PW SKILLS

Lesson Plan

Java
Recursion 2

PW SKILLS

Q1. Write a function to calculate the nth Fibonacci number using recursion.

(In mathematical terms, the sequence Fn of Fibonacci numbers is defined by the recurrence relation: f(n)=f(n-1)+f(n=2) with seed values and f(0)=1 and f(1))

Example1: Input : n = 1 Output: 1
Explanation: 1 is the 1st number of the Fibonacci series.

Example2: Input : n = 9 Output : 34
Explanation: 34 is the 9th number of Fibonacci series.

Java Solution Code:

```
public class Fibonacci {
    public static int fibonacci(int n) {
        if (n ⩽ 1) {
            return n;
        }
     return fibonacci(n - 1) + fibonacci(n - 2);
    }
    public static void main(String[] args) {
        int n = 6; // Change this to calculate a different
Fibonacci number
        System.out.println("Fibonacci number at position " +
n + " is: " + fibonacci(n));
    }
}
```

Explanation:

1. Base Case: If the input `n` is 0 or 1, return `n` itself as the Fibonacci sequence starts with 0 and 1.

2. Recursive Calls: For `n` greater than 1, the function `Fibonacci (n)` calls itself twice with `Fibonacci (n - 1)` and `Fibonacci (n - 2)` to calculate the Fibonacci number at position `n`.

3. Combine Results: The function keeps making these recursive calls, adding the results of `Fibonacci (n - 1)` and `Fibonacci (n - 2)` to find the nth Fibonacci number.

Time Complexity: O(2^n) as every function calls two other functions.

Space Complexity: O(n) as the maximum depth of the recursion tree is n.

Q2. Write a program to calculate pow(x, n)

Input : x = 2, n = 3
Output: 8 (since 2^3=8)

Input : x = 7, n = 2
Output: 49 (since 7^2=49)

Java solution code:

```
static int power(int x, int y)
{
    int temp;
    if (y == 0)
        return 1;
    temp = power(x, y / 2);
    if (y % 2 == 0)
        return temp * temp;
    else
        return x * temp * temp;
}
```

Explanation:

1. Base Case: If the exponent `y` is 0, the function returns 1 as any number raised to the power of 0 is 1.

2. Dividing the Exponent: For `y` greater than 0, the function divides the exponent by 2 (`y / 2`), and then recursively calls itself with the base number `x` and the halved exponent (`power(x, y / 2)`).

3. Optimization for Even Exponents: The function stores the result of the recursive call in `temp`. If the exponent `y` is even (`y % 2 == 0`), it returns the square of `temp` (`temp * temp`). This is an optimization as it avoids repetitive multiplications by squaring the result.

4. Handling Odd Exponents: If the exponent `y` is odd, it returns `x * temp * temp`.It multiplies the base number `x` once with the result of the recursive call `temp`, and then squares `temp`.

Time Complexity: O(log n) as at each step n is being halved

Space Complexity: O(log n) for recursive call stack.

Q3. There are n stairs, a person standing at the bottom wants to climb stairs to reach the nth stair. The person can climb either 1 or 2 stairs at a time, the task is to count the number of ways that a person can reach the top.

Input : n = 1
Output: 1 => only one way to climb 1 stair

Input : n = 2
Output: 2 => (1, 1) and (2)

Input: n = 4
Output: 5 => (1, 1, 1, 1), (1, 1, 2), (2, 1, 1), (1, 2, 1), (2, 2)

Java solution code:

```
public class ClimbingStairs {
    public static int climbStairs(int n) {
        if (n ⩽ 1) {
            return 1;
        }
        return climbStairs(n - 1) + climbStairs(n - 2);
    }

    public static void main(String[] args) {
        int stairs = 4; // Example input for number of
stairs
        int ways = climbStairs(stairs);
        System.out.println("Number of ways to climb the
stairs = " + ways);
    }
}
```

Code Explanation:

1. Base Case: If `n` is 0 or 1, there's only one way to climb (return 1), for zero only way is to not climb.

2. Recursive Steps: For n > 1, we can either climb 1 or 2 at each step so, count ways by adding the results of climbing (n - 1) and (n - 2) stairs.

3. Repeat: Keep adding ways until you reach the base case.

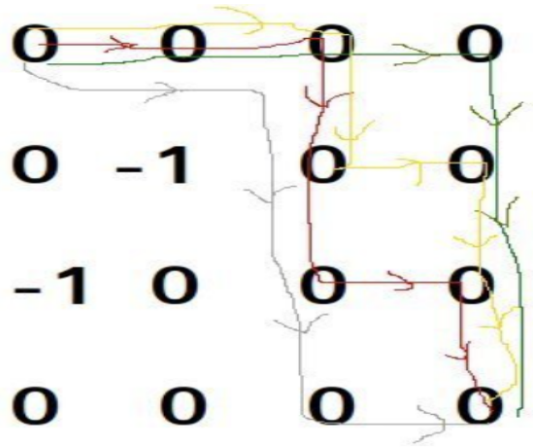
4. Final Result: Return the total ways to climb `n` stairs by considering all step combinations.

Time Complexity: O(2^n) because at each of n stairs, there are two choices

Space Complexity: O(n), because of recursive stack space.

Q4. Given a maze with obstacles, count the number of paths to reach the rightmost-bottommost cell from the topmost-leftmost cell. A cell in the given maze has a value of -1 if it is a blockage or dead-end, else 0. From a given cell, we are allowed to move to cells (i+1, j) and (i, j+1) only.

Input: maze[R][C] = { {0, 0, 0, 0},
{0, -1, 0, 0},
{-1, 0, 0, 0},
{0, 0, 0, 0} };
Output: 4
There are four possible paths as shown in below diagram




```
public class ClimbingStairs {
import java.util.*;

public class UniquePathsObstacles {
 static long MOD = 1000000007;
 static long helper(int i, int j, int[][] obstacleGrid) {
  // Base cases
int n = obstacleGrid.length;
  int m = obstacleGrid[0].length;


if (i ⩾ n || j⩾m) {
   return 0;
  }
  if (obstacleGrid[m][n] == -1) {
   return 0;
  }
  if (i==n-1 && j==m-1) {
   return 1;
  }

  // Recursively calculate the number of unique paths by
  //Considering downward and rightward movements
  return (helper(i+1, j, obstacleGrid)+ helper(i, j+1,
obstacleGrid));
 }

 // Function to calculate the number of unique paths with
obstacles
 static long uniquePathsWithObstacles(int[][] obstacleGrid)
{
  int m = obstacleGrid.length;
  int n = obstacleGrid[0].length;

  return helper(0, 0, obstacleGrid);
 }
```

```
 public static void main(String[] args) {
  // Example obstacle grid
  int[][] grid =      {                              { 0, 0,
0, 0 },
      { 0, -1, 0, 0 },
      { -1, 0, 0, 0 },
      { 0, 0, 0, 0 } };

  // Calculate and print the number of unique paths with
obstacles
  System.out.println(uniquePathsWithObstacles(grid));
 }
}
```

Code Explanation:

1. Base Case: If we are already at target (n-1,m-1) return 1, if we face an obstacle or go out of the matrix there's no path so return 0;

