PW SKILLS

Lesson Plan

Recursion

PW SKILLS

Today's Checklist:

1. What is Recursion
2. Why Recursion
3. Properties of Recursion
4. Algorithmic Steps for Recursion
5. The call stack
6. Application of Recursion to solve various problems

What is Recursion?

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function. examples of such problems are the Fibonacci Series, power function, Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc.
A recursive function solves a problem by breaking it down into smaller parts and calling itself to handle these smaller pieces. To avoid endless repetition, it's crucial to set a stopping point (often called base case) where the function stops calling itself. This way, each recursive call deals with a simpler version of the initial problem.

Why Recursion:

Recursion is a cool method that helps make code shorter and simpler to understand and write. It's better than iteration as it makes the code mode compact and elegant and it mirrors the problem structure, enhancing readability for recursive-natured problems. Whenever a task can be broken down into smaller versions of itself, recursion is a great way to handle it. Like when we calculate the factorial of a number.

Properties of Recursion:

Performing the same operations multiple times with different inputs.
Breaking the problem into smaller parts with every try.
A base condition is needed to stop the recursion otherwise, an infinite loop will occur.

Algorithmic steps:

Step 1. Establish a Base Case: Find the simplest scenario where you know the solution or where the problem becomes straightforward. This prevents endless loops by setting a point for the recursion to stop.

Step 2. Break Down the Problem: Express the problem in smaller, solvable parts. Use these parts to call the function again, dealing with each smaller problem.

Step 3. Ensure Stopping: Guarantee that the function eventually gets to the base case, preventing infinite loops.

Step 4. Merge Solutions: Combine the answers from the smaller parts to solve the main problem more effectively. This integration enhances the final result.
How are recursive functions stored in memory?
Recursion uses more memory, because the recursive function adds to the stack with each recursive call, and keeps the values there until the call is finished. The recursive function uses LIFO (LAST IN FIRST OUT) Structure just like the stack data structure.

What is the base condition in recursion?

In the recursive program, the solution to the base case is provided and the solution to the bigger problem is expressed in terms of smaller problems.
The Call Stack
Recursive functions use something called “the call stack.” When a program calls a function, that function goes on top of the call stack. This is similar to a stack of books. You add things one at a time. Then, when you are ready to take something off, you always take off the top item.

What is the difference between direct and indirect recursion?

Direct Recursion: A function fun is called direct recursive if it calls itself i.e. fun.

Indirect Recursion: A function is called indirect recursive if it calls another function, then that function calls this function. For better understanding consider two functions fun1 and fun2, when fun1 is executed it makes calls to fun2, and when fun2 is executed it makes calls to fun1. So here when we call fun1 it will not directly call itself so it’s not a direct recursion but it calls fun2 which eventually calls fun1, so indirect recursion is still present

Function Calls

In C++, functions can call other functions to perform specific tasks. When a function is invoked within another function, it creates a chain of function calls. This process is crucial for breaking down complex problems into smaller, manageable tasks, allowing for better organization and readability in your code.

Function Calling Itself (Recursion)

Recursion is a powerful concept where a function calls itself during its execution. In C++, a recursive function breaks down a problem into smaller, similar subproblems until it reaches a base case that directly resolves without further recursion. This technique offers an elegant way to solve complex problems by reducing them into simpler instances of the same problem. However, it requires careful handling of base cases to prevent infinite loops and stack overflow errors.

Q. Make a function which calculates the factorial of n using recursion.

Code:

```
 int factorialint n) {
    if (n ⩽ 1) {
        return 1; // Base case: factorial of 0 and 1 is 1
    } else {
        return n * factorial(n - 1); // Recursive call to
calculate factorial
    }
}
```

Time: O(n), n function calls
Space: O(n), recursion stack

Q. Print n to 1

Code:

```
void print(int n){
  if(n==0) return;
  cout<<n<<" ";
  print(n-1);
}
```

Time: O(n), n function calls
Space: O(n), recursion stack

Q. Print 1 to n (extra parameter)

```
void printNumbers(int current, int n) {
    if (current > n) {
        return;
    } else {
        std::cout << current << " ";
        printNumbers(current + 1, n);
    }
}
```

Explanation: recursively prints numbers from current to n. The base case is when current becomes greater than n.

Time: O(n), n function calls
Space: O(n), recursion stack

Q. Print 1 to n (after recursive call)

Code:

```
void printNumbers(int n) {
    if (n < 1) {
        return;
    } else {
        printNumbers(n - 1);
        std::cout << n << " ";
    }
}
```

Explanation: It starts with the input n, and decrements n in each recursive call until it reaches the base case when n is less than 1. After the recursive call, it prints the current number n.

