Introduction to Algorithms and Data Structures

Lecture 9 - Deja Vu All Over Again:
An Introduction to Recursion

What is Recursion?

• Recursion involves defining a solution to a problem in terms of another case of the problem except for one simple (or more) simpler cases in which the solution is defined explicitly.

• Example - Factorial

\[ n! = n \cdot (n-1)! \quad \text{for } n > 0 \]
\[ = 1 \quad \text{for } n = 0 \]

• We can write functions that call themselves recursively as long as they lead to a simple case that we can use as the basis for a solution.
The Original **factorial** Program

```java
public class TestFactorial {
    // main() - A Driver for our factorial
    //          function
    public static void main(String[] args) {
        Factorial f = new Factorial();
        System.out.println(f.factorial(5));
    }
}
```

```java
public class Factorial {
    // factorial() - A Recursive solution for n!
    public double factorial(int n) {
        int nless1;
        double f;
        if (n == 0) {
            // The simple case
            return 1.0;
        } else {
            // The recursive solution
            nless1 = n - 1;
            f = factorial(n-1);
            return n*f;
        }
    }
}
```
Tracing \texttt{factorial}

\begin{align*}
\begin{array}{c|c|c}
 n & nless1 & f \\
5 & 4 & 5 \\
4 & 3 & 4 \\
3 & 2 & 3 \\
2 & 1 & 2 \\
1 & 0 & 1 \\
0 & 1 & 0 \\
\end{array}
\end{align*}
Tracing \textit{factorial} (continued)

<table>
<thead>
<tr>
<th>n</th>
<th>nless1</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n</th>
<th>nless1</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4</td>
<td>24</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Tracing \textit{factorial} (continued)

<table>
<thead>
<tr>
<th>n</th>
<th>nless1</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
The Traceable Factorial Program

// factorial() - A Recursive solution for n!
public double factorial(int n) {
    int nless1;
    double f;
    if (n == 0)  // The simple case
        return(1.0);
    else {
        // The recursive case
        nless1 = n - 1;
        f = factorial(n-1);
        System.out.println(nless1 + "! = " + f);
        return(n*f);
    }
}

Tracing fact2

Output
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
120
A simpler factorial

// factorial() - A Recursive solution for n!
public double factorial(int n) {
    if (n == 0)  
        // The simple case
        return(1.0);
    else
        // The recursive case
        return(n*factorial(n-1));
}

Example - Greatest Common Divisor

• The solution is defined as:
  gcd(x, y) = gcd(y, x)  if x < y
  y  if x MOD y = 0
  gcd(y, x MOD y) if x >= y
    & x MOD y ≠ 0
The GCD main program

```java
public class TestGcd {
    // main() - A Driver for the GCD function
    public static void main(String[] args) {
        Gcd g = new Gcd();
        final int x = 81, y = 180;

        System.out.println("The greatest common" + " divisor of " + x + " and " + y + " is " + g.gcd(x, y));
    }
}

public class Gcd {
    // gcd() - Finds the Greatest Common Divisor
    public int gcd(int x, int y) {
        if (x < y) // Reverse the parameter // y cannot be larger
            return(gcd(y, x));
        else if (x % y == 0) // The simple case
            return(y);
        else // The Recursive case
            return(gcd(y, x % y));
    }
}
```
## Tracing gcd

\[
\begin{array}{cc|cc|cc|cc}
\hline
x & y & x & y & x & y & x & y \\
81 & 180 & 81 & 180 & 81 & 180 & 81 & 180 \\
180 & 81 & 180 & 81 & 81 & 18 & 81 & 18 \\
\hline
\end{array}
\]

## Tracing gcd (continued)

\[
\begin{array}{cc|cc|cc|cc}
\hline
x & y & x & y & x & y \\
81 & 180 & 81 & 180 & 81 & 18 & 18 & 9 \\
\hline
\end{array}
\]
A Traceable Form of `gcd`

```java
class Gcd {
    // gcd() - Finds the Greatest Common Divisor
    public int gcd(int x, int y) {
        if (x < y) {
            // Reverse the parameter
            // y cannot be larger
            System.out.println("x = " + x + ", " + y);
            return(gcd(y, x));
        }
        else if (x % y == 0) {
            // The simple case
            System.out.println("x = " + x + ", " + y);
            return(y);
        }
        else {
            // The Recursive case
            System.out.println("x = " + x + ", " + y);
            return(gcd(y, x % y));
        }
    }
}
```
Tracing gcd

