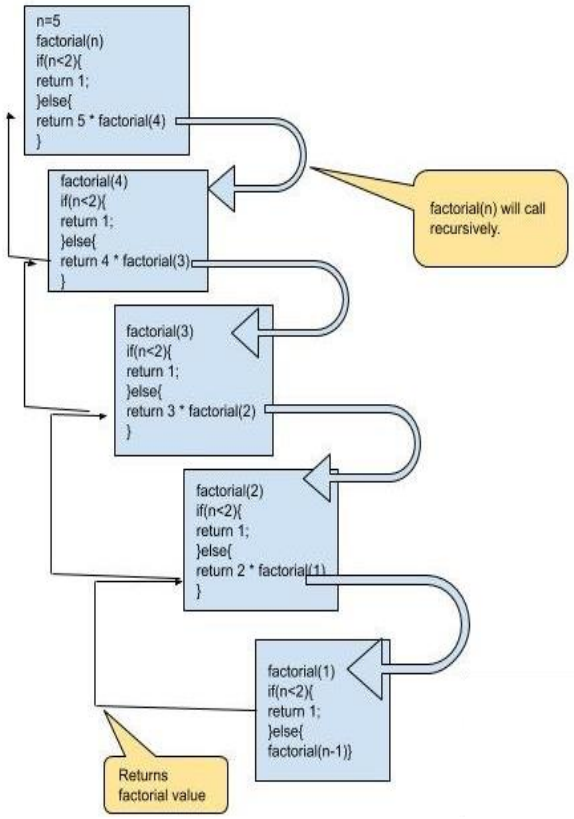
Professional Development Skills Session-4

Basic Coding--Recursion- Euclid's algorithm
Advanced Coding--Dynamic programming
Practice programs

When a function calls itself, it is called Recursion, it makes code efficient and reduces LOC.




```
class Test{
 // Our method
 static int calc(int n) {
  if (n >= 1) {
   return n*factMethod(n-1); // Recursive Calls
  } else {
   return 1; // Factorial 0 is 1
  }
 }

 public static void main(String[] args) {
  // Calling the method
  int res = calc(5);
  System.out.println("Factorial = "+res);
 }
}
```

Time Complexity: O(N)
Space Complexity : O(N)

Mathematical induction.

Recursive programming is directly related to *mathematical induction*, a technique for proving facts about natural numbers. Proving that a statement involving an integer *n* is true for infinitely many values of *n* by mathematical induction involves the following two steps:

- The *base case*: prove the statement true for some specific value or values of *n* (usually 0 or 1). For factorial(), the base case is *n* = 1.
- The *induction step*: assume that the statement to be true for all positive integers less than *n*, then use that fact to prove it true for *n*.

Euclid's algorithm.

The *greatest common divisor* (gcd) of two positive integers is the largest integer that divides evenly into both of them. For example, the gcd(102, 68) = 34.

We can efficiently compute the gcd using the following property, which holds for positive integers p and q:
If $p > q$, the gcd of p and q is the same as the gcd of q and p % q.

Problem Statement

Given two integers a and b, write a function gcd() to compute GCD. The function inputs two integers a and b and returns the GCD.
Examples:
Input: a = 5 , b = 10
Output: 5
Explanation: GCD of 5 and 10 is 5

```
public class Main {
   public static int findGcd(int n1, int n2) {
      // Initialize gcd to 1
      int gcd = 1;

      // Iterate from 1 up to minimum of n1 and n2
      for (int i = 1; i <= Math.min(n1, n2); i++) {
         // Check i is common factor of both n1 & n2
         if (n1 % i == 0 && n2 % i == 0) {
      // Update gcd to the current common factor i
            gcd = i;
         }
      }
   return gcd;
   }

   public static void main(String[] args) {
      int n1 = 20, n2 = 15;

      // Find the GCD of n1 and n2
      int gcd = findGcd(n1, n2);

      System.out.println("GCD of " + n1 + " and " +
n2 + " is: " + gcd);
   }
}
```