Time: O(n), n function calls
Space: O(n), recursion stack

Q. Print sum from 1 to n (Parameterised)

Code:

```
int calculateSum(int n, int sum) {
    if (n == 0) {
        return sum; // Base case: Return the accumulated sum when
n reaches 0
    } else {
        return calculateSum(n - 1, sum + n); // Recursive call to
accumulate the sum
    }
}
```

Time: O(n), n function calls
Work.Space: O(n), recursion stack

Q. Print sum from 1 to n (Return type)

Code:

```
int calculateSum(int n) {
    if (n == 0) {
        return 0; // Base case: Sum from 1 to 0 is 0
    } else {
        return n + calculateSum(n - 1); // Recursive call to
calculate sum
    }
}
```

Time: O(n), n function calls, where each function call does a constant amount of work.
Space: O(n), recursion stack

Q. Make a function that calculates 'a' raised to the power 'b' using recursion.

Code:

```
double power(double a, int b) {
    if (b == 0) {
        return 1; // Any number raised to the power 0 is 1
    } else if (b > 0) {
        return a * power(a, b - 1); // Recursive call to
calculate power
    } else {
        return 1 / power(a, -b); // For negative exponents,
return reciprocal of the positive power
    }
}
```

Explanation: This code defines a function power() that calculates a raised to the power b using recursion. It has a base case for b = 0, where any number raised to the power 0 is 1. For positive exponents, it recursively multiplies a by itself b times. For negative exponents, it returns the reciprocal of the positive power

Time Complexity: O(b) - The time complexity is proportional to the exponent b as the function is called recursively b times.

Space Complexity: O(b) - The space complexity is also proportional to the exponent b due to the recursive calls, potentially leading to stack overflow for large b.

Time Complexity in Recursion:

Recursion involves breaking down a problem into smaller, similar subproblems until reaching a base case.
Analyzing time complexity in recursive algorithms involves understanding the number of recursive calls made concerning the input size.
Each recursive call contributes to the overall time complexity, and understanding the number of calls made and their operations is crucial.
Express time complexity in terms of the number of recursive calls and the work done within each call.

Space Complexity in Recursion:

Recursion utilizes the call stack to store information about each recursive call.
Space complexity in recursion involves analyzing the memory space used by the call stack concerning the depth of recursion.
The space complexity increases with each recursive call and the amount of space required for each call's variables and data.
Understanding the maximum depth of recursion and the auxiliary space used per call is vital in evaluating space complexity.

Analyzing Recursive Algorithms:

To assess time complexity, count the number of recursive calls made and the operations performed in each call.
Identify the dominant factor affecting the number of recursive calls and express time complexity using Big O notation.
For space complexity, evaluate the maximum recursion depth and the auxiliary space required per call (variables, data structures).
Express space complexity using Big O notation based on the maximum space used by the recursion stack.

Considerations:
Recursion offers an elegant way to solve problems but may lead to higher space complexity due to the call stack.
Carefully handling recursive calls and understanding their impact on time and space complexity is essential for efficient recursive algorithms.

Q. Write a function to calculate the nth fibonacci number using recursion.

Code:

```
 int fibonacci(int n) {
    if (n ⩽ 1) {
        return n; // Base case: Fibonacci of 0 is 0, and
Fibonacci of 1 is 1
    } else {
        return fibonacci(n - 1) + fibonacci(n - 2); // Recursive
call to calculate Fibonacci
    }
}
```

Explanation: This function calculates the nth Fibonacci number using recursion. It has a base case for n <= 1, where Fibonacci of 0 is 0, and Fibonacci of 1 is 1. For n > 1, it recursively calculates the nth Fibonacci number by summing the (n-1)th and (n-2)th Fibonacci numbers.

Time Complexity: O(2^n) - The time complexity of this recursive approach is exponential as it involves redundant calculations. For each call, two more recursive calls are made.

Space Complexity: O(n) - The space complexity is O(n) due to the recursive calls occupying space on the call stack, potentially leading to stack overflow for large n.

Q. Power function (logarithmic)

Code:

```
long long int power(int base, int exponent) {
    if (exponent == 0) {
        return 1; // Any number raised to power 0 is 1
    } else if (exponent % 2 == 0) {
        long long int result = power(base, exponent / 2);
        return result * result; // For even exponents, square the
result
    } else {
        long long int result = power(base, (exponent - 1) / 2);
        return base * result * result; // For odd exponents,
multiply base once
    }
}
```

Explanation: It calculates the power of a number using recursion and the exponentiation by squaring technique. It handles both even and odd exponents efficiently by halving the exponent at each recursive step.

Time Complexity: O(log n) - The time complexity is logarithmic because the exponent is divided by 2 in each recursive step.

Space Complexity: O(log n) - The space complexity is also logarithmic due to the recursive calls, potentially leading to stack overflow for large n.

Q. Stair Path (Leetcode 70)
You are climbing a staircase. It takes n steps to reach the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Example 1:

Input: n = 2
Output: 2

Explanation: There are two ways to climb to the top.

1. 1 step + 1 step
2. 2 steps

Example 2:

Input: n = 3
Output: 3

Explanation: There are three ways to climb to the top.

1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step

Code:

```
int climbStairs(int n) {
        if(n<=1) return 1;
        return climbStairs(n-1)+climbStairs(n-2);
    }
```

Explanation: It uses a recursive approach by breaking down the problem into subproblems: the number of ways to reach n steps is the sum of ways to reach n-1 steps and n-2 steps.

Time Complexity: O(2^n) because at each of n stairs, there are two choices
Space Complexity: O(n), because of recursive stack space.

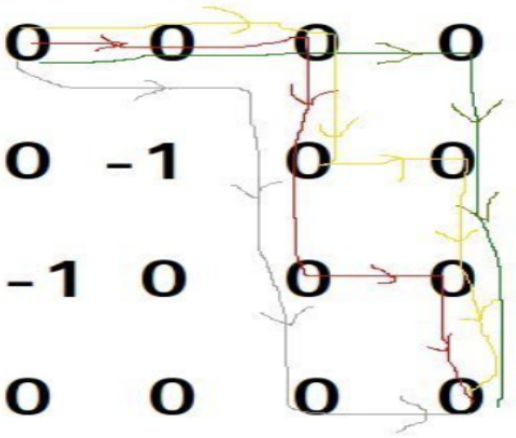
Q. Maze path

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



Code:

```
#include <bits/stdc++.h>
using namespace std;

long long int MOD = 1e9 + 7;

long long int helper(long long int m, long long int n,
vector<vector<long long int>>& obstacleGrid) {
    if (m < 0 || n < 0) return 0;
    if (obstacleGrid[m][n] == -1) return 0;
    if (m == 0 && n == 0) return 1;

    return (helper(m - 1, n, obstacleGrid) % MOD + helper(m, n -
1, obstacleGrid) % MOD) % MOD;
}

long long int uniquePathsWithObstacles(vector<vector<long long
int>>& obstacleGrid) {
    long long int m = obstacleGrid.size();
    long long int n = obstacleGrid[0].size();
    return helper(m - 1, n - 1, obstacleGrid);
}

int main() {
    vector<vector<long long int>> grid = { { 0, 0, 0, 0 },
                                           { 0, -1, 0, 0 },
                                           { -1, 0, 0, 0 },
                                           { 0, 0, 0, 0 } };
    cout << uniquePathsWithObstacles(grid);
    return 0;
}
```

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.

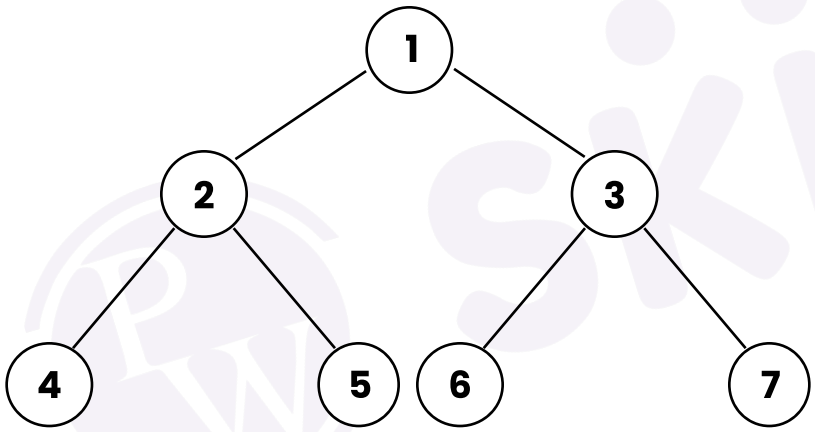
Thus we'll have our answer

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.

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

Solution:

Tree Traversal Techniques



Inorder Traversal

4	2	5	1	6	3	7

Preorder Traversal

1	2	4	5	3	6	7

Postorder Traversal

4	5	2	6	7	3	1

Code:

```
//Inorder Traversal
void printInorder(Node* node) {
    if (node == nullptr) return;
    printInorder(node->left);
    cout << node->key << " ";
    printInorder(node->right);
}

// Preorder traversal
void printPreorder(Node* node) {
    if (node == nullptr) return;
    cout << node->key << " ";
    printPreorder(node->left);
    printPreorder(node->right);
}
```

```
// Postorder traversal
void printPostorder(Node* node) {
    if (node == nullptr) return;
    printPostorder(node->left);
    printPostorder(node->right);
    cout << 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.

The Call Stack: Recursive functions use something called "the call stack." When a program calls a function, that function goes on top of the call stack. This is similar to a stack of books. You add things one at a time. Then, when you are ready to take something off, you always take off the top item.

Q. 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

Code:

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

    cout << n << " ";
    pzz(n - 1);
    cout << n << " ";
    pzz(n - 1);
    cout << n << " ";
}
```

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.

PW SKILLS

THANK YOU !

PW SKILLS