Output
\[
\begin{align*}
  x &= 81 & y &= 180 \\
  x &= 180 & y &= 81 \\
  x &= 81 & y &= 18 \\
  x &= 18 & y &= 9 \\
\end{align*}
\]

The greatest common divisor of 81 and 180 is 9

Example - Fibonacci Numbers

- Fibonacci numbers start with 0 and 1 and in each subsequent case are the sum of the two previous numbers.
- Fibonacci numbers are defined by the relationship:
  \[
  \begin{align*}
  \text{Fib}_n &= \text{Fib}_{n-1} + \text{Fib}_{n-2} & \text{for } n > 1 \\
  &= n & \text{for } n \leq 1 \\
  \end{align*}
  \]
- Fibonacci numbers is different from our previous example because it is doubly-recursive, i.e., requires two recursive calls
TestFib.java

import java.util.Scanner;

public class TestFib {
    // main() - Drives the Fibonacci function
    public static void main(String[] args)
        throws IOException {
        Scanner keyb = new Scanner(System.in);
        Fib f = new Fib();
        int n;
        // Input n
        System.out.print("Which Fibonacci number do you want \n? ");
        n = keyb.nextInt();
        System.out.println("The number is ");
        System.out.println(f.fib(n));
    }
}

Fib.java

public class Fib {
    // fib() - A Recursive Fibonacci function
    public int fib(int n) {
        if (n <= 1)
            // The simple case
            return(n);
        else
            // The Recursive case
            return(fib(n-1) + fib(n-2));
    }
}

Program Run
Which Fibonacci number do you want ? 6
The 6th Fibonacci number is 8
The Traceable Fib - fib.java

```java
public class Fib {
    // fib() - A Recursive Fibonacci function
    public int fib(int n) {
        int fibnless1, fibnless2, fibn;
        if (n <= 1)
            return(n);
        else {
            fibnless1 = fib(n-1);
            fibnless2 = fib(n-2);
            fibn = fibnless1 + fibnless2;
            System.out.println("The " + n + "th Fibonacci number is " + fibn);
            return(fibn);
        }
    }
}
```

Tracing fib

<table>
<thead>
<tr>
<th></th>
<th>fibnless1</th>
<th>fibnless2</th>
<th>fibn</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>

```
Tracing \texttt{fib} (continued)

\begin{align*}
\begin{array}{c|c|c|c|c|c}
  n & \text{fibnless1} & \text{fibnless2} & \text{fibn} \\
  \hline
  3 & & & \\
  2 & & & \\
  1 & & & \\
\end{array}
\end{align*}

\begin{align*}
\begin{array}{c|c|c|c|c|c}
  n & \text{fibnless1} & \text{fibnless2} & \text{fibn} \\
  \hline
  3 & & & \\
  2 & 1 & & \\
  1 & 1 & & \\
\end{array}
\end{align*}

Tracing \texttt{fib} (continued)

\begin{align*}
\begin{array}{c|c|c|c|c|c}
  n & \text{fibnless1} & \text{fibnless2} & \text{fibn} \\
  \hline
  3 & & & \\
  2 & 1 & & \\
  0 & 0 & & \\
\end{array}
\end{align*}

\begin{align*}
\begin{array}{c|c|c|c|c|c}
  n & \text{fibnless1} & \text{fibnless2} & \text{fibn} \\
  \hline
  3 & & & \\
  2 & 1 & 0 & \\
  0 & 0 & 0 & \\
\end{array}
\end{align*}
Tracing $\text{fib}$ (continued)

<table>
<thead>
<tr>
<th>n</th>
<th>fibnless1</th>
<th>fibnless2</th>
<th>fibn</th>
<th>n</th>
<th>fibnless1</th>
<th>fibnless2</th>
<th>fibn</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Tracing $\text{fib}$ (continued)

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\text{fibnless1}$</th>
<th>$\text{fibnless2}$</th>
<th>$\text{fibn}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Tracing $\text{fib}$ (continued)

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\text{fibnless1}$</th>
<th>$\text{fibnless2}$</th>
<th>$\text{fibn}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Binary Search

• A binary search is a reasonably fast way to search an array of data items or structures to find one with a particular value or a particular value in one field (known as the \textit{key}).

• Although it can be written without recursion, the recursive version is easier to follow.

BinarySearch.java

public class BinarySearch {

    // The main binarySearch function and the
    // recursive function that it calls, named
    // binSearch

    // binarySearch() — A Binary Search function
    // that calls the recursive
    // function binsearch
    public int binarySearch(int key, int[] x,
                          int n) {
        int index;
        index = binSearch(key, x, 0, n-1);
        return(index);
    }
/ **binsearch**() - The recursive binary search
*/

private int binSearch(int key, int[] x,
                      int low, int high) {

  int mid;

  // Find the mid point
  mid = (low + high) / 2;
  if (low > high)
    // We've searched the whole array
    // - it isn't in it
    return(-1);
  else if (x[mid] == key)
    // We found it
    return(mid);

  else if (x[mid] < key)
    // Search the upper half of this section
    return(binSearch(key, x, mid+1, high));
  else
    // Search the lower half of this section
    return(binSearch(key, x, low, mid - 1));
}
}
A Driver for BinarySearch

