PW SKILLS

Lesson Plan

Java
Recursion 1

PW SKILLS

Pre-requisites:

- Basics of programming like loops, conditionals, functions, and data structures (like arrays, lists, etc.)

List of Concepts Covered:

- Properties of Recursion
- Algorithmic Steps for recursion
- Memory Allocation in Recursion and the call stack
- Application of Recursion to solve various problems

What is Recursion?

In simple words, Recursion is a method that calls itself. The concept of recursion involves breaking down a problem into smaller subproblems and solving each subproblem by calling itself.

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 infinite loop will occur.

Why do we need Recursion?

Recursion is particularly useful when dealing with complex problems that can be broken down into smaller, more manageable subproblems.

Advantages:

- Recursive solutions can often be more concise and easier to read than their iterative counterparts.
- Reuse code effectively

Syntax:

```
function name(argument1 ,argument2..)
{
        return statement
}

Example:
Add two numbers:

public static int sumofNumbers(int a,int b)
{
  return a+b;
}
```

Function Calls Itself:

A function can call other functions, and by that logic, it can call itself as well.A function calling itself is known as recursion in programming.

Syntax of a function calling itself or Recursion:

```
    myRecursiveFunction(parameter) {

        if (base case)
            return base_case_value ;
        else
            return myRecursiveFunction(modified_parameter);
  //Recursive call
}
```

Base case: In simple terms, it is the very basic and smallest unit of any task.
The condition where the function stops calling itself or the recursive calls are stopped from growing in memory. The recursion stops when this case is met and the base case is required to avoid an infinite loop.

Recursive call: This is a function calling itself with a smaller input each time, which is similar to reducing a big task into smaller bits. It progressively moves us towards the base case.

Although a Call stack is invisible to a programmer, it is important to understand how it works, as it forms the core concept of recursion.

Call stack :

A call stack is a data structure in memory, used to manage the execution of function calls in a program. It keeps track of the order in which functions are called and their respective local variables and parameters.

The call stack operates on a Last In, First Out (LIFO) basis, meaning that the last function called is the first one to finish and be removed from the stack.

Each function call creates a new stack frame, which is a block of memory that stores information specific to that function call, including the return address (the location in the program where execution should continue after the function call completes).

In the case of recursive functions, each recursive call adds a new stack frame. The stack frames continue to stack up until the base case is reached, at which point the functions start to complete, and the stack frames are popped off.

To understand this better,

Let us take a straightforward and real-life situation of a call stack:
Mary needs to go to school, but she needs to pack lunch before she leaves her house. She makes a to-do list dividing her big task into four smaller tasks.

Now, let's relate this to a call stack:

Gather Ingredients: This is the first task that Mary needs to do. In the call stack, it's like putting a function or a task on top of the stack.

Prepare Food: Once Mary has gathered the ingredients, she moves on to the next task. Similarly, in the call stack, when a function is done, it's taken off the top of the stack.

Pack Lunch: After preparing the food, Mary can pack her lunch. In the call stack, when the last function or task is done, the stack is empty, and the overall process (or program) is complete.

So, in simple terms, the call stack is like Mary's to-do list. Each task (or function) is added to the top of the stack, and when a task is completed, it's taken off the top. This continues until all tasks are done, just like Mary going through her to-do list to get ready for school.

Coding Questions:

1. **Make a function that calculates the factorial of n using recursion.**

Code:

```
public class FactorialCalculator {

  public static int factorial(int n) {
        if (n == 0 || n == 1) {
            return 1;
        } else {
            return n * factorial(n - 1);
        }
    }

    public static void main(String[] args) {
        int result = factorial(number); //Replace number
here with any positive integer
        System.out.println("Factorial of number  is: " +
result);
    }

 }
```