Time Complexity: O(min(N1,N2))
Space Complexity : O(1)

```
public class Main {
   public static int findGcd(int n1, int n2) {
      // Initialize gcd to 1
      int gcd = 1;

      // Iterate from 1 up to minimum of n1 and n2
      for (int i= Math.min(n1, n2); i>0;  i--) {
         // Check i is common factor of both n1 & n2
         if (n1 % i == 0 && n2 % i == 0) {
      // Update gcd to the current common factor i
            return  i;
         }
      }
   return 1;
   }

   public static void main(String[] args) {
      int n1 = 20, n2 = 15;

      // Find the GCD of n1 and n2
      int gcd = findGcd(n1, n2);

      System.out.println("GCD of " + n1 + " and " +
n2 + " is: " + gcd);
   }
}
```

Time Complexity: O(min(N1,N2))
Space Complexity : O(1)

```
public class Main {
   // Continue loop as long as both a and b are greater than 0
   public static int findGcd(int a, int b) {
      while(a > 0 && b > 0) {
         // If a is greater than b, subtract b from a and update a
         if(a > b) {
            // Update a to the remainder of a divided by b
            a = a % b;
         }
         // If b is greater than or equal to a, subtract a from b and update b
         else {
            // Update b to the remainder of b divided by a
            b = b % a;
         }
      }
      // Check if a becomes 0, if so, return b as the GCD
      if(a == 0) {
         return b;
      }
      // If a is not 0,return a as the GCD
      return a;
   }

   public static void main(String[] args) {
      int n1 = 20, n2 = 15;

      // Find the GCD of n1 and n2
      int gcd = findGcd(n1, n2);

      System.out.println("GCD of " + n1 + " and " + n2 + " is: " + gcd);
   }
}
```

Time Complexity: O(log$_{\phi}$(min(N,N2)))
Space Complexity : O(1)

Problem Statement

Fibonacci Number

The Fibonacci numbers, commonly denoted F(n) form a sequence, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1. That is,

F(0) = 0, F(1) = 1

F(n) = F(n - 1) + F(n - 2), for n > 1.

Given n, calculate F(n).

Example :

Input: n = 2

Output: 1

Explanation: F(2) = F(1) + F(0) = 1 + 0 = 1.

Fibonacci without Recursion

```
public class Fibonacci {
  public static void main(String args[]) {
    int n = 5;
    if (n == 0) {
      System.out.println(0);
    } else {
      int fib[] = new int[n + 1];
      fib[0] = 0;
      fib[1] = 1;
      for (int i = 2; i <= n; i++) {
        fib[i] = fib[i - 1] + fib[i - 2];
      }
      System.out.println("The Fibonacci Series up to "+n+"th term:");
      for (int i = 0; i <= n; i++) {
        System.out.print(fib[i] + " ");
      }
    }
  }
}
```

Time Complexity: O(N)+ O(N)

Space Complexity : O(N)

Fibonacci with Recursion

```
class Recursion {
    static int fibonacci(int N){
      // Base Condition.
          if(N <= 1){
              return N;
          }
          // Problem broken down into 2 functional calls and their results combined and returned.
          int last = fibonacci(N-1);
          int slast = fibonacci(N-2);

         return last + slast;
       }
```

```
   public static void main(String[] args) {

     // Here, let's take the value of N to be 4.
     int N = 4;
     System.out.println(fibonacci(N));
   }
}
```

Time Complexity: O(2^N)
Space Complexity : O(N)

Dynamic programming.

A general approach to implementing recursive programs, The basic idea of *dynamic programming* is to recursively divide a complex problem into a number of simpler subproblems; store the answer to each of these subproblems; and, ultimately, use the stored answers to solve the original problem. By solving each subproblem only once (instead of over and over), this technique avoids a potential exponential blow-up in the running time.

Top-down dynamic programming. In *top-down* dynamic programming, we store or *cache* the result of each subproblem that we solve, so that the next time we need to solve the same subproblem, we can use the cached values instead of solving the subproblem from scratch.

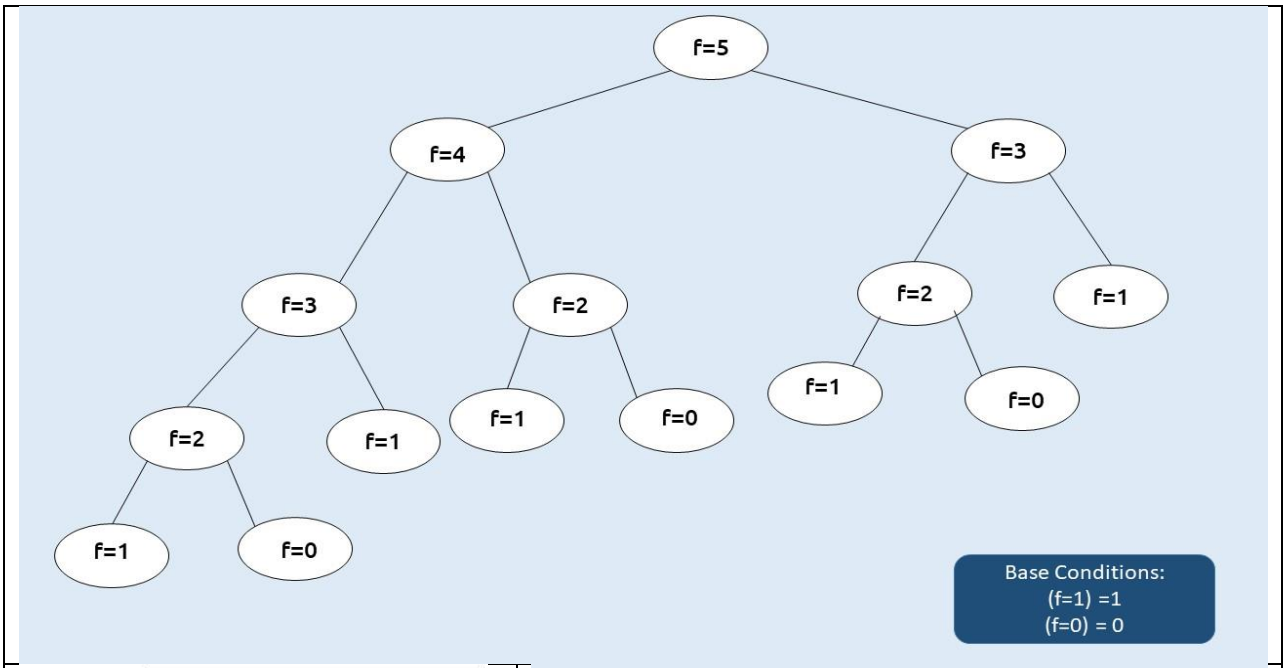
Bottom-up dynamic programming. In *bottom-up* dynamic programming, we compute solutions to all of the subproblems, starting with the "simplest" subproblems and gradually building up solutions to more and more complicated subproblems.

=> As every number is equal to the sum of the previous two terms, the recurrence relation can be written as:

```
f(n) = f(n-1) + f(n-2)
```

=>The basic pseudo-code for the problem will be given as:

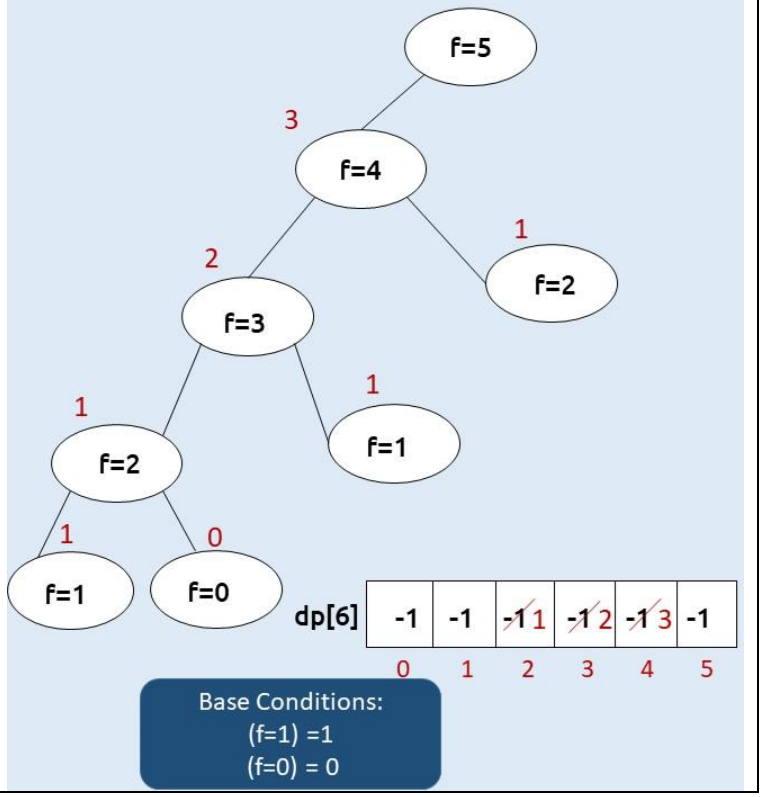
```
f(n) {
      if( n<= 1) return n

      return f(n-1) + f(n-2)

      }
```



dp[6]

-1	-1	-1	-1	-1	-1
0	1	2	3	4	5

Declare a dp[] array of size n+1 and initialize it to -1



Top-Down Dynamic Programming	Bottom-Up Dynamic Programming
class TopDownFibonacci { static int f(int n, int[] dp){ if(n<=1) return n; **if(dp[n]!= -1) return dp[n];** **return dp[n]= f(n-1,dp) + f(n-2,dp);** } public static void main(String args[]) { int n=5; int dp[]=new int[n+1]; **Arrays.fill(dp,-1);** System.out.println(f(n,dp)); } } **Time Complexity: O(N)** **Space Complexity : O(N)**	class BottomUpFibonacci { public static void main(String args[]) { int n=5; **int dp[]=new int[n+1];** **Arrays.fill(dp,-1);** dp[0]= 0; dp[1]= 1; for(int i=2; i<=n; i++){ **dp[i] = dp[i-1]+ dp[i-2];** } System.out.println(dp[n]); } } **Time Complexity: O(N)** **Space Complexity : O(N)**

Space Optimization,can be implemented with dynamic programming, recursion or non-recursion ways

```
import java.util.*;
class Demo{
public static void main(String args[]) {
 int n=5;

 int prev2 = 0;
 int prev = 1;

 for(int i=2; i<=n; i++){
    int cur_i = prev2+ prev;
    prev2 = prev;
    prev= cur_i;
 }
 System.out.println(prev);


}
}
```

Time Complexity: O(N)
Space Complexity : O(1)

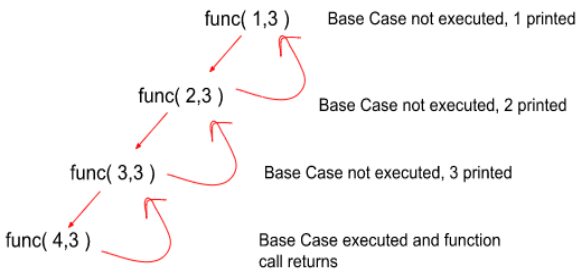
Practice Problems

Problem Statement
Print given Name n times without the loop. class Recursion {   static void func(int i, int n){   *// Base Condition.*   if(i>n) return;   System.out.println("");   *// Function call to print till i increments.*   func(i+1,n);   }   public static void main(String[] args) {   *// Here, let's take the value of n to be 4.*   int n = 4;   func(1,n);   } } **Time Complexity: O(N)** **Space Complexity : O(N)**