2. Recursive Steps: at each step, we can either go down or right so go both side and their paths

3. Repeat: Keep adding ways until you reach the base case.

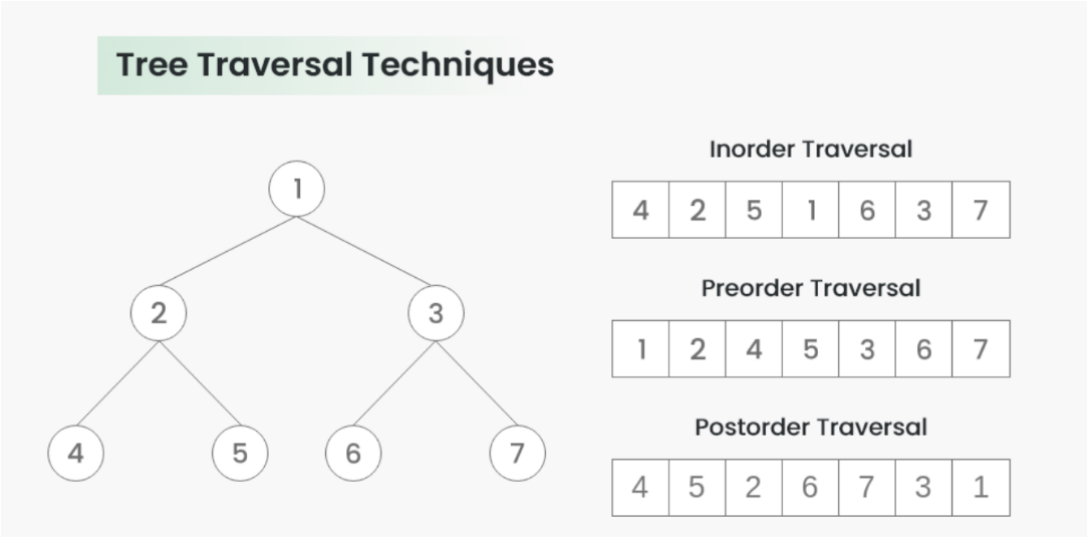
Thus we'll have our answer

Time Complexity: O(2^n*m) because at each of box (n*m boxes) in matrix we have 2 choices

Space Complexity: O(n*m), because of recursive stack space.

Q5. Traversal of Tree (inorder, preorder, postorder)

Sol:



#java code

Inorder:

```
void printInorder(Node node)
    {
 //left⇒curr⇒right
          if (node == null)  return;
           printInorder(node.left);
           System.out.print(node.key + " ");
           printInorder(node.right);
    }
```

Preorder:

```
void printPreorder(Node node)
    {
 //curr⇒left⇒right
       if (node == null)  return;
       System.out.print(node.key + " ");
        printPreorder(node.left);
        printPreorder(node.right);
    }
```

Preorder:

```
void printPostorder(Node node)
    {
 //left⇒right⇒curr
       if (node == null) return;
        printPostorder(node.left);
        printPostorder(node.right);
         System.out.print(node.key + " ");
    }
```

//Explanation: All three three codes are pretty similar in inorder we print left->curr->right,

//so we just wrote the same thing in the recursion function and similarly in preorder it goes curr->left->right while post order goes like left->right->curr

In all the cases base case will be the same i.e if we reach a NULL node we'll simply return

Time Complexity: O(n) each node of the tree is visited once

Space Complexity: O(height of tree), because of recursive stack space.

Q6. Print ZigZag, given a number n Figure out the pattern from sample inputs and write a recursive function to achieve the above for any positive number n.

Input1 -> 1
Output1 -> 1 1 1

Input2 -> 2
Output2 -> 2 1 1 1 2 1 1 1 2

Input2 -> 3
Output3 -> 3 2 1 1 1 2 1 1 1 2 3 2 1 1 1 2 1 1 1 2 3

Java solution code:

```
public static void pzz(int n) {
  if(n == 0){
    return;
  }

  System.out.print(n + " ");
  pzz(n - 1);
  System.out.print(n + " ");
  pzz(n - 1);
  System.out.print(n + " ");
}
```

Code Explanation:

From sample input we observe from nth we write the n 3 times then we insert the pattern of n-1 in between for e.g for 1 it was 1 1 1 now for 2 we write 2 2 2 then insert the pattern of 1 in all the two in between gap thus it becomes 1 2 2 2 1 2 2 2 1 same thing you can observe for 3 as well

//thus simple recursion => print(n) -> printpattern(n-1) -> print(n) -> printpattern(n-1) -> print(n)

//base case will printpattern(0) will be nothing so just return as we are observing we not inserting anything between 1's printing pattern for 1

Time Complexity: O(2^n) because each time it is making two calls

Space Complexity: O(n), because of recursive stack space.

Q7. Tower of Hanoi

The tower of Hanoi is a famous puzzle where we have three rods and N disks. The objective of the puzzle is to move the entire stack to another rod. You are given the number of discs N. Initially, these discs are in rod 1. You need to print all the steps of disc movement so that all the discs reach the 3rd rod. Also, you need to find the total moves.

Note: The discs are arranged such that the top disc is numbered 1 and the bottom-most disc is numbered N. Also, all the discs have different sizes and a bigger disc cannot be put on the top of a smaller disc..

Input: 2
Output: 3

Disk 1 moved from A to B

Disk 2 moved from A to C

Disk 1 moved from B to C

Explanation: For N=2, steps will be as follows in the example and a total of 3 steps will be taken.

Input: 3
Output: 7

Disk 1 moved from A to C

Disk 2 moved from A to B

Disk 1 moved from C to B

Disk 3 moved from A to C

Disk 1 moved from B to A

Disk 2 moved from B to C

Disk 1 moved from A to C

Explanation: For N=3 , steps will be
as follows in the example and a total of 7 steps will be taken.

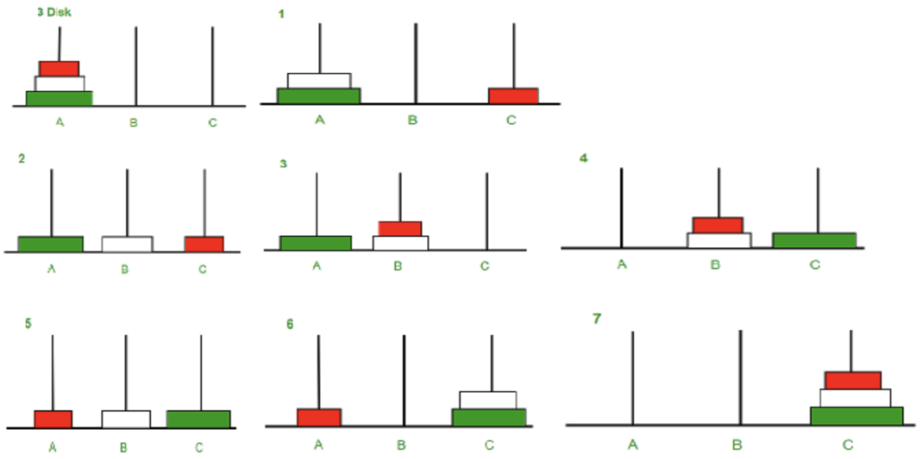


Image illustration for 3 disks

Java solution code:

```
class Hanoi {

    public long toh(int N, int from, int to, int aux) {
        long moves = 0L;
        if (N ⩾ 1) {
            // recursive call to move top disk from "from"
to aux in the current call
            moves += toh(N - 1, from, aux, to);
            System.out.println("move disk " + N + " from rod
" + from +
                               " to rod " + to);

            // increment moves
            moves++;

            // recursive call to move top disk from aux to
"to" in the current call
            moves += toh(N - 1, aux, to, from);
        }
        return moves;
    }
}
```

Code Explanation:

The idea is to use the helper node to reach the destination using recursion. Below is the pattern for this problem:

1. Shift ‘N-1’ disks from ‘A’ to ‘B’, using C.
2. Shift the last disk from ‘A’ to ‘C’.
3. Shift ‘N-1’ disks from ‘B’ to ‘C’, using A.

Steps 1 and 3 are actually the same as the original problem. Thus we have derived a recursive solution for the given problem.

//base case N=1 then simply move the disk from the source rod to the destination rod.

1. Create a function toh where pass the N (current number of disk), srcRod, destRod, helperRod.
2. Now Making function call for N – 1 disk. (move them from src to helperRod)
3. Then we are printing the current disk along with srcRod and destRod
4. Again making a fucntion call for N – 1 disk (move them from helperRod to srcROD).

Time complexity: O(2^N), As each function call calls itself two times by decreasing N just by 1.

Auxiliary Space: O(N), As at an instant we will have at most N function calls in the stack.

Q8. Array Traversal using recursion

Java solution code:

```
     void traverseArray(int[] arr, int idx) {
        // Base case: Exit when index goes beyond array
length
        if (idx ⩾ arr.length) {
            return;
        }
     // Print current element
        System.out.print(arr[idx] + " ");
        // Recursive call for the next index
        traverseArray(arr, idx + 1);
    }
```

Code Explanation:

We are passing the array and current index as input and for each index

We print the curr element the call function for the next index and here the base will be when the index goes out of bound i.e index>=arr.length

Time Complexity: O(n) , visiting each element of array once

Space Complexity: O(n), Stack space grows linearly with array size due to recursion.

Q9. Max element of Array using recursion

Input: arr[] = {10, 20, 4}
Output: 20

Explanation: Among 10 20 and 4, 20 is the largest.

Input: arr[] = {20, 10, 20, 4, 100}
Output: 100 //cleary 100 is largest

Java solution code:

```
static int largest(int arr[], int n, int i)
    {
        // Last index returns the element
        if (i == n - 1) {
            return arr[i];
        }
         // Find the maximum from the rest of the array
        int recMax = largest(arr, n, i + 1);

        // Compare with i-th element and return
        return Math.max(recMax, arr[i]);
    }
```

Code Explanation:

- Set an integer i = 0 to denote the current index being searched.
- Check if i is the last index, return arr[i].
- Increment i and call the recursive function for the new value of i.
- Compare the maximum value returned from the recursion function with arr[i].
- Return the **max between these two from the current recursion call.**

Time complexity: O(N), as for each index we are calling function only one (i.e for the next index), there will total of n such calls 1 for each index

Auxiliary Space: O(N), since this will size the recursion stack, as function calls get piled up

Q10. Remove all occurrences of a character in a string

Input : s = "banana"
c = 'a'
Output : s = "bnn" ,removed all 'a'

Java solution code:

```
static String removeChar(String str, char ch)
    {
        // Base Case
        if (str.length() == 0) {
            return "";
        }
         // Check the first character of the given string if
its ch then remove it
        if (str.charAt(0) == ch) {
            // Pass the rest of the string
            // to recursion Function call
            return removeChar(str.substring(1), ch);
        }


        // Add the first character since if its not ch if we
reached here
        return str.charAt(0)  + removeChar(str.substring(1),
ch);
    }
```

Code Explanation:

1. Here our recursive function's base case will be when string length becomes 0.
2. And in the recursive function, if the encountered character is the character that we have to remove then call the recursive function from the next character.
3. else store this character in answer and then call the recursive function from the next character.

Time Complexity: O(n), as for each index we are making the function call once.

Auxiliary Space: O(n), if we consider recursion stack space

Q11. Given an integer array nums of unique elements, return all possible subsets(the power set).(Leetcode 78)

Input: nums = [1,2,3]
Output: [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]

Java solution code:

```
class Solution {
    public List<List<Integer>> subsets(int[] nums) {
        return recSubsets(0, nums, new ArrayList<>(), new
ArrayList<>());
    }
```

```
    List<List<Integer>> recSubsets(int i, int[] nums,
List<Integer> prevSubSet, List<List<Integer>> res){
        if(i==nums.length){
            res.add(prevSubSet);
            return res;
        }

        List<Integer> extraSubSet = new
ArrayList<>(prevSubSet);
        extraSubSet.add(nums[i]);
         recSubsets(i+1, nums, prevSubSet, res);
        recSubsets(i+1, nums, extraSubSet, res);
         return res;
    }
}
```

Code Explanation:

We are solving the problem using resSubsets() function, which takes the following inputs, i curr index of elements in nums, the nums array of elements, prevSubset which has subset form up to this index, and then last we have the res array which will store the final result

Base Case: so in rec function call base case if if each completes the array i.e i==n then we'll put push our subset formed (i.e prevSubset) to the res

Now for each index, we have two choices either we include the curr char or we exclude it thus we form a new extraSubset array in which we include the curr element of array

Now we make two function calls for the next element one including curr element (i.e with prevSubset) and one excluding curr element (i.e with extra subset)

At last we return this res array which will have all the subset

Time Complexity: O(2^n), for each char in the string we make 2 calls

Space Complexity: O(n), due to recursive stack

Q12. Print subsets of a string containing duplicate characters.

Input : aaa
Output : a, a, a, aa, aa, aa, aaa

Java solution code:

```
// Java program for the above approach
class solution {
 // Declare a global list
 static List<String> al = new ArrayList<>();
 // Creating a public static llist such that we can directly
store generated substring in this global array
```

```
 public static void main(String[] args)
 {
  String s = "abcd";
  findsubsequences(s, ""); // Calling a function
  System.out.println(al);
 }

 private static void findsubsequences(String s,String ans)
 {
  if (s.length() == 0) {
   al.add(ans);
   return;
  }

  // adding 1st character in string
  findsubsequences(s.substring(1), ans + s.charAt(0));

  // Not adding first character of the string
  // because the concept of subsequence either
  // character will present or not
  findsubsequences(s.substring(1), ans);
 }
}
```

Code Explanation:

In each recursive function call, we are making two calls when moving to the next one selecting the curr char and one not selectin curr char

1. The base case when the string has nothing left we can thus add generated string to array and return.
2. For other case make two calls first one while adding curr character to string and the other one without adding it.
3. Thus on running this function all the generated substrings will be pushed to global array of string i.e al

Time Complexity: O(2^n),for each char in string we make 2 calls

Space Complexity: O(n), due to recursive stack

Q13. Print all increasing sequences of length k from first n natural numbers

Input: k = 2, n = 3
Output: 1 2
1 3
2 3

Input: k = 5, n = 5
Output: 1 2 3 4 5

Java solution code:

```
class Solution {
 static void printArr(int[] arr, int k)
 {
  for (int i = 0; i < k; i++)
   System.out.print(arr[i] + " ");
  System.out.print("\n");
 }
 // A recursive function to print
 static void printSeqUtil(int n, int k, int len, int[] arr)
 {
  if (len == k)
  {
   printArr(arr, k);
   return;
  }
  int i = (len == 0) ? 1 : arr[len - 1] + 1;
  len++;
  while (i <= n)
  {
arr[len - 1] = i;
   printSeqUtil(n, k, len, arr);
   i++;
  }
  len--;
 }
 static void printSeq(int n, int k)
 {
  // An array to store
  // individual sequences
  int[] arr = new int[k];
  // Initial length of
  // current sequence
  int len = 0;
  printSeqUtil(n, k, len, arr);
 }

 // Driver Code
 static public void main (String[] args)
 {
  int k = 3, n = 7;
  printSeq(n, k);
 }
}
```

Code Explanation:

The idea is to create an array of lengths k. The array stores the current sequence. For every position in the array, we check the previous element and one by one put all elements greater than the previous element. If there is no previous element (first position), we put all numbers from 1 to n.

1. Base Case: if len ==k If the length of the current increasing sequence becomes k, print it.
2. Decide the starting number to put at the current position if len is 0 then it's 1 otherwise it is prev element +1, as we are adding a new element increase the length.
3. In the while loop Put all numbers (which are greater than the previous element) at the new position.
4. Now undo the increase made to len to subtract 1 from it since len is shared among all functions all.

Thus all increasing subsequence of length k will be printed

Time Complexity: O(n*n!), as there are n! permutations and it requires O(n) time to print a permutation. T(n) = n * T(n-1) + O(1) when reduced will result in O(n*n!) time.

Auxiliary Space: O(n), as at any time there will at most n function calls in the stack

Q14. Finding all permutations of a string given all elements of the string are unique

Input: S = "ABC"
Output: "ABC", "ACB", "BAC", "BCA", "CBA", "CAB"

Input: S = "XY"
Output: "XY", "YX"

Java solution code:

```
public class Permutation {
 // Function call
 public static void main(String[] args)
 {
  String str = "ABC";
  int n = str.length();
  Permutation permutation = new Permutation();
  permutation.permute(str, 0, n - 1);
 }

 private void permute(String str, int l, int r)
 {
  if (l == r)
   System.out.println(str);
  else {
   for (int i = l; i <= r; i++) {
    str = swap(str, l, i);
    permute(str, l + 1, r);
    str = swap(str, l, i);
   }
  }
 }
```

```
public String swap(String a, int i, int j)
 {
  char temp;
  char[] charArray = a.toCharArray();
  temp = charArray[i];
  charArray[i] = charArray[j];
  charArray[j] = temp;
  return String.valueOf(charArray);
 }
}
```

Code Explanation:

- Create a function permute() with parameters as input string, starting index of the string, ending index of the string
- Call this function with values input string, 0, size of string – 1
- In this function, if the value of L and R is the same then print the same string
- Else run a for loop from L to R and swap the current element in the for loop with the inputString[L]
- Then again call this same function by increasing the value of L by 1
- After that again swap the previously swapped values to initiate backtracking

Time Complexity: O(N * N!) Note that there are N! permutations and it requires O(N) time to print a permutation.

Auxiliary Space: O(N) As `l` progresses from 1 to `n-1`, the function generates up to `n` recursive calls until it reaches the base condition `l == r`. So at any instant, there can be utmost N calls in the call stack.