USN | | | | | | | | | |

Internal Assessment Test 1 –March 2025
Scheme & Solution

Sub:	ANALYSIS AND DESIGN OF ALGORITHMS					Sub Code:	BCS401	Branch:	ISE
Date:	**27/03/2025**	Duration:	90 min's	Max Marks:	50	Sem/Sec:	IV A, B & C		OBE

Answer any FIVE FULL Questions		MARKS	CO	RBT
1.	**Write an algorithm to find the maximum element in an array of n elements. Give the mathematical analysis of this non recursive algorithm.**	10	CO1	L2

Algorithm: - [5 Marks]
Analysis step by step: - [5 Marks]

Ans: -

Algorithm

Algorithm FindMaxElement(arr):
Input: An array arr of n elements
Output: Maximum element max_element
1. max_element = arr[0] // Initialize max_element to the first element of the array
2. for i = 1 to n-1 do // Iterate through the array starting from the second element
3. if arr[i] > max_element then // Compare current element with max_element
4. max_element = arr[i] // Update max_element if current element is greater
5. return max_element // Return the maximum element found

Mathematical Analysis

The measure of an input's size here is the number of elements in the array, i.e., n.

- There are two operations in the for loop's body:
  - o The comparison A[i]> maxval and
  - o The assignment max val←A[i].
- The comparison operation is considered as the algorithm's basic operation, because the comparison is executed on each repetition of the loop and not the assignment.
- The number of comparisons will be the same for all arrays of size n; therefore, there is no need to distinguish among the worst, average, and best cases here.
- Let C(n) denotes the number of times this comparison is executed. The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bounds 1 and n − 1, inclusive. Therefore, the sum for C(n) is calculated as follows:

–
() = ∑
=

	i.e., Sum up 1 in repeated n-1 times
	–
	() = ∑ = − ∈ ()
	
	• **Time Complexity**: The algorithm involves a single traversal of the array, which means it runs in O(n)O(n)O(n) time, where nnn is the number of elements in the array. This is because it checks each element exactly once to determine if it is greater than the current max_element.
	• **Space Complexity**: The algorithm uses only a constant amount of extra space O(1)O(1)O(1) (for max_element), regardless of the size of the input array. Hence, the space complexity is constant.
	
	Example:
	
	Let's consider an array arr = [5, 2, 9, 1, 7].
	
	1. Initialize max_element = 5.
	2. Iterate through the array:
	   ○ arr[1] = 2, 2 < 5 (no update),
	   ○ arr[2] = 9, 9 > 5 (update max_element = 9),
	   ○ arr[3] = 1, 1 < 9 (no update),
	   ○ arr[4] = 7, 7 > 9 (no update).
	3. Return max_element = 9.
	
	Thus, the maximum element in the array [5, 2, 9, 1, 7] is 9, and the algorithm correctly identifies it.			

2.	**Apply a quick sort algorithm to sort the list E, X, A, M, P, L, E, S in alphabetical order. Draw the tree of recursive calls made.** **Algorithm: - [5 Marks]** **Solution step by step: - [5 Marks]** **Ans: -** Array and partition the other elements into two sub-arrays according to whether they are less than or greater than the pivot. The sub-arrays are then recursively sorted. **Steps of Quick Sort:** 1. **Choose a Pivot**: Select an element from the array as the pivot (typically the last element in this example). 2. **Partitioning**: Rearrange the array so that all elements less than the pivot are on its left, and all elements greater than the pivot are on its right. 3. **Recursively Apply**: Recursively apply the above steps to the sub-arrays formed by partitioning until the entire array is sorted.	10	CO2	L3

Implementation:

Let's apply Quick Sort step-by-step to ["E", "X", "A", "M", "P", "L", "E"]:

1. **Initial Array**: ["E", "X", "A", "M", "P", "L", "E"]
2. **Choose Pivot**: Let's choose the last element as the pivot. In this case, "E".
3. **Partitioning**:
   - Rearrange elements so that all elements less than "E" come before it, and all elements greater than "E" come after it.

   After partitioning: ["A", "E", "M", "P", "L", "E", "X"]

   - "E" is now in its correct position.
4. **Recursive Calls**:
   - Apply Quick Sort recursively to the sub-arrays to the left and right of "E".