Problem Statement
Print numbers from **1 to n** without the help of loops. You only need to complete the function printNos() that takes n as a parameter and prints the number from 1 to n recursively. Example:Input: n = 5 Output: 1 2 3 4 5

```
class Recursion {
  static void func(int i, int n){
      // Base Condition.
      if(i>n) return;
          System.out.println(i);
      // Function call to print i till i increments to n.
      func(i+1,n);
  }
  public static void main(String[] args) {
    // Here, let's take the value of n to be 4.
    int n = 4;
    func(1,n);
  }
}
```

Time Complexity: O(N)
Space Complexity : O(N)

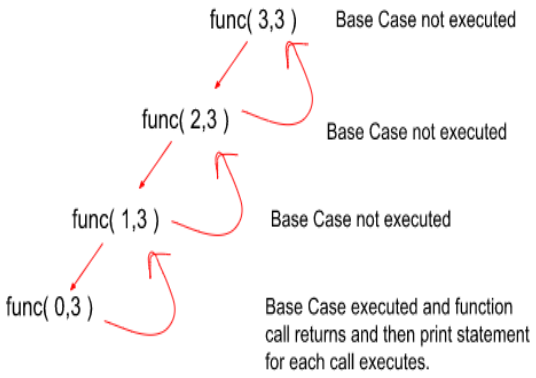


Problem Statement

Print numbers from **N to 1** (space separated) without the help of loops.
Example:
Input: n = 5
Output: 5 4 3 2 1

```
class Recursion {
   static void func(int i, int n){
         // Base Condition.
         if(i<n) return;
             System.out.println(i);
        // Function call to print i till i increments to n.
         func(i-1,n);
   }
   public static void main(String[] args) {
     // Here, let's take the value of n to be 4.
      int n = 4;
      func(n,n);
   }
}
```

Time Complexity: O(N)
Space Complexity : O(N)



Use the formula for the sum of N numbers, i.e **N(N+1)/2**

```
public class Demo {
  public static void main(String[] args) {
    solve(6);
  }
  public static void solve(int N) {
    int sum = N * (N + 1) / 2;
    System.out.println("The sum of the first " + N + " numbers is: " + sum);
  }
}
```

Time Complexity: O(1)
Space Complexity : O(1)

Problem Statement

Reverse an Array

You are given an array of integers arr[]. Your task is to reverse the given array.

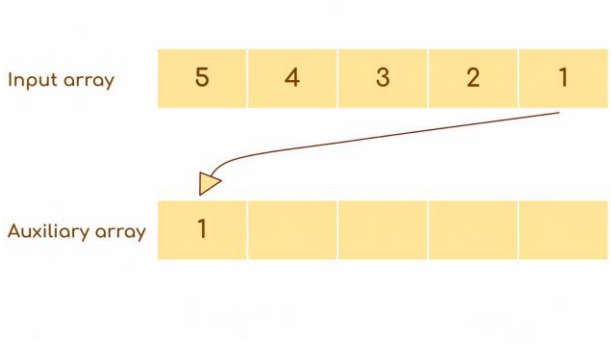
Note: Modify the array in place.

Example:

Input: arr = [1, 4, 3, 2, 6, 5]

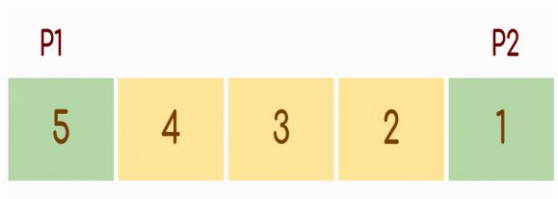
Output: [5, 6, 2, 3, 4, 1]

```
public class Main {
  //Function to print array
  static void printArray(int ans[], int n) {
    System.out.print("Reversed array is:- \n");
    for (int i = 0; i < n; i++) {
      System.out.print(ans[i] + " ");
    }
  }
  //Function to reverse array using an auxiliary
array
  static void reverseArray(int arr[], int n) {
    int[] ans = new int[n];
    for (int i = n - 1; i >= 0; i--) {
      ans[n - i - 1] = arr[i];
    }
    printArray(ans, n);
  }
  public static void main(String[] args) {
    int n = 5;
    int arr[] = { 5, 4, 3, 2, 1};
    reverseArray(arr, n);
  }
}
```



