1.00 Lecture 12

Recursion

Reading for next time: Big Java: sections 11.1-11.4

Recursion

- Recursion is a divide-and-conquer (or divide-and-combine) approach to solving problems:

```
method Recurse(Arguments)
        if (SmallEnough(Arguments))                    // Termination
                return Answer
        else                                           // "Divide"
                Identity= Combine( SomeFunc(Arguments),
                          Recurse(SmallerArguments))
        return Identity                                // "Combine"
```

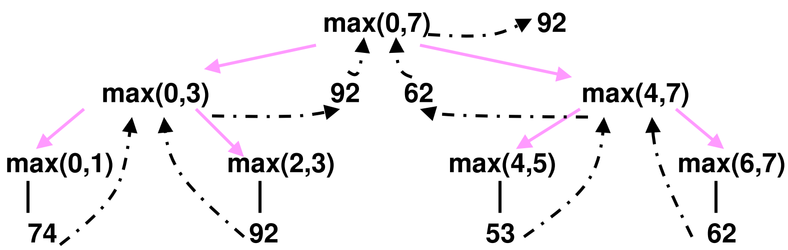
- If you can write a problem as the combination of smaller problems, you can implement it as a recursive algorithm in Java

Finding maximum of array

Assume we can only find max of 2 numbers at a time. Suppose we want to find the max of a set of numbers, say 8 of them.

35 74 32 92 53 28 50 62

Our recursive max method calls itself:



Code for maximum method

```
public class MaxRecurse {
    public static void main(String[] args) {
        int[] AData= {35, 74, 32, 92, 53, 28, 50, 62};
        System.out.println("Max: " + maxArray(0, 7, AData));
    }

    public static int combine(int a, int b) {
        if (a >= b) return a;
          else return b;
    }

    public static int maxArray( int i, int j, int[] Arr) {
        if ( (j - i) <= 1) {          // Small enough
            if (Arr[j] >= Arr[i])
                return Arr[j];
            else
                return Arr[i]; }
        else                                // Divide and combine
            return (combine(maxArray(i, (i+j)/2, Arr),
                            maxArray((i+j)/2+1, j, Arr)));
    }
}
```

Maximum code with more output

```
public class MaxRecurse2 {
  public static void main(String[] args) {
      int[] AData= {35, 74, 32, 92, 53, 28, 50, 62};
      System.out.println("Main Max:" + maxArray(0, 7, AData)); }
  public static int combine(int a, int b) {
      if (a>=b) return a;
        else return b;  }
  public static int maxArray( int i, int j, int[] Arr) {
      System.out.println("Max(" + i + "," + j + ")");
      if ( (j - i) <= 1) {
          if (Arr[j] >= Arr[i]) {                   // Small enough
              System.out.println("  " + Arr[j]);
              return Arr[j]; }
          else {
              System.out.println("  " + Arr[i]);
              return Arr[i]; } }
      else {                                        // Divide, combine
          int aa= (combine(maxArray(i, (i+j)/2, Arr),
                          maxArray((i+j)/2+1, j, Arr)));
          System.out.println("Max(" +i + "," +j + ")= "+ aa);
          return aa;
    }  }  }
```

Exponentiation

- Exponentiation, done ‘simply’, is inefficient
  - Raising x to y power can take y multiplications:
    - E.g., $x^7 = x * x * x * x * x * x * x$
  - Successive squaring is much more efficient, but requires some care in its implementation
  - For example: $x^{48} = ((((x * x * x)^2)^2)^2)^2$ uses 6 multiplications instead of 48
- Informally, simple exponentiation is O(n)
  - Squaring is O(lg n), because raising a number to the n^{th} power take about lg n operations (base 2)
    - $Lg(48) = Log_2(48)$ = about 6
    - $2^5 = 32$; $2^6 = 64$
  - To find $x^{1,000,000,000}$, squaring takes 30 operations while the simple method takes 1,000,000,000!

Exponentiation cont.

- Odd exponents take a little more effort:
  - $x^7 = x * (x*x*x)^2$ uses 4 operations instead of 7
  - $x^9 = x * ((x*x)^2)^2$ uses 4 operations instead of 9
- We can generalize these observations and design an algorithm that uses squaring to exponentiate quickly
- Writing this with iteration and keeping track of odd and even exponents can be tricky
- It is most naturally written as a recursive algorithm
  - We write a series of 3 identities and then implement them as a Java function!

Exponentiation, cont.

