Date: - 12th April 2024 Day 3

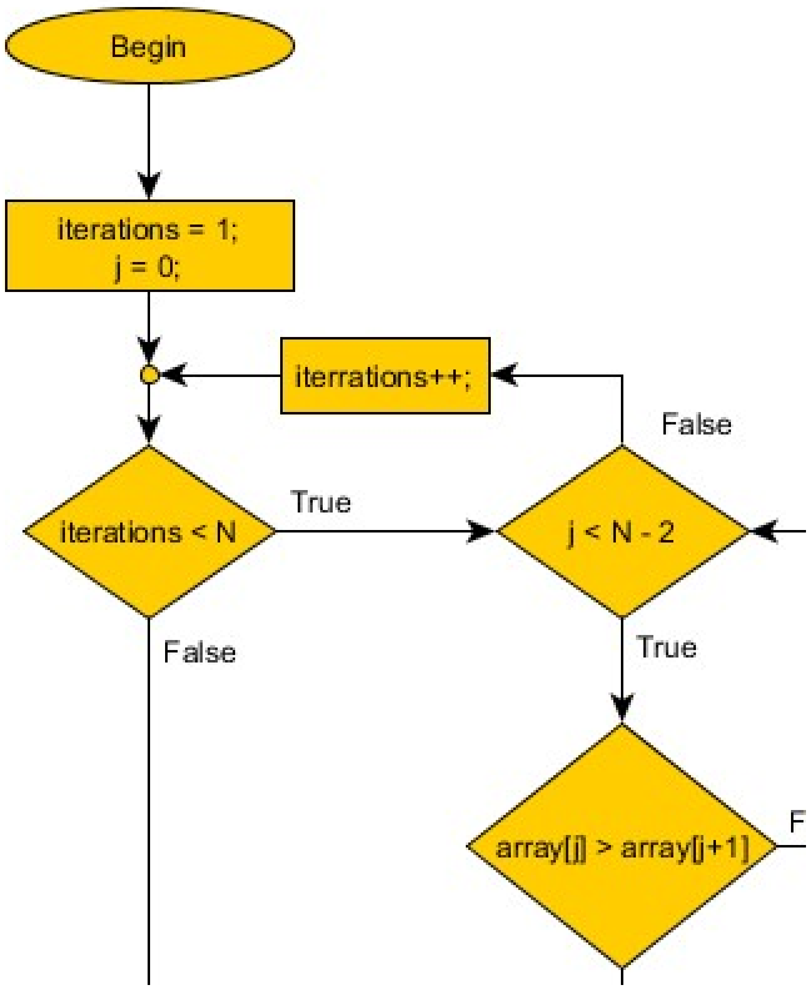
Topic:

1. Write the algorithm of Binary search.
2. Flowchart of Bubble short.
3. Algorithm of quick short.
4. Algorithm of heap short.
5. How to find the efficiency of an algorithm.
6. Big O Notation.
7. Time complicity.
8. PPT of Binary search.
9. 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.
10. Recursive Function and Efficiency Analysis - 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.

Answer:

1.
```
Step 1: set beg = lower_bound, end = upper_bound, pos = - 1
Step 2: repeat steps 3 and 4 while beg <=end
Step 3: set mid = (beg + end)/2
Step 4: if a[mid] = val
set pos = mid
print pos
go to step 6
else if a[mid] > val
set end = mid - 1
else
set beg = mid + 1
[end of if]
[end of loop]
Step 5: if pos = -1
print "value is not present in the array"
[end of if]
Step 6: exit
```

2.



3.

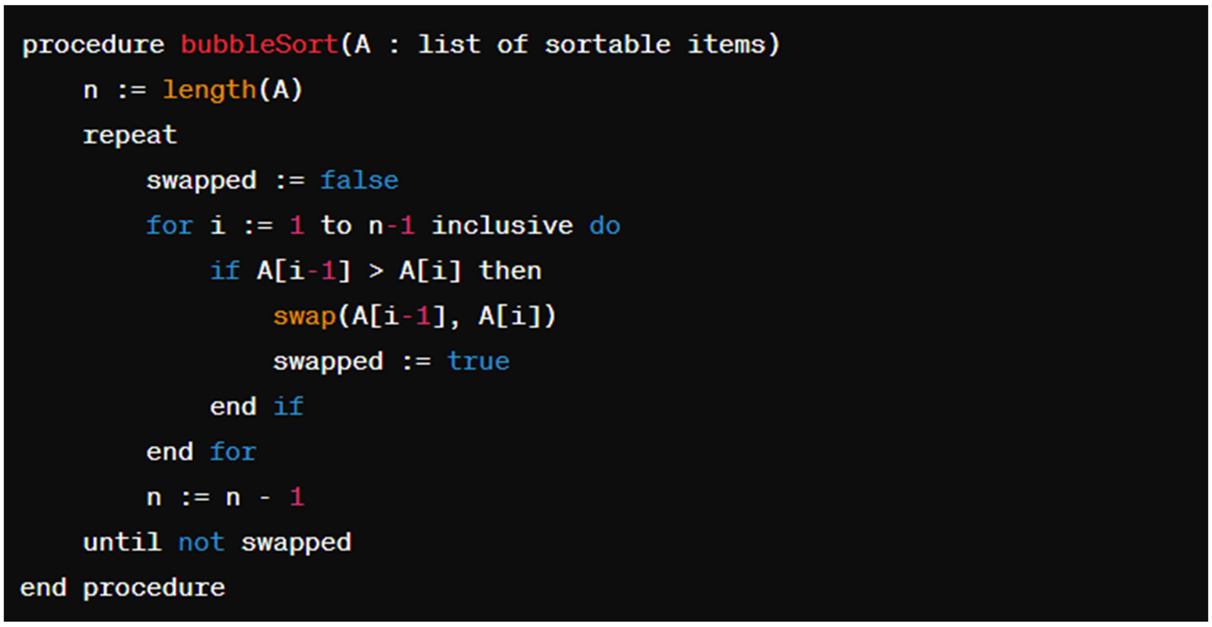
```
QUICKSORT (array A, start, end)
{
 1 if (start < end)
 2 {
3 p = partition(A, start, end)
4 QUICKSORT (A, start, p - 1)
5 QUICKSORT (A, p + 1, end)
6 }
}
```

4.

```
HeapSort(arr)
BuildMaxHeap(arr)
for i = length(arr) to 2
```

```
    swap arr[1] with arr[i]
       heap_size[arr] = heap_size[arr] ? 1
       MaxHeapify(arr,1)
End
```

5. Time and space complexity are the two main measures for calculating algorithm efficiency, determining how many resources are needed on a machine to process it. Where time measures how long it takes to process the algorithm, space measures how much memory is used.



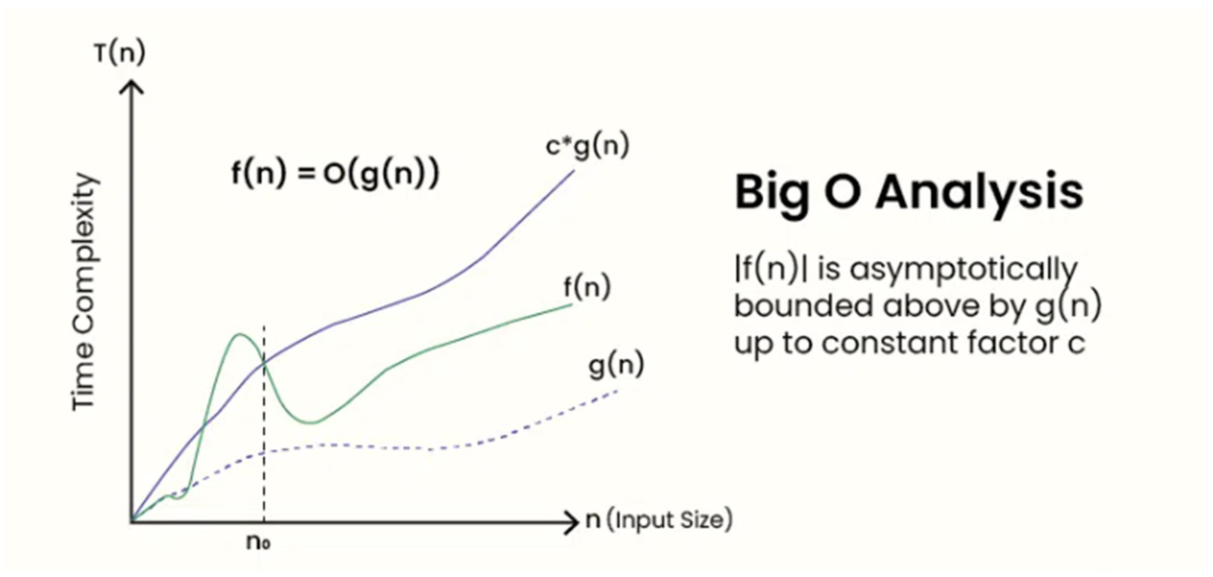
```
procedure bubbleSort(A : list of sortable items)
    n := length(A)
    repeat
        swapped := false
        for i := 1 to n-1 inclusive do
            if A[i-1] > A[i] then
                swap(A[i-1], A[i])
                swapped := true
            end if
        end for
        n := n - 1
    until not swapped
end procedure
```

6.

- Big O notation is a mathematical notation used to describe the upper bound of the time or space complexity of an algorithm in terms of the size of its input. It provides a way to express how the runtime or memory usage of an algorithm scales with the size of the input.

- In Big O notation, the letter "O" stands for "order of" or "order magnitude," and it is followed by a mathematical expression that represents the growth rate of the algorithm's resource usage as a function of the input size.

- Big O Notation is important because it helps analyze the efficiency of algorithms.
- It provides a way to describe how the runtime or space requirements of an algorithm grow as the input size increases.
- Allows programmers to compare different algorithms and choose the most efficient one for a specific problem.
- Helps in understanding the scalability of algorithms and predicting how they will perform as the input size grows.
- Enables developers to optimize code and improve overall performance.



