CSE 142

Introduction to Recursion

Outline for Today

- Review
  - Method calls and scope
  - Static methods
- Today
  - Recursion – methods that call themselves
  - Recursive and base cases
  - Implementation in Java

Method Calls and Static Methods (Review)

- Recall that a static method is one that is associated with a class, not a particular instance of a class
- Often used for computations that are not naturally associated with some object

```
public class Math {
    /** return the square root of x */
    public static double sqrt(double x) { ... }
    /** return the trigonometric sin of theta */
    public static double sin(double theta) { ... }
}
```

- Use

```
double sqrt2 = Math.sqrt(2.0);
```


Recursive Definitions

- Classic example: factorial
- Mathematical definition

$$n! = \begin{cases} 1 & n \le 1 \\ n*(n-1)! & otherwise \end{cases}$$

- Example

Factorial in Java

- Could write a loop to multiply 1 * 2 * 3 * ... * *n*
- Can also use the recursive definition directly!

```
/** return n! = 1 * 2 * 3 * ... * n */
public static int factorial(int n) {



}
```


Trace

- Evaluate

```
factorial(4)
```

How Can This Possibly Work?

- This is an example of a ***recursive*** method call – a method that calls itself as part of its implementation
- There is nothing really new here. A method call works as it always does:
  - First, allocate a new scope for the method's parameters and local variables
  - Second, initialize parameters with method call arguments
  - Third, begin execution of the method body
- Recursive methods work exactly the same
- Also works fine for non-static methods

Method Call Trace

- Evaluate factorial(4)

Recursive and Base Cases

- A recursive definition always has two parts
  - One or more ***recursive*** cases where the method calls itself
  - One or more ***base*** cases that return a result without an additional recursive call
- Rules
  - There *must* be at least one base case
  - Each recursive case *must* make progress towards reaching a base case
- Forgetting either one of these rules is a common source of errors in recursive methods
  - In particular, "infinite" recursion – never reaching a base case; each call generates yet another recursive call

Towers of Hanoi

- Classic problem
- Setup
  - Three pegs
  - Set of disks of different diameters, initially on one peg with disks stacked in order – largest on bottom, smallest on top
- Problem: move all of the disks from the initial peg to one of the other two, without ever placing a larger disk on top of a smaller one
- Can you think of an algorithm to do this?
  - Hint: recursion *is* your friend

Algorithm for Towers of Hanoi

- Your algorithm here

Demonstration

Iteration vs Recursion

- Turns out that any iterative algorithm can be reworked as a recursive algorithm, and vice versa
  - Use recursive calls wherever “looping” is needed
- Sometimes this is straightforward – e.g., factorial
- Sometimes less obvious – how would you implement towers of Hanoi iteratively?
  - A non-recursive solution to a naturally recursive problem often requires extra bookkeeping to keep track of what’s been done already and what needs to be done

When to Use Recursion

- Recursion is a natural fit for problems that…
  - … have one or more simple cases with a straightforward non-recursive solution (base cases)
  - … have other cases that can be redefined as simpler versions of the original problem, and repeating these redefinitions gets closer to one of the simple cases (recursive cases)
- Take advantage of recursion when the problem matches

  (Usually – there are occasions where a naturally recursive implementation is too slow or has too much overhead)

Example: QuickSort

- Supposed we are asked to sort a list
- QuickSort is an very fast algorithm that makes use of recursion

```
/** Sort the items in list[from] to list[to] */
public void QuickSort(int from, int to) {
    if (to – from <= 1) {
      return;                  // base case – at most one item; nothing to do
    } else {
      pick some element x from list[from] to list[to];
      rearrange the list so that x is in position mid, and all items in list[from] to
               list[mid-1] are <= x and all items from list[mid+1] to list[to] are > x;
      QuickSort(from, mid-1);
      QuickSort(mid+1, to);
    }
}
```


QuickSort Example

Another Problem – Path Planning

- Idea: want to discover if there is a path from square at 0, 0 to square labeled F (which could be anywhere)
- Black squares represent obstacles
- Unless a path is blocked, can move up, down, left, or right
- Can you design an algorithm for this?
  - Hint: can you use recursion to help?
- Answer next time