Recursion

CMSC 132

Department of Computer Science

University of Maryland, College Park

Overview

- Recursion
- Base Case/Recursive Step
- Recursion vs. Iteration
- Examples

Definition

- Recursion (a.k.a – divide and conquer)
- Strategy to solve problems

  if (problem instance is simple/trivial)

  Solve it directly

  else

  1. Break the problem instance into smaller instances of the original problem
  2. Solve each smaller instance
  3. Combine solutions to solve original problem
- Example: Single person moving 100 books from one room to another

Recursive Method

- Recursion relies on the call stack.
- Let's review how the stack is used in a sequence of nested method calls
- Classical recursive example is the computation of the factorial.
- Let's see that code
- Let's draw a memory diagram illustrating the computation of factorial(4)

Base Case/Recursive Step

- Base case – solution to small problem
- Recursive step
  - Simplifies original problem
  - Recursively applies current algorithm to sub-problem
  - Combine results if necessary
- A recursive method is a method that “calls itself”
- Alternative definition: A recursive method is a method that calls a method that performs the same task the current method performs

Recursion vs. Iteration

- Any problem you can solve recursively can be solved without recursion
- Iteration over a list/sequence is very similar to using recursion to combine the result from the first element with the result from the rest of the list
- Counting vowels with recursion

```
int countVowels(String s) {
  if (s.length() == 0) return 0;
  int tailResult = countVowels(s.substring(1));
  switch (s.charAt(0)) {
    case 'a': case 'e': case 'i': case 'o': case 'u':
      return tailResult+1;
    default:
      return tailResult;
  }
}
```

Recursion vs. Iteration

- Counting vowels with iteration

```
int countVowels(String s) {
  int result = 0;
  for(int i = 0; i < s.length(); i++) {
  switch (s.charAt(i)) {
    case ‘a’: case ‘e’: case ‘i’: case ‘o’: case ‘u’:
      result++; break;
    }
  return result;
}
```

Recursion vs. Iteration

- Recursion can be expensive (see naive implementation of fibonacci in code distribution (Fibonacci.java))
- We can solve fibonacci efficiently by using dynamic programming
- Recursion can impose limits on your computations due to the stack limitation

How many ways are there to order the numbers 1..n?

- permutation(1) = 1
- permutation(n) =
  - n different numbers that could occur first
  - remaining elements could appear in permutation(n-1) different orders
- thus, permutation(n) = n * permutation(n-1)
  = n!
- See PrintPermutations.java in code distribution

Recursive Permutation Generation

- Define a function

  `calculatePermutations(int [] a)`

- that generates each permutation of a and calls

  `handlePermutation(int [] a)`

  with each generated permutation

- Hint: define a recursive helper function

  `calculatePermutations(int [] a, int k)`

  that permutes the values in a[k ... a.length-1]

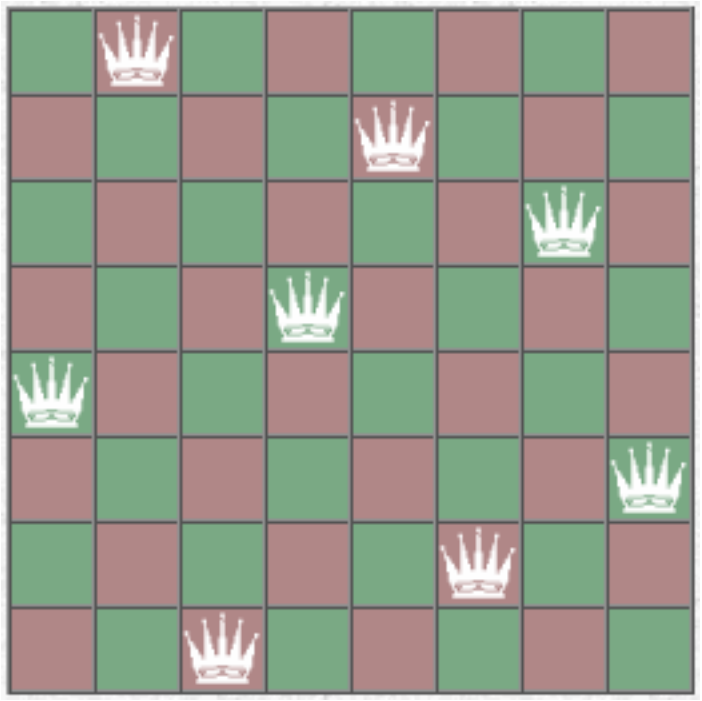
- See PrintPermutations.java in code distribution

Not Exciting Recursion

- Using recursion where iteration would work just as well isn't that exciting
  - Sometimes, a little to think about it as a recursive problem
  - In Java, it isn't particularly efficient for long lists or deep recursion
  - In some languages (such as Scheme) that support tail recursion, it is as efficient as iteration. In fact, it can be the only/preferred way to implement iteration
    - has to be a special kind of recursion, called tail recursion
- Interesting recursion
  - Binary search
  - Quicksort
  - Mergesort

nQueens

- Place queens on a board such that every row and column contains one queen, but no queen can attack another queen
  - place queens on nxn board
  - recursive approach: assume you've already placed k queens



Change making

- We've been asked to help devise a new set of coins for a country ruled by a mathematician.
- Our task is to figure out what the denominations of the coins should be.
  - For example, in the US, the coins have denominations (in pennies) of: 1, 5, 10, 25 (we'll ignore half dollars for now)
- Assume we want to have only three coins.
- What values should the coins have so as to minimize the average number of coins needed to make any number of cents from 0 to 99.

Greedy vs. Non-Greedy

- **Greedy approach:**
  - use as many of the largest coin possible, then go to the next largest coin
- **Greedy is not always optimal**
  - If your coins are 25, 10, 1, then to make change for 30 cents, the greedy approach requires 6 coins whereas only 3 dimes are required
- **For US coins, greedy is optimal.**

Fractals

- See fractal package in code distribution