- Three identities:
  - $x^1 = x$ (small enough)
  - $x^{2n} = x^n * x^n$ (reduces problem)
  - $x^{2n+1} = x * x^{2n}$ (reduces problem)

Exercise

- **Write pseudocode for exponentiation**
  - Write your pseudocode on paper or Eclipse
  - Use the standard pattern:
  - You can write the identities as expressions; you don't have to use a 'Combine' method
    - 'Combine' is usually just * or + or Math.max()…

```
method Recurse(Arguments)
        if (SmallEnough(Arguments))            // Termination
                return Answer
        else                                    // "Divide"
                Identity= Combine( SomeFunc(Arguments),
                                   Recurse(SmallerArguments))
        return Identity                         // "Combine"
```

How the recursion works

x= 5, y= 9

ExpResult(5, 9)

5 * ExpResult (5, 8) = 1953125

square(ExpResult(5, 4)) = 390625

square(ExpResult(5, 2)) = 625

square(ExpResult(5, 1)) = 25

ExpResult(5, 1) =5

Exponentiation Exercise

```
// Download Exponentiation class and complete it

import javax.swing.*;

public class Exponentiation {
    public static void main(String[] args) {
        int z;
        String input= JOptionPane.showInputDialog("Enter x");
        int x= Integer.parseInt(input);
        input= JOptionPane.showInputDialog("Enter y");
        int y= Integer.parseInt(input);
        z= expResult(x, y);
        System.out.println(x + " to " + y + " power is: " + z);
    }

// You can use BigInteger to handle large numbers. A bit clumsy.
```

Exponentiation Exercise, p.2

```
public static int expResult(int x, int y) {
   int result;

   // Write code when y is small enough

   // Write code when we need to divide the problem further

   // Add System.out.println as desired to trace results

   return result;
   }
}
```

Recursion and iteration

- It's a tricky exercise to write the exponentiation iteratively
  - Try it if you have time and are interested!
- It's often easier to see a correct recursive implementation
  - Recursion is often closer to the underlying mathematics
- There is a mechanical means to convert recursion to iteration, used by compilers and algorithm designers. It's complex, and is used to improve efficiency.
  - Overhead of method calls is noticeable, and converting recursion to iteration within a method speeds up execution
  - Small or infrequently used methods can be left as recursive

Exercise 1

- An example sequence is defined as:
  - $q_0 = 0$
  - $q_n = (1 + q_{n-1})^{1/3}$
- Write a recursive method to compute q_n
- Download Sequence1 (or type it from next page)
  - Main is written for you
    - Write method q() in class Sequence1. q() is a method in Sequence1, just like main()
  - The recursive method 'signature' is written also
  - The body of the recursive method follows the template:
    - If small enough, determine value directly
    - Otherwise, divide and combine
  - Use `Math.pow(base,exponent)` to take the cube root
    - Remember to make the exponent 1.0/3.0, not 1/3
- Save/compile and run or debug it
  - Try n= 10, or n= 20

Download Code 1

```
import javax.swing.*;

public class Sequence1 {
   public static void main(String[] args) {
        String input= JOptionPane.showInputDialog("Enter n");
        int n= Integer.parseInt(input);
        double lastTerm= q(n);
        System.out.println("Last term: "+ lastTerm);
   }
   public static double q(int n) {
        // Write your code here
        // Put in System.out.printlns when you return values
        }

// Sample output:
 n: 0 answer: 0.0
 n: 1 answer: 1.0
 n: 2 answer: 1.2599210498948732
 n: 3 answer: 1.3122938366832888
```

Exercise 2

- A second sequence is defined as:
  - $q_0 = 0$
  - $q_1 = 0$
  - $q_2 = 1$
  - $q_n = q_{n-3} + q_{n-2}$ for n >= 3
- Write a recursive method to compute q_n
- Download Sequence2 (or type it from next page)
  - Main is written for you
    - Write method q() in class Sequence2. q() is a method in Sequence2, just like main()
  - The recursive method 'signature' is written also
  - The body of the recursive method follows the template:
    - If small enough, determine value directly
    - Otherwise, divide and combine
- Save/compile and run or debug it
  - Try n= 10, or n= 20

Download Code 2

```
import javax.swing.*;

public class Sequence2 {
   public static void main(String[] args) {
       String input= JOptionPane.showInputDialog("Enter n");
       int n= Integer.parseInt(input);
       for (int i= 0; i < n; i++) // Call it for all i<=n
             System.out.println("i: "+ i + " q: " + q(i));
   }
   public static int q(int n) {
       // Write your code here
   }
}
// Sample solution
i: 0 q: 0
i: 1 q: 0
i: 2 q: 1
i: 3 q: 0
i: 4 q: 1
```

Exercise 3

- A pair of sequences is defined as:
  - $x_0 = 1$; $x_n = x_{n/2} + y_{n/3}$
  - $y_0 = 2$; $y_n = x_{n/3} * y_{n/2} + 2$ (Note the *, not +)
- Write two recursive methods to compute x_n and y_n
  - Subscripts n/2 and n/3 use integer division
- Download Sequence3 (or type it from next page)
  - Main is written for you
    - Methods x() and y() are methods in class Sequence3, just like main().
  - The bodies of the recursive methods follow the template:
    - If small enough, determine value directly
    - Otherwise, divide and combine
- Save/compile and run or debug it
  - Try n= 10, or n= 20

Download Code 3

```
import javax.swing.*;

public class Sequence3 {
   public static void main(String[] args) {
       String input= JOptionPane.showInputDialog("Enter n");
       int n= Integer.parseInt(input);
       System.out.println("i x y");
       for (int i= 1; i <= n; i++)
           System.out.println(i + " " + x(i) + " " + y(i));
   }
   // Write your methods for x(i) and y(i) here
}
// Sample solution
i x y
1 3 4
2 5 6
3 7 14
4 9 20
5 9 20
```