Left sub-array: ["A", "E", "M", "P", "L", "E"] Right sub-array: ["X"]
Now, let's sort each of these sub-arrays:
For the left sub-array ["A", "E", "M", "P", "L", "E"]:

- Choose pivot: Let's choose the last element, "E".
- Partitioning: After partitioning, we get: ["A", "E", "L", "E", "M", "P"]
  Left sub-array: ["A"] Right sub-array: ["L", "E", "M", "P", "E"]

- Recursively sort each sub-array:
  - Left sub-array ["A"] is already sorted.
  - Right sub-array ["L", "E", "M", "P", "E"]:
  - Choose pivot: Let's choose the last element, "E".
  - Partitioning: After partitioning, we get: ["E", "E", "L", "M", "P"]
  Left sub-array: ["E", "E"] Right sub-array: ["L", "M", "P"]

  - Recursively sort each sub-array:
    - Left sub-array ["E", "E"] is already sorted.
    - Right sub-array ["L", "M", "P"]:
    - Choose pivot: Let's choose the last element, "P".
    - Partitioning: After partitioning, we get: ["L", "M", "P"]
    Left sub-array: ["L"] Right sub-array: ["M", "P"]

    - Recursively sort each sub-array:

    - Left sub-array ["L"] is already sorted.
    - Right sub-array ["M", "P"] is sorted after partitioning.

Combine all sorted sub-arrays: ["A", "E", "E", "L", "M", "P", "E"]
For the right sub-array ["X"]:

- Since it has only one element, it is already sorted.

5. **Combine Results**:

- After all recursive calls and combining results, the sorted array is ["A", "E", "E", "L", "M", "P", "X"].

Tree of Recursive Calls:

Here's the tree structure showing the recursive calls made during the Quick Sort process:
css
Copy code

```
["E", "X", "A", "M", "P", "L", "E"]

     QuickSort(["E", "X", "A", "M", "P", "L", "E"])
                 |
    +-------------+-----------------+
    |                               |
QuickSort(["A", "E", "M", "P", "L", "E"]) QuickSort(["X"])
    |                               |
  +---+---+                       sorted
  |       |                      ["X"]
  QSA(["A", "E", "L", "E", "M", "P"])
      |
    +---+---+
    |       |
    QSA(["A", "E", "E"]) QSA(["L", "E", "M", "P"])
           |
       +---+---+
       |       |
       QSA(["E", "E"]) QSA(["L", "M", "P"])
               |
          +---+---+
          |       |
          sorted  QSA(["M", "P"])
          ["E", "E"] |
                  |
             +---+---+
             |       |
             sorted  sorted
             ["L"]   ["M", "P"]
```

Each node in the tree represents a recursive call to Quicksort function, with the arrows indicating the flow of execution. The sub-arrays being sorted are shown at each level of recursion, with "sorted" indicating the sorted result of each sub-problem.
This tree visualization helps illustrate how Quick Sort recursively partitions and sorts the original array until all sub-arrays are sorted, resulting in the final sorted array ["A", "E", "E", "L", "M", "P", "X"].

3.	**Give the general plan for analysing time efficiency of Recursive algorithms and analyse the Tower of Hanoi Recursive algorithm.** **Description: - [5 Marks]** **Solution step by step: - [5 Marks]** **Ans: -** **General Plan for Analysing Time Efficiency of Recursive Algorithms:** Analysing the time efficiency of recursive algorithms involves understanding how the algorithm's time complexity evolves with respect to the size of its input. Here's a general plan for analysing the time efficiency of recursive algorithms: **1.Identify Recursive Structure: -** Understand how the recursive algorithm divides the problem into smaller subproblems and recursively solves them. **2.Recurrence Relation: -** Define a recurrence relation that describes the time complexity T(n) of the algorithm in terms of the size of the input n. The recurrence relation expresses how the time complexity of the algorithm for an input of size n relates to the time complexities of the algorithm for smaller inputs. **3.Base Case: -** Identify the base case(s) where the recursion stops and the solution is directly computed without further recursion. The base case typically has a constant time complexity. **4.Solve the Recurrence:** - Solve the recurrence relation to determine the overall time complexity of the algorithm. This step involves finding a closed-form solution or asymptotic bounds (Big O notation) for T(n). **5.Summarize Time Complexity:** - Summarize the time complexity of the algorithm using Big O notation or another suitable asymptotic notation. This notation provides an upper bound on the growth rate of the algorithm's time complexity as the input size n increases. **Analysis of Tower of Hanoi Recursive Algorithm:** The Tower of Hanoi is a classic example of a recursive algorithm. It consists of three rods and a number of disks of different sizes that can slide onto any rod. The objective is to move the entire stack of disks from the first rod to the third rod, adhering to the following rules: 1. Only one disk can be moved at a time.	10	CO1	L2

