Revision Note - Recursion

Recursion is a powerful programming technique that involves a function calling itself to solve a problem by breaking it down into smaller, simpler instances of the same problem. It allows for elegant and concise solutions to a wide range of problems by leveraging the concept of self-reference. Understanding how to write recursion code is essential to effectively use this technique.

How to write Recursion Code

When writing recursion code, there are three main components to consider: assumption, main logic, and base condition.

Assumption:

The assumption defines what the recursive function does. For the given problem of calculating the sum of all integers upto a given value N, we an assume:

```
int sum(int N) =>  returns the correct sum of first N natural Numbers
```

The assumption serves as the foundation for the recursive solution.

Main Logic:

The main logic represents the solution based on the assumption and the current problem. It defines how the problem is solved using the solutions of the subproblems. The main logic utilizes the assumption to derive the solution for the current problem based on the solutions of its subproblems.
For the sum of N natural numbers example, the main logic can be described as follows:

- Calculate the sum of the first N-1 natural numbers using recursion.
- Add N to the sum obtained from the previous step to get the final sum of the first N natural numbers.

```
sum(N) = sum(N-1) + N
```

Base Condition:

The base condition defines the terminating condition for the recursion. It specifies when the recursion should stop and return a result. The base condition is crucial to prevent infinite recursion and ensure that the recursion eventually reaches a valid stopping point.
For the sum of N natural numbers example, the base condition can be defined as:

- If N is 1, return 1.
- This base condition represents the smallest subproblem that can be directly solved without further recursion.

By combining these three components, we can construct a recursive solution. The recursion will continue breaking down the problem until it reaches the base condition, at which point it starts returning the results from the base cases and combines them to solve the larger problem.
Pseudo code for finding the sum of N natural numbers using recursion:

```
function sum(N):
    if N == 1:
        return 1 (Base condition)
    else:
        return sum(N-1) + N (Main logic)
```

In this recursive function, the assumption is that the sum of N natural numbers can be obtained by finding the sum of the first N-1 natural numbers and adding N to it.
Recursion offers an elegant way to solve various problems, but it requires careful consideration of the assumption, main logic, and base condition. It is important to ensure that the recursion converges towards the base case to avoid infinite recursion. Additionally, the efficiency of a recursive solution should be evaluated, as excessive recursion can lead to stack overflow and suboptimal performance.
By understanding the principles of recursion and practicing its application, programmers can harness its power to solve complex problems in a concise and intuitive manner.

Question 1 - Finding the N-th Fibonacci Number

Q. Finding the N-th Fibonacci Number using Recursion

The Fibonacci sequence is a series of numbers in which each number (after the first two) is the sum of the two preceding ones. To find the N-th Fibonacci number using recursion, we can use the following assumptions:

Assumption:

The function finonacci(N) calculates and returns the correct Nth Fibonacci term.

Main Logic:

The main logic involves defining the steps to calculate the N-th Fibonacci number based on the assumption. We will recursively call the function to calculate the (N-1)th and (N-2)th Fibonacci numbers and sum them to obtain the N-th Fibonacci number.

Base Condition:

The base condition is when N is either 0 or 1. In this case, the N-th Fibonacci number is simply the value of N itself.

Pseudo code for finding the N-th Fibonacci number using recursion:

```
function fibonacci(N):
    if N <= 1:
        return N (Base condition)
    else:
        return fibonacci(N-1) + fibonacci(N-2) (Main logic)
```

Explanation:

In the recursive function, the assumption is that the N-th Fibonacci number can be obtained by calculating the (N-1)th and (N-2)th Fibonacci numbers recursively. The base condition ensures that when N is either 0 or 1, the function directly returns N as the Fibonacci number.
By breaking down the problem into smaller subproblems and utilizing recursion, the function calculates the N-th Fibonacci number.

Question 2 - Finding Modular Power using Recursion

To find the power of A raised to the exponent p modulo MOD (A^p % MOD) using recursion, we can use the following assumptions:

Assumption:

The function power(A, p, MOD) calculates and returns the correct value of A^p % MOD
The power of A raised to p modulo MOD can be calculated by dividing p into smaller subproblems and utilizing the properties of modulo arithmetic.