Time Complexity: O(N)
Space Complexity : O(N)

```
public class Main {
  //Function to print array
  static void printArray(int arr[], int n) {
    System.out.print("Reversed array is:- \n");
    for (int i = 0; i < n; i++) {
      System.out.print(arr[i] + " ");
    }
  }
  //Function to reverse array
  static void reverseArray(int arr[], int n) {
    int p1 = 0, p2 = n - 1;
    while (p1 < p2) {
      int tmp = arr[p1];
      arr[p1] = arr[p2];
      arr[p2] = tmp;
      p1++;
      p2--;
    }
    printArray(arr, n);
  }
  public static void main(String[] args) {
    int n = 5;
    int arr[] = { 5, 4, 3, 2, 1};
    reverseArray(arr, n);

  }
}
```



Time Complexity: O(N)
Space Complexity : O(1)

Reverse Array Using Recursion

```
public class Main {
  //Function to print array
  static void printArray(int arr[], int n) {
    System.out.print("Reversed array is:- \n");
    for (int i = 0; i < n; i++) {
      System.out.print(arr[i] + " ");
    }
  }
  //Function to reverse array using recursion
  static void reverseArray(int arr[], int start, int end) {
    if (start < end) {
      int tmp = arr[start];
      arr[start] = arr[end];
      arr[end] = tmp;
      reverseArray(arr, start + 1, end - 1);
    }
  }
  public static void main(String[] args) {
    int n = 5;
    int arr[] = { 5, 4, 3, 2, 1};
    reverseArray(arr, 0, n - 1);
    printArray(arr, n);
  }
}
```

Time Complexity: O(N)
Space Complexity : O(1)

Problem Statement

Valid Palindrome

A phrase is a palindrome if, after converting all uppercase letters into lowercase letters and removing all non-alphanumeric characters, it reads the same forward and backward. Alphanumeric characters include letters and numbers.

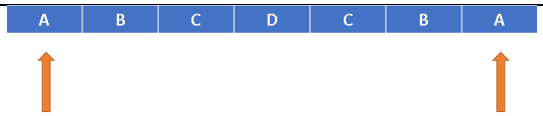
Given a string s, return true if it is a palindrome, or false otherwise.

Example:

Input: s = "madam"

Output: true

```
import java.util.Arrays;
 static private boolean isPalindrome(String s) {
  int left = 0, right = s.length()-1;
     while(left<right)
     {
        char l = s.charAt(left), r = s.charAt(right);
        if(!Character.isLetterOrDigit(l))
           left++;
        else if(!Character.isLetterOrDigit(r))
           right--;
        else
if(Character.toLowerCase(l)!=Character.toLowerCase(r))
           return false;
        else {
           left++;
           right--;
        }
     }
     return true;
 }
```



Time Complexity: O(N)

Space Complexity : O(1)

```
class Recursion {
static boolean palindrome(int i, String s){
        // Base Condition
        // If i exceeds half of the string, means all the elements are compared, we return true.
        if(i>=s.length()/2) return true;
        // If start is not equal to end, not palindrome.
        if(s.charAt(i)!=s.charAt(s.length()-i-1)) return false;
        // If both characters are same, increment i and check start+1 and end-1.
        return palindrome(i+1,s);
  }
public static void main(String[] args) {
     // Example string.
     String s = "madam";
     System.out.println(palindrome(0,s));
   }
}
```

Time Complexity: O(N/2)

Space Complexity : O(1)