2. A disk can only be moved if it is the uppermost disk on a stack.

3. No disk may be placed on top of a smaller disk.

Recursive Algorithm for Tower of Hanoi:

```
Algorithm TowerOfHanoi (n, source, auxiliary, target):

   Input: Number of disks n, source rod, auxiliary rod, target rod

   Output: Instructions to move n disks from source to target using auxiliary rod

   if n == 1 then

      Move disk from source to target

   else

      TowerOfHanoi (n-1, source, target, auxiliary) // Move top n-1 disks from source to auxiliary

      Move disk from source to target              // Move the nth disk from source to target

      TowerOfHanoi (n-1, auxiliary, source, target) // Move n-1 disks from auxiliary to target
```

Time Complexity Analysis:

To analyse the time complexity of the Tower of Hanoi recursive algorithm:

1.Identify Recursive Structure: The algorithm divides the problem of moving n disks into smaller sub-problems, where each sub-problem involves moving n-1 disks.

2.Recurrence Relation: Let T(n) denote the number of moves required to solve the Tower of Hanoi problem with n disks. The recurrence relation is:

$T(n)=2T(n-1)+1T(n) = 2T(n-1) + 1T(n)=2T(n-1)+1$

oThe term $2T(n-1)2T(n-1)2T(n-1)$ accounts for the two recursive calls to solve the sub-problems with n-1 disks.

oThe +1+1+1 accounts for the move of the largest disk from the source rod to the target rod.

	3.Base Case: The base case occurs when n = 1, where only one move is required: T (1) =1T (1) = 1T (1) =1. **4.Solve the Recurrence:** Solve the recurrence relation to find the closed-form solution for T(n): oBy solving recursively, we get T(n)=2n−1T(n) = 2^n - 1T(n)=2n−1. **5.Time Complexity:** Therefore, the time complexity of the Tower of Hanoi algorithm, in terms of the number of moves required, is O(2n)O(2^n)O(2n). This exponential time complexity indicates that the number of moves grows exponentially with the number of disks.			

4.	**Design a divide and conquer algorithm for computing the number of levels in a binary tree (In particular, the algorithm must return 0 and 1 for the empty and single node trees, respectively). What is the time efficiency class of your algorithm?**
	
	Description: - [5 Marks]
	
	Solution step by step: - [5 Marks]
	
	Ans:
	Algorithm: Compute Levels (T)
	1. **Base Case:**
	o If the tree TTT is empty (T=NULLT = NULLT=NULL), return **0**.
	o If the tree has only one node (i.e., no left or right children), return **1**.
	2. **Recursive Case:**
	o Recursively compute the number of levels in the left and right subtrees.
	o The height of the tree is **1 + max (left-subtree height, right-subtree height)**.
	
	Pseudocode:
	def ComputeLevels(T):
	    if T is None:
	        return 0
	    left_levels = ComputeLevels(T.left)
	    right_levels = ComputeLevels(T.right)
	    return 1 + max (left_levels, right_levels)
		10	CO1	L3

	Time Complexity Analysis: • Let nnn be the number of nodes in the binary tree. • The function makes two recursive calls (one for the left subtree and one for the right subtree) at each node. • If the tree is **balanced**, each recursive call works on approximately n/2n/2n/2 nodes, leading to the recurrence: T(n)=2T(n/2) +O(1)T(n) = 2T(n/2) + O(1)T(n)=2T(n/2) +O (1) Using the **Master Theorem/Substitution Method** (a=2, b=2, d=0a = 2, b = 2, d = 0a=2, b=2, d=0), we get T(n)=O(n)T(n) = O(n)T(n)=O(n). • If the tree is **skewed**, the recursion goes as deep as nnn, leading to O(n)O(n)O(n) complexity in the worst case. **Final Complexity:** O(n)O(n)O(n) This is optimal since we must visit every node at least once to determine the number of levels.			
5.	**Sort the following element using Merge Sort 10,5,7,6,1,7,8,3,2,9,4** **Algorithm: - [5 Marks]** **Solution step by step: - [5 Marks]** **Ans: -** To sort the elements [10, 5, 7, 6, 1, 7, 8, 3, 2, 9] using Merge Sort, we'll follow these steps: **Merge Sort Algorithm Explanation:** Merge Sort is a divide-and-conquer algorithm that works as follows: 1. **Divide**: Divide the array into two halves recursively until each sub-array contains only one element or is empty. 2. **Conquer**: Merge the smaller sorted arrays (sub-arrays) into larger sorted arrays until the whole array is merged. 3. **Merge Function**: The merge function combines two sorted arrays into a single sorted array. **Steps for Sorting [10, 5, 7, 6, 1, 7, 8, 3, 2, 9] using Merge Sort:**	10	CO2	L3