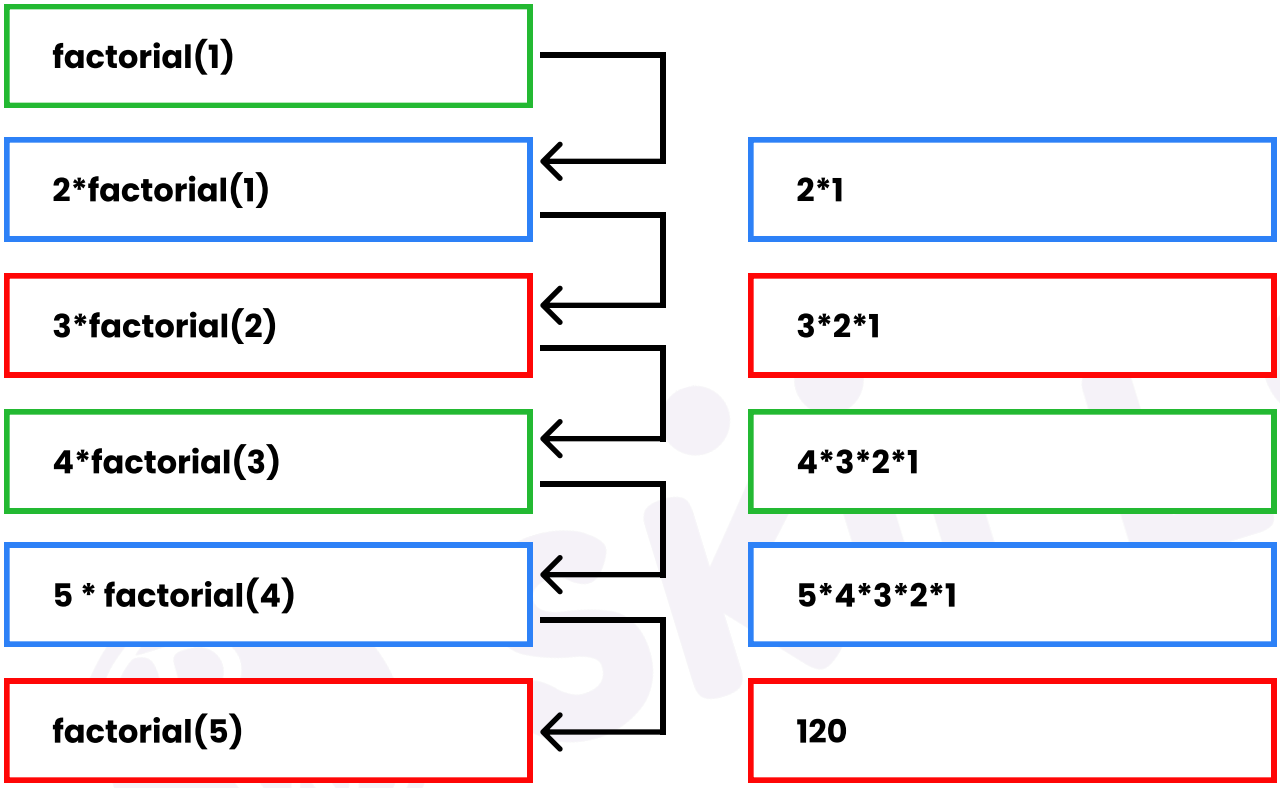
Input:

number=5

Output:

Factorial of number is: 120

Code Explanation:



Each stack frame is added as we recursively call the function reducing the current input by 1, and multiplying it by the current input. This continues till the base case is reached ie. either 0 or 1, where 1 is returned and the top of the stack frame is evaluated as 1 and popped off. The values are evaluated one-by-one from top to bottom of the stack, and after evaluating each expression, the top of the stack is cleared and the final output is returned.

Time and Space Complexity:

In the worst-case scenario, the time complexity of the recursive factorial function is O(n),since the function makes n recursive calls in total.

The space complexity of the recursive factorial function is O(n) as well. Since the maximum depth of the call stack is n (the number of recursive calls made).

2. Print n to 1

Code:

```
public class PrintNTo1 {

    public static void main(String[] args) {
             printNTo1(n); // Pass in n value

    }

    // Recursive method to print numbers from n to 1
    private static void printNTo1(int n) {
        // Base case: if n is less than 1, return
        if (n < 1) {
            return;
        }

                System.out.print(n + " ");

        // Recursive case: call the function with the next value of n
        printNTo1(n - 1);
    }
}
```

Input:

n=12

Output:

12 11 10 9 8 7 6 5 4 3 2 1

Code Explanation:

The printNumbers method is a recursive function that takes an integer n as its parameter. It has two parts:

If n is less than or equal to 0, the function returns without doing anything, stops the recursion.
If n is greater than 0, it prints the current value of n and then makes a recursive call to printNumbers with the argument n-1. This leads to a sequence of decreasing numbers until the base case is reached.

Time and Space Complexity:

The time complexity is O(n), since each recursive call, the program prints one number, and there are n recursive calls.The space complexity is O(n) due to the recursion stack.n each recursive call, a new stack frame is added to the call stack, and the maximum depth of the call stack is n.

3. Print 1 to n(extra parameter)

Code:

```
public class Numbers {

    static void printNumbers(int start, int n) {
        if (start > n) {
            return;         }
        System.out.print(start+" ");
        printNumbers(start + 1, n);
    }

    public static void main(String[] args) {
           printNumbers(1, n); //Pass an integer in place of n
    }

}
```

Input:

n=5

Output:

1 2 3 4 5

Code Explanation:

A recursive call is made to printNumbers with the incremented start and the input n, continuing the sequence of increasing numbers until the base case is met. If start is less than or equal to n, it prints the current value of start. If start is greater than n, the function returns, stopping the recursion. This is the condition to break out of the recursion.

Time and Space Complexity:

The time complexity is O(n), since each recursive call, the program prints one number, and there are n recursive calls.The space complexity is O(n) due to the recursion stack.n each recursive call, a new stack frame is added to the call stack, and the maximum depth of the call stack is n.