7. PPT of Binary search.

8. Pseudocode for shorting algorithm

```
function bubbleSort(arr)
  n = arr.length
  for i from 0 to n-1
    for j from 0 to n-1-i
      if arr[j] > arr[j+1]
        swap(arr[j], arr[j+1])
  return arr
```

Write pseudocode and create a flowchart for a bubble sort algorithm

```
procedure bubbleSort(arr: array of integers)
  n = arr.length
  for i from 0 to n-1
    for j from 0 to n-1-i
      if arr[j] > arr[j+1]
        swap(arr[j], arr[j+1])
```

- It starts by comparing the first two elements of the list.
- If the first element is greater than the second element, it swaps them.
- It then moves to the next pair of elements and repeats the comparison and swapping process.
- This process continues until the algorithm reaches the end of the list.

- After completing one iteration through the list, the largest unsorted element will be at the end of the list.
- The algorithm then repeats the process for the remaining unsorted elements until the entire list is sorted.

Now, let's demonstrate a dry run of the algorithm with a simple array of integers:

Array: [5, 3, 8, 2, 1]

1st Pass:

Compare 5 and 3 (5 > 3), swap -> [3, 5, 8, 2, 1]
Compare 5 and 8 (no swap needed) -> [3, 5, 8, 2, 1]
Compare 8 and 2 (8 > 2), swap -> [3, 5, 2, 8, 1]
Compare 8 and 1 (8 > 1), swap -> [3, 5, 2, 1, 8]
2nd Pass:

Compare 3 and 5 (no swap needed) -> [3, 5, 2, 1, 8]
Compare 5 and 2 (5 > 2), swap -> [3, 2, 5, 1, 8]
Compare 5 and 1 (5 > 1), swap -> [3, 2, 1, 5, 8]
3rd Pass:

Compare 3 and 2 (3 > 2), swap -> [2, 3, 1, 5, 8]
Compare 3 and 1 (3 > 1), swap -> [2, 1, 3, 5, 8]
4th Pass:

Compare 2 and 1 (2 > 1), swap -> [1, 2, 3, 5, 8]
After the 4th pass, the array is sorted: [1, 2, 3, 5, 8].

9.

- **Base Case**: This is the terminating condition that stops the recursion. It defines the simplest form of the problem that does not require further recursion to solve. Without a base case, the recursion would continue indefinitely, leading to a stack overflow or infinite loop.

- **Recursive Case**: This is the part of the function that calls itself with a modified version of the problem. It reduces the original problem into smaller, simpler subproblems until the base case is reached.

- **Progress Towards Base Case**: Each recursive call must move the function closer to the base case. Otherwise, the recursion would not converge, and the function would not terminate.

- Recursive functions are commonly used to solve problems that exhibit recursive structure, such as tree traversal, factorial calculation, Fibonacci sequence generation, and sorting algorithms like Merge Sort or Quick Sort.

```
function recursiveFunction(parameter(s)):
  // Base case(s)
  if base_case_condition:
    return base_case_result

  // Recursive case(s)
  else:
    // Perform some operations or calculations
    // Call the function recursively with modified parameters
    return recursiveFunction(modified_parameters)
```

Efficiency Analysis:

- Bubble Sort, whether implemented iteratively or recursively, has a time complexity of O(n^2) in the worst and average case scenarios. This is because, in the worst case, it needs to make n*(n-1)/2 comparisons and swaps.

- The space complexity of the recursive implementation is O(n) due to the recursive calls. However, this can be optimized to O(1) by implementing tail recursion or converting it into an iterative approach.
- Recursive implementations of sorting algorithms like Bubble Sort are generally less efficient compared to their iterative counterparts due to the overhead of function calls and additional memory usage for the call stack.

```
function fibonacciRecursive(n: integer) returns integer
  if n <= 1
    return n
  else
    return fibonacciRecursive(n - 1) + fibonacciRecursive(n - 2)


function fibonacciIterative(n: integer) returns integer
  if n <= 1
    return n
  else
    fib_0 = 0
    fib_1 = 1
    for i from 2 to n
      fib = fib_0 + fib_1
      fib_0 = fib_1
    fib_1 = fib
  return fib
```

Recursive Approach:

Pros:

- Generally more concise and easier to understand for certain problems with recursive structures.
- Mimics the mathematical definition of the problem closely, making it easier to translate from mathematical expressions.

Cons:

- Exponential time complexity (O(2^n)) due to redundant calculations and overlapping subproblems, making it inefficient for large values of n.
- Exponential space complexity (O(n)) due to the function call stack, potentially leading to stack overflow errors for large values of n.

Iterative Approach:

Pros:

- Linear time complexity (O(n)) since it only needs to iterate through the sequence once, making it more efficient for large values of n compared to the recursive approach.
- Constant space complexity (O(1)) since it only requires a constant amount of space for storing variables, regardless of the input size n.

Cons:

- May be less intuitive for certain problems, especially those with recursive structures, compared to the recursive approach.
- Requires explicit management of variables and loop control, which may increase the complexity of the code compared to the recursive approach.