1. **Divide** the array into halves recursively until each sub-array contains one element:
    - [10, 5, 7, 6, 1, 7, 8, 3, 2, 9]
    - Divide into [10, 5, 7, 6, 1] and [7, 8, 3, 2, 9]
    - Continue dividing until each sub-array contains one element.

2. **Merge** the sorted sub-arrays:
    - Merge [10] and [5] to get [5, 10]
    - Merge [7] and [6] to get [6, 7]
    - Merge [1] and [7] to get [1, 7]
    - Merge [8] and [3] to get [3, 8]
    - Merge [2] and [9] to get [2, 9]

    Now, recursively merge these pairs until the entire array is sorted.

Sorted Array using Merge Sort:

After sorting through the recursive merges, the sorted array will be:

[1,2,3,5,6,7,7,8,9,10]

Detailed Steps with Merge Function:

Let's go through the detailed steps with merging:

1. Divide the array recursively:
    - [10, 5, 7, 6, 1, 7, 8, 3, 2, 9]
    - Divide into [10, 5, 7, 6, 1] and [7, 8, 3, 2, 9]
    - Continue dividing until each sub-array contains one element.

2. Merge the divided arrays:
    - Merge [10] and [5] to get [5, 10]
    - Merge [7] and [6] to get [6, 7]
    - Merge [1] and [7] to get [1, 7]
    - Merge [8] and [3] to get [3, 8]
    - Merge [2] and [9] to get [2, 9]

3. Continue merging the sorted arrays:
    - Merge [5, 10] and [1, 7] to get [1, 5, 7, 10]
    - Merge [6, 7] and [3, 8] to get [3, 6, 7, 8]
    - Merge [2, 9] and [3, 6, 7, 8] to get [2, 3, 6, 7, 8, 9]

4. Finally, merge [1, 5, 7, 10] and [2, 3, 6, 7, 8, 9] to get the sorted array:
    - [1, 2, 3, 5, 6, 7, 7, 8, 9, 10]

	Therefore, the sorted array using Merge Sort for the elements [10, 5, 7, 6, 1, 7, 8, 3, 2, 9] is [1, 2, 3, 5, 6, 7, 7, 8, 9, 10].			
6.	**Apply both merge & quick sort algorithms to sort the characters VTUBELAGAVI** **Algorithm: - [5 Marks]** **Solution step by step: - [5 Marks]** **Ans: -** **Merge Sort Algorithm:** Merge Sort is a divide-and-conquer algorithm that divides the array into two halves, recursively sorts each half, and then merges them back together in sorted order. **Steps for Merge Sort:** 1. **Divide**: Divide the string into two halves. 2. **Conquer**: Recursively sort each half using Merge Sort. 3. **Combine**: Merge the sorted halves to produce the sorted output. **Implementation:** Merge Sort: 1. Split the string "VTUBELAGAVI" into "VTUBELA" and "GAVI". 2. Recursively sort "VTUBELA" and "GAVI". 3. Merge "AELTUBV" (sorted "VTUBELA") and "AAGIV" (sorted "GAVI"). 4. Combine and sort "AELTUBV" and "AAGIV" to get "AAEGILTVUB". **Quick Sort Algorithm:** Quick Sort is a divide-and-conquer algorithm that selects a pivot element, partitions the array around the pivot, recursively sorts the sub-arrays, and combines them. **Steps for Quick Sort:** 1. **Partitioning**: Choose a pivot (often the last character), partition the string around the pivot, placing smaller characters to the left and larger characters to the right. 2. **Recursively Sort**: Recursively apply Quick Sort to the left and right partitions. 3. **Combine**: Combine the sorted partitions. **Implementation:** Quick Sort: 1. Choose pivot 'I' (last character) for "VTUBELAGAVI". 2. Partition: "AEGAVIVTUBL".	10	CO2	L3

	3. Recursively sort "AEGAVI" and "UBL". 4. Final sorted string: "AAEGILTVUB". **Sorted Output:** • **Merge Sort**: "AAEGILTVUB" • **Quick Sort**: "AAEGILTVUB"			