Lesson-4-Class Notes & Review

Slide-4

Iterative Solution

```
F = 1;
For I = 1 to n
  F = f * I
Console.log(f)
```

Find a Factorial of a given number n. (n>=0)

1! = 1
2! = 1 * 2
3! = 1 * 2 * 3
4! = 4 * 3!
n! = n * (n-1)!

The factorial of *n*, or *n*! is defined as follows:

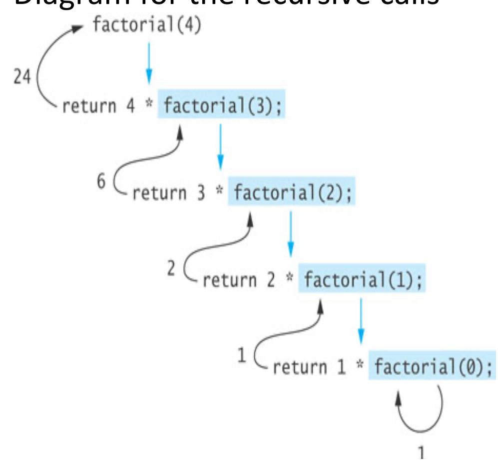
0! = 1
n! = *n* x (*n* -1)! (n > 0)

The base case: *n* is equal to 0
The second formula is a recursive definition.

// Example of Linear Recursion

```
Algorithm factorial(n)
    // Base case to stop recursion
        if (n = 0)
            return 1;
        else
        // Smaller version of itself / Recursive call
            return n * factorial(n – 1);
```

Diagram for the recursive calls



<u>Slide-6 – Linear Recursion</u>

Algorithm sumFirst(n)
  if n < 0 then Throw InvalidInputException
  if n = 0 then
    return 0
  else
    return n + sumFirst(n-1)

Trace Calls for n = 5, after the call return

|--- 0 (base case) 0
|--- 1 + sumFirst(0) 1
|--- 2 + sumFirst(1)
|--- 3 + sumFirst(2) 3
|--- 4 + sumFirst(3) 7
|--- 5 + sumFirst(4) 12

Stack Calls --> All the method calls are maintained in a stack by introducing stack frame, which includes all the arguments and local variables status. Once the method is returned that call is removed from the stack.

Stack method Call	**Return**
~~**sumFirst(0) Return 0**~~	**0 Call removed from the stack**
~~**sumFirst(1) n = 1**~~	**N+ Previous call answer 0 + 1**
sumFirst(2) n= 2	**N+ Previous call answer 1 + 2**
sumFirst(3) n=3	**N+ Previous call answer**
sumFirst(4) n=4	
sumFirst(5) n = 5	
Main()	

Slide-7 – Tail Recursion

```
Algorithm sumFirstHelper(n, s)
   if n = 0 then
          return s
   else
          return sumFirstHelper(n-1, n+s)
```

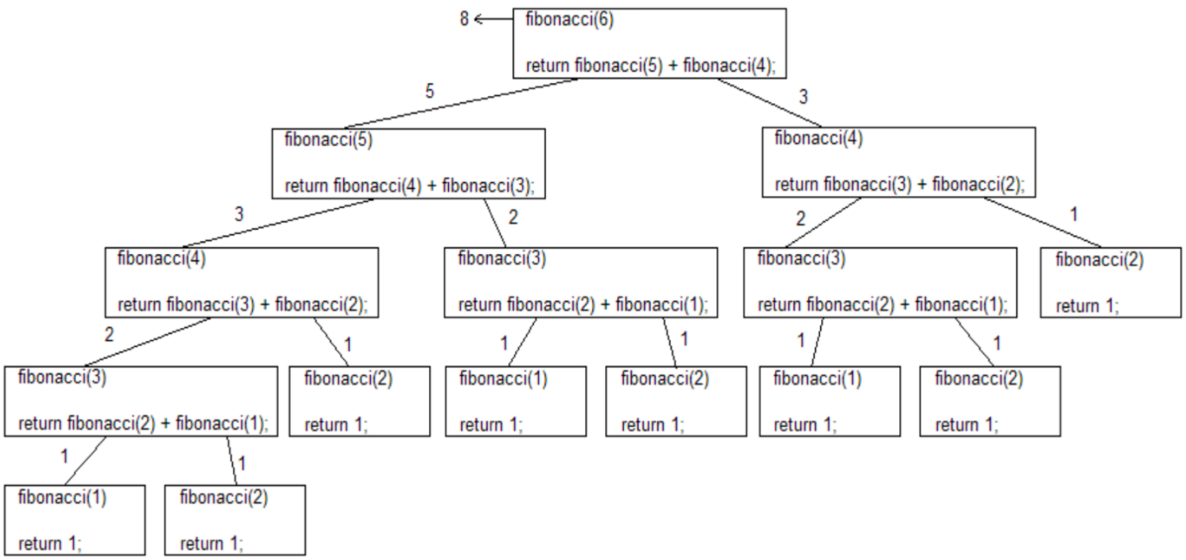
Trace Calls for n=5

```
               |--- 15 (base case)
            |--- sumFirstHelper(0, 1+14)
         |--- sumFirstHelper(1, 2+12)
      |--- sumFirstHelper(2, 3+9)
   |--- sumFirstHelper(3, 4+5)
|--- sumFirstHelper(4, 5+0)
```

