USN | | | | | | | | | | |

Internal Assessment Test 1 –June 2024

Sub:	**ANALYSIS AND DESIGN OF ALGORITHMS**					Sub Code:	BCS401	Branch:	ISE
Date:	**/05/2024**	Duration:	90 min's	Max Marks:	50	Sem/Sec:	III A, B & C		OBE

	Answer any FIVE FULL Questions	MARKS	CO	RBT
1.a	Define an algorithm.Discuss the characteristics of an algorithm.	4	CO1	L1
1.b	Write an algorithm to find the maximum element in an array of n elements.Give the mathematical analysis of this non recursive algorithm.	6	CO1	L2
2.a	Explain the general plan for analyzing the efficiency of a recursive algorithm. Write the algorithm to find a factorial of a given number. Derive its efficiency.	6	CO1	L2
2.b	Apply a quick sort algorithm to sort the list E, X, A, M, P, L, E in alphabetical order. Draw the tree of recursive calls made.	4	CO2	L3
3.a	List out the advantages and disadvantages of divide and conquer approach.	5	CO2	L2
3.b	Sort the following element using Merge Sort 10,5,7,6,1,7,8,3,2,9	5	CO2	L3

4.a	Explain in brief the basic asymptotic efficiency classes	5	CO1	L2
4.b	Apply both merge & quick sort algorithms to sort the characters VTUBELAGAVI	5	CO2	L3
5.a	Give the general plan for analyzing time efficiency of Recursive algorithms and also analyze the Tower of hanoi Recursive algorithm.	6	CO1	L2
5.b	Discuss how quick sort works to sort an array; trace the following data set. 1,1,9,9,5,5,6,6.	4	CO2	L3

6.a	Consider the following algorithm.
	ALGORITHM Mystery(n)
	//Input: A nonnegative integer n
	S ← 0
	for i ← 1 to n do
	S ← S + i ∗ i
	return S
	a. What does this algorithm compute?
	b. What is its basic operation?
	c. How many times is the basic operation executed?
	d. What is the efficiency class of this algorithm?
	e. Suggest an improvement, or a better algorithm altogether, and indicate its efficiency class. If you cannot do it, try to prove that, in fact, it cannot be done.
		5	CO2	L3

6.b	Compare the order of growth of log (n) & n^2;	5	CO1	L2

CI CCI HOD

1.a. An algorithm is a precise step-by-step sequence of instructions designed to perform a specific task or solve a particular problem. It's a fundamental concept in computer science and mathematics, serving as a blueprint or recipe for carrying out computations, data processing, and automated reasoning.

Characteristics of an Algorithm:

1. **Well-defined**: Every step of the algorithm must be clear and unambiguous. There should be no room for interpretation or uncertainty in how each instruction is to be executed.
2. **Input**: An algorithm takes zero or more inputs, which are the initial data or values on which the algorithm operates. These inputs are processed according to the defined steps.
3. **Output**: After processing the inputs according to the algorithm's instructions, it produces one or more outputs. The outputs are the results of applying the algorithm to the given inputs.
4. **Finiteness**: An algorithm must terminate after a finite number of steps. This means there should be a clear endpoint where the algorithm completes its task and produces the desired output.
5. **Effectiveness**: Also known as computability, an algorithm must be effective in the sense that its steps can be carried out and executed using available resources (such as time and memory) within a feasible time frame.
6. **Feasibility**: The algorithm must be practical and feasible for implementation. This means that while it might involve complex computations, those computations should be possible to perform given the constraints of available computational resources.
7. **Uniqueness**: For a given problem, there can be multiple algorithms that solve it, but each algorithm must provide the same output for the same input, adhering to the problem's requirements.
8.

1.b. 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:

- **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.a. General Plan for Analyzing Recursive Algorithