**Assignment**

1. "**Write a recursive function pseudocode and calculate the nth Fibonacci number and use Big O notation to analyze its efficiency. Compare this with an iterative approach and discuss the pros and cons in terms of space and time complexity**."

function fibonacci (n):

if n <= 1:

return n

else:

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

Now, let’s analyze its efficiency using Big O notation.

In the recursive approach, for each Fibonacci number computed, two recursive calls are made until the base cases (n = 0 or n = 1) are reached. This leads to a time complexity of approximately O (2^n) because the number of function calls doubles with each increment of n.

Now, let's compare this with the iterative approach:

function fibonacci\_iterative(n):

if n <= 1:

return n

else:

fib = [0, 1]

for i from 2 to n:

fib.append (fib[i-1] + fib[i-2])

return fib[n]

The time complexity of this iterative approach is O(n), as it only requires a single loop through the range of numbers from 2 to n to compute the nth Fibonacci number.

In terms of space complexity, the recursive approach requires additional memory to store the call stack for each recursive call, leading to a space complexity of O(n) due to the depth of the recursion stack. The iterative approach, on the other hand, only requires space to store the Fibonacci sequence up to the nth number, leading to a space complexity of O(1) since the space used does not increase with the input size.

In summary, the recursive approach has a higher time complexity compared to the iterative approach, making it less efficient for large values of n. However, the iterative approach has a constant space complexity, while the recursive approach may consume more memory due to the recursion stack. Therefore, the choice between the two approaches depends on the specific requirements of the problem and the constraints on time and space.

**2. “Pseudocode and Flowchart for Sorting Algorithm - Write pseudocode and create a flowchart for a bubble sort algorithm. Provide a brief explanation of how the algorithm works and a simple array of integers to demonstrate a dry run of your algorithm**.”

procedure bubbleSort (A : list of sortable items)

n = length(A)

for i from 0 to n-1 do

for j from 0 to n-i-1 do

if A[j] > A[j+1] then

swap(A[j], A[j+1])

end if

end for

end for

end procedure

**PSEUDOCODE**

Start

Set n to length of A

for i from 0 to n-1 do

for j from 0 to n-i-1 do

if A[j] > A[j+1] then

swap (A[j], A[j+1])

end if

end for

end for

end

**Note**: Bubble Sort works by repeatedly stepping through the list, comparing each pair of adjacent items and swapping them if they are in the wrong order. This process is repeated until no swaps are needed, which indicates that the list is sorted.

**Example Dry Run**:

Let's take an array of integers [5, 3, 8, 1, 4]:

1. Pass 1:

Comparing 5 and 3, swap -> [3, 5, 8, 1, 4]

Comparing 5 and 8, no swap

Comparing 8 and 1, swap -> [3, 5, 1, 8, 4]

Comparing 8 and 4, swap -> [3, 5, 1, 4, 8]

2. Pass 2:

Comparing 3 and 5, no swap

Comparing 5 and 1, swap -> [3, 1, 5, 4, 8]

Comparing 5 and 4, swap -> [3, 1, 4, 5, 8]

3. Pass 3:

Comparing 3 and 1, swap -> [1, 3, 4, 5, 8]

Comparing 3 and 4, no swap

Comparing 4 and 5, no swap

Comparing 5 and 8, no swap

4. Pass 4:

No swaps needed.

The final sorted array is [1, 3, 4, 5, 8]

**FLOW CHART**

[Start] -> Input n flow chart

|

v

Check if n <= 1?

| \

| \

Yes No

| |

v v

Return n Initialize variables: fibPrev = 0, fibCurr = 1, i = 2

|

v

Loop while i <= n

|

|

v

fibNext = fibPrev + fibCurr

|

|

v

fibPrev = fibCurr

|

|

v

fibCurr = fibNext

|

|

v

Increment i

|

|

v

End Loop

|

|

v

Return fibCurr

|

|

v

[End]