 1, 1 | 1 die

1, 1, 1, 1 | 1 die

1, 1, 1, 2 | 1 die

1, 1, 2 | 1 die

1, 1, 3 | 1 die

1, 1, 3, 1 | 1 die

1, 2 | 2 dice

1, 3 | 2 dice

1, 4 | 2 dice

2 | 3 dice
Private helpers

- Often the method doesn't accept the parameters you want.
  - So write a **private helper** that accepts more parameters.
  - Extra params can represent current state, choices made, etc.

```csharp
public int methodName(params):
    ...
    return helper(params, moreParams);

private int helper(params, moreParams):
    ...
    (use moreParams to help solve the problem)
```
Exercise solution

public static void diceRolls(int dice) {
    List<Integer> chosen = new ArrayList<Integer>();
    diceRolls(dice, chosen);
}

private static void diceRolls(int dice, List<Integer> chosen) {
    if (dice == 0) {
        System.out.println(chosen);   // base case
    } else {
        for (int i = 1; i <= 6; i++) {
            chosen.add(i);               // choose
            diceRolls(dice - 1, chosen); // explore
            chosen.remove(chosen.size() - 1); // un-choose
        }
    }
}
Write a method `diceSum` similar to `diceRoll`, but it also accepts a desired sum and prints only combinations that add up to exactly that sum.

`diceSum(2, 7);`

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

`diceSum(3, 7);`

- `[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]`
- `[3, 1, 3]`
- `[3, 2, 2]`
- `[3, 3, 1]`
- `[4, 1, 2]`
- `[4, 2, 1]`
- `[5, 1, 1]`
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>

1, 1 2 dice  2, 2 dice  3, 2 dice  4, 2 dice  5, 2 dice  6, 2 dice

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

...
Optimizations

• We need not visit every branch of the decision tree.
  – Some branches are clearly not going to lead to success.
  – We can preemptively stop, or prune, these branches.

• Inefficiencies in our dice sum algorithm:
  – Sometimes the current sum is already too high.
    • (Even rolling 1 for all remaining dice would exceed the desired sum.)
  – Sometimes the current sum is already too low.
    • (Even rolling 6 for all remaining dice would exceed the desired sum.)
  – When finished, the code must compute the sum every time.
    • (1+1+1 = ..., 1+1+2 = ..., 1+1+3 = ..., 1+1+4 = ..., ...)
Exercise solution, improved

```java
public static void diceSum(int dice, int desiredSum) {
    List<Integer> chosen = new ArrayList<Integer>();
    diceSum(dice, desiredSum, chosen, 0);
}

private static void diceSum(int dice, int desiredSum, List<Integer> chosen, int sumSoFar) {
    if (dice == 0) {
        if (sumSoFar == desiredSum) {
            System.out.println(chosen);
        }
    } else if (sumSoFar <= desiredSum && sumSoFar + 6 * dice >= desiredSum) {
        for (int i = 1; i <= 6; i++) {
            chosen.add(i);
            diceSum(dice - 1, desiredSum, chosen, sumSoFar + i);
            chosen.remove(chosen.size() - 1);
        }
    }
}
```
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:

    ```java
    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
    ```
Experiencing 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></td>
<td>TEAM</td>
</tr>
</tbody>
</table>

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);
            chosen += c;
            permute(s, chosen);
            // explore
            s = s.substring(0, i) + c + s.substring(i + 1);
            chosen = chosen.substring(0, chosen.length() - 1);
            // un-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
            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)
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:
    ```java
    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.
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);
            }
        }
    }
}
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);
            }
        }
    }
}