4. Print 1 to n (after recursive call)

```
public class PrintNumbers {

  static void printNumbers(int start, int n) {
            if (start > n) {
            return;
        }
```

```
            printNumbers(start + 1, n);

        // Print the current value of start after the recursive call
        System.out.print(start+" ");
    }



    public static void main(String[] args) {
            printNumbers(1, 7); //Pass a positive integer in place
of n
    }


}
```

Input:

n=7

Output:

7 6 5 4 3 2 1

Code Explanation:

If start is greater than n, the function returns, stopping the recursion. This is the condition to break out of the recursion.
It makes a recursive call to printNumbers with the incremented start and the same n, continuing the sequence of increasing numbers.
After the recursive call, it prints the current value of start, reversing the order of numbers

Time and Space Complexity:

The time complexity is O(n), since each recursive call, the program prints one number, and there are n recursive calls.The space complexity is O(n) due to the maximum depth of recursion stack being O(n).

5. Print sum from 1 to n (Parameterised)

Code:

```
public class SumCalculator {

    public static void main(String[] args) {
            int result = calculateSum(1, n, 0); //Pass a positive
integer value in place of sum
        System.out.println("Sum is: " + result);
    }
```

```
  static int calculateSum(int start, int end, int sum) {
              if (start > end) {
            return sum;
        }

        // Recursive case: add the current start to the sum and call
the function with the next start
        return calculateSum(start + 1, end, sum + start);
    }
}
```

Input:

n=5

Output:

Sum is: 15

Code Explanation:

The base case checks if start is greater than end. If true, it returns the current sum.In the recursive case, it adds the current value of start to the sum and makes a recursive call with the next value of start and the updated sum.Once the base case is reached,the sum is returned.

Time and Space Complexity:

The time complexity of this recursive solution is O(n), where n is the input value. This is because the recursive function makes n calls, each contributing O(1) work.The space complexity is O(n) due to the recursive call stack. In the worst case, the stack depth will be n, as each recursive call consumes space on the call stack.

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

Code:

```
public class SumCalculator {

    public static void main(String[] args) {
        int result = calculateSum(n);   // Pass an integer value in
place of n value
        System.out.println("Sum is: " + result);
    }

    private static int calculateSum(int n) {
        // Base case: if n is 1, return 1
        if (n == 1) {
            return 1;
        }
```

```
        // Recursive case: add n to the sum of the numbers from 1 to
n-1
        return n + calculateSum(n - 1);
    }
}
```

Input:

n=10

Output:

Sum is: 55

Code Explanation:

The base case checks if n is 1. If true, it returns 1 as the sum of integers from 1 to 1 is 1.In the recursive case, it adds the current value of n to the sum of the numbers from 1 to n-1 by making a recursive call with the value n-1.

Time and Space Complexity:

The time complexity of this recursive solution is O(n), whereas space complexity is O(n) as well(same as above explanation).

7. Make a function which calculates 'a' raised to the power 'b' using recursion.

Code:

```
public class PowerCalculator {

    public static void main(String[] args) {
        long result = power(base, exponent); //Pass in base and
exponent values.We use long here since the number maybe be huge since
its an exponent
        System.out.println("Exponential value is:"+ result);
    }

    // Recursive method to calculate a^b
    private static long power(int a, int b) {
        // Base case: if exponent is 0, return 1
        if (b == 0) {
            return 1;
        }

        // Recursive case: calculate a^(b-1) and multiply it by a
        return a * power(a, b - 1);
    }
}
```

Input:

Passing in 2 and 3 as values in power method

Output:

Exponential value is 8

Code Explanation:

The base case checks if b is 0. If true, it returns 1 as any number raised to the power 0 is 1.In the recursive case, it calculates a^(b-1) by making a recursive call with the value b-1 and then multiplies it by a.

Time and Space Complexity:

The time complexity of the provided program is O(b), where 'b' is the exponent. This is because, in each recursive call, the exponent decreases by 1 until it reaches 0 (base case), and there are 'b' recursive calls in total.The space complexity is determined by the maximum depth of the call stack during the recursive calls. In this case, the maximum depth is 'b' (the exponent), as each recursive call adds a frame to the call stack.

Calculating Time and Space complexity

Calculating the time and space complexity of a recursive function involves understanding the number of recursive calls made and the space required for each call. Here are some general steps to calculate time and space complexity for recursive functions:

Time Complexity:

Determine the number of recursive calls: Identify how many times the function is recursively called in terms of the input size.

Define the work done per call: Analyze the amount of work done in each recursive call in terms of constant time or the input size.

Write the recurrence relation: Express the time complexity in a recurrence relation that describes the number of times the function is called and the work done in each call.

Solve the recurrence relation: Solve the recurrence relation to obtain the overall time complexity of the algorithm.

Space Complexity:

Determine the maximum depth of the call stack:Identify how many times the function will be called recursively before reaching the base case. This gives you the maximum depth of the call stack.

Define the space used per call:Analyze the amount of space used in each call in terms of constant space or the input size.

Write the recurrence relation for space:Express the space complexity in a recurrence relation that describes the maximum depth of the call stack and the space used in each call.

Solve the recurrence relation for space:Solve the recurrence relation to obtain the overall space complexity of the algorithm.