```java
public class RunBinarySearch {
    public static void main(String[] args) {
        TestBinarySearch tbs = new TestBinarySearch();
        tbs.run(13);
    }
}
```

```java
import java.util.Scanner;
// TestBinarySearch - A Driver for the binary search function
//
public class TestBinarySearch {
    public static void run(int key) {
        int i = 0;
        int[] x = new int[10];
        BinarySearch bs = new BinarySearch();

        for (i = 0; i < 10; i++) {
            x[i] = 2*i + 5;
            System.out.println("x[" + i + "] = " + x[i]);
        }
        System.out.println("\n\n\n");
        i = bs.binarySearch(key, x, 10);
        System.out.println("x[" + i + "] = 13");
    }
}
```
Tracing `binSearch`

```
5 7 9
11 13 15 17 19 21 23
```

`low` -> Returns 4

`mid` ->

`high` ->

RunBinarySearch with 19

```java
class RunBinarySearch {
    public static void main(String[] args) {
        TestBinarySearch tbs = new TestBinarySearch();
        tbs.run(19);
    }
}
```
Tracing RunBinarySearch with 19

Returns 7

RunBinarySearch with 22

```java
public class RunBinarySearch {
    public static void main(String[] args) {
        TestBinarySearch tbs = new TestBinarySearch();
        tbs.run(22);
    }
}
```
Tracing RunBinarySearch with 22 (continued)

Returns -1
Towers of Hanoi

• The Towers of Hanoi is a classical example of a problem that is difficult to solve without recursion.
• The Towers consists of three pegs, one of which has a series of disks placed on it, with the disks stacked with smaller disks on top of larger disks.
• The challenge is to move them from one peg to another using an extra peg:
  – moving them one at a time
  – no disk can be placed on top of a smaller disk.

Solving the Towers of Hanoi

• Our solution involves three steps:
  1. Move all but the bottom disk to the extra peg
  2. Move the bottom disk to its final location.
  3. Move the other disks on top of the bottom disk.
• This is the basis of our recursive solution.
Our Recursive Solution - Initially

Our Recursive Solution - After Step 1
TestTowers.java

import java.util.Scanner;
public class TestTowers {
    // main() - Runs the recursive solution to
    //           the famous Towers of Hanoi problem

    public static void main(String[] args) {
        int numDisks;
        Scanner keyb = new Scanner(System.in);
        Towers tower = new Towers();
        // Get the number of disks to be used
        System.out.println("How many disks\t(1-9)?");
        numDisks = keyb.nextInt();

        // Solve the problem
        tower.towersOfHanoi('A', 'B', 'C', numDisks);
    }
}

Towers.java

public class Towers {
    // TowersOfHanoi() - The recursive solution to
    //                   the famous Towers of
    //                   Hanoi problem
    public void towersOfHanoi(char origin,
                                char destination, char via, int disks) {
        // Use the recursive solution for more than
        // one disk
        if (disks > 1) {
            // Move all but the bottom to the auxiliary
            // peg "via"
            // Move the bottom disk to the
            // destination peg
            // Move the rest on top of the bottom peg
        }
    }
}
towersOfHanoi(origin, via, destination, disks - 1);
System.out.println("Move disk #" + disks + " from " + origin + " to " + destination);
towersOfHanoi(via, destination, origin, disks - 1);
}
else
  // Move the only disk into position
  System.out.println("Move disk #1 from " + origin + " to " + destination);
}