Linear Recursion	Tail Recursion
Linear recursion occurs when methods call themselves only once inside a body. when referring to methods that process the result of the recursive call somehow before producing or returning its own output. The subproblem is added to n, in order to generate the function's result. n + sumFirst(n-1)	Tail recursion also known as Final Recursion. Methods that fall into this category also call themselves once inside a body, but the recursive call is the last operation carried out in the recursive case. Therefore, they do not manipulate the result of the recursive call. The recursive case is simply specifying relationships between sets of arguments for which the function returns the value. As the algorithm carries out recursive calls it modifies the arguments until it is possible to compute the solution easily in a base case. sumFirstHelper(n-1, n+s)

Slide-8

Algorithm Fib(n)
   if n = 0 then
        return 0
   else if n = 1 then
        return 1
   else
        return Fib(n-2) + Fib(n-1)



Slide-9 – Mutual Recursion

Problem: Check the given number is even or odd

Algorithm isEven(n)
   if n = 0 then
         return true
   else
         return isOdd(n-1)
	Algorithm isOdd(n)
	   if n = 0 then
	         return false
	   else
	         return isEven(n-1)

Call Trace for n = 3

isEven(3) calls isOdd(2).
isOdd(2) calls isEven(1).
isEven(1) calls isOdd(0).
isOdd(0) is the base case and returns false because 0 is not odd.

Slide-13-In-Class Exercise Solution

```
Algorithm sum(L)
    if L.isEmpty() then return 0
      return sumHelper(L, L.first())

Algorithm sumHelper(L, p)
    if L.isLast(p) then
        return p.element()
    else
        sum := sumHelper(L, L.after(p))
        return sum + p.element()
```

Trace Calls of L = { 1,2,3,4,5}

```
            |--- Returning sum of 1 + 14 = 15
          |--- Returning sum of 2 + 12 = 14
        |--- Returning sum of 3 + 9 = 12
      |--- Returning sum of 4 + 5 = 9
      |--- 5 (base case)   sum = 5
    |--- sumHelper(Element 4)
  |--- sumHelper(Element 3)
 |--- sumHelper(Element 2)
|--- sumHelper(Element 1)
```

Searching Algorithms

1. Linear Search (O(n)) – search from 0 position to n-1 position, not necessary in sorted order.
2. Binary Search (O(log n))
    a. Your inputs need to be in sorted order.
    b. Find the middle value
    c. Divide the array into two parts,
        i. Search either in the left or right with the help middle value

Binary Search Runtime Observation. Finding the upper bound, Worst-case Analysis. (finding an element not in the list). – Optional Part

1	2	3	4	5	6	7	8
Ind 0	1	2	3	4	5	6	7

Arr = [1 2 3 **4** 5 6 7 8] (n= 8). Low =0, upper = 7, mid = (0+7)/2 = 3

Search right side [5 **6** 7 8] (n = 4) Low =4, upper = 7, mid = (4+7)/2 = 5

Search right side. [**7** 8] (n=2). Low =6, upper = 7, mid = (6+7)/2 = 6

Search right side. [**8**] (n=1) Low =7, upper = 7, mid = (7+7)/2 = 7
Next step will reach base case
Low =8, upper = 7, mid = (8+7)/2 = 7

Here's an example with n = 8

1. Start with n = 8.
2. Divide by 2: n = 8 / 2 = 4.
3. Divide by 2: n = 4 / 2 = 2.
4. Divide by 2: n = 2 / 2 = 1.

In each recursive call, the size of the subarray is reduced using the result of the previous comparison.

- Initial length of array $=n$
- Self-call 1 - Length of array $= n/2$
- Self-call 2 - Length of array $=(n/2)/2= n/2^2$
- Self-call m - Length of array $=n/2^m$

- After m self-calls, the size of the array becomes 1.

Now, let's analyze how many steps it takes to reduce n to 1.

Length of array $=n/2^m=1$
$\Rightarrow n=2^m$

Each step involves dividing n by 2, which corresponds to taking the logarithm base 2 of n. Applying log function on both sides:
$\Rightarrow \log_2(n)= log_2(2^m)$
$\Rightarrow \log_2(n)= m*log_2 2=m$
$\Rightarrow m=log_2(n)$

Review Questions

1. What is recursion?
2. Why base case and recursive case is important in a Recursive thinking.
3. Give examples for various kinds of recursion.
4. Able to write a valid recursive algorithm for the given problem requirements and can analyze its performance.
5. How to perform Binary search using iterative and recursive way. Analyze its performance.