Main Logic:

The main logic involves defining the steps to calculate the power of A modulo MOD based on the assumption. We will recursively call the function to calculate the power of A raised to half of p modulo MOD and then apply the necessary operations to obtain the final result.

Base Condition:

The base condition is when p is 0. In this case, the power of A raised to 0 modulo MOD is 1.
Pseudo code for finding A^p % MOD using recursion:

```
function power(A, p, MOD):
    if p == 0:
        return 1 (Base condition)
    else:
        half_power = power(A, p/2, MOD)
        result = (half_power * half_power) % MOD
        if p is odd:
            result = (result * A) % MOD
        return result
```

Explanation:

In the recursive function, the assumption is that the power of A raised to p modulo MOD can be obtained by calculating the power of A raised to half of p modulo MOD recursively and applying the necessary operations to obtain the final result.
By dividing the problem into smaller subproblems and leveraging recursion, the function calculates the power of A raised to p modulo MOD efficiently.

Time Complexity from given Recursive Relation

Given recursive relation is **f(n) = 2*f(n/2) + 1**

- Expand the recursive relation:
  $f(n) = 2f(n/2) + 1$
  $= 2 * [2 * f(n/4) + 1] + 1$
  $= 2^2 * f(n/4) + 2 + 1$
  $= 2^2 * [2f(n/8) + 1] + 2 + 1$
  $= 2^3 * f(n/8) + 2^2 + 2 + 1$
  .
  .
  .
  k times
  $= 2^k * f(n/2^k) + 2^{(k-1)}$
- Determine the value of k when $n/2^k = 1$:
  $n/2^k = 1$
  $n = 2^k$ **(Base Case)**
  Taking the logarithm base 2 of both sides:
  $log(n) = log(2^k)$
  $log(n) = k * log(2)$
  $k = log(n)$

So, **f(n) = n * f(n/n) + (n - 1) = 2n - 1**
From the deduced relation **f(n) = 2n - 1**, we can conclude that the time complexity is **O(n)**. The function grows linearly with the input size n.

Time and Space Complexity for Recursion

- **Space Complexity = maximum stack size during execution**
  - Space Complexity: O(n) for sum, O(n) for Fibonacci series, O(n) for factorial.
- **Time Complexity = Number of recursive calls * (Time needed per recursion)**
  - Time Complexity: O(n) for sum, O(2^n) for Fibonacci series, O(n) for factorial.

Problem 3 - Tower of Hanoi

Problem Description:

The Tower of Hanoi problem is a mathematical puzzle that consists of three towers (A, B, C) and N disks of different sizes placed on tower A. The goal is to move all the disks from tower A to tower C, using tower B as an intermediate tower. The problem follows certain

Constraints:

- Only one disk can be moved at a time.
- A larger disk cannot be placed on top of a smaller disk.
- 1 <= N <= 16.

Example:

Input:
N = 3 (Number of disks)
Output:
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C

Explanation:

The above output shows the sequence of moves required to solve the Tower of Hanoi problem with 3 disks. The disks are represented by numbers, with smaller numbers denoting smaller disks.

Brute Force Solution:

A brute force solution involves considering all possible combinations of moves to solve the problem. It would require exponential time complexity, making it inefficient for larger numbers of disks.

Optimal Solution:

The Tower of Hanoi problem can be efficiently solved using a recursive approach. The key idea is to break down the problem into smaller subproblems, by considering the top (N-1) disks as a single unit and moving the largest disk to the destination tower. This process is recursively applied until all disks are moved to the destination tower.
Pseudo Code:

```
towerOfHanoi(N, source, destination, auxiliary):
    if N == 0:
        return
    towerOfHanoi(N-1, source, auxiliary, destination)
    print("Move disk", N, "from", source, "to", destination)
    towerOfHanoi(N-1, auxiliary, destination, source)
```

Master Theorem for Finding Time Complexity

The master theorem is a useful tool for determining the time complexity of divide-and-conquer algorithms with recursive relations. It provides a framework for analyzing algorithms of the form:
`T(n) = a * T(n/b) + f(n)`,
where:

- `T(n)` represents the time complexity of the algorithm for an input size of n.
- `a` represents the number of recursive subproblems generated in each recursion.
- `n/b` represents the size of each subproblem.
- `f(n)` represents the time complexity of any extra work done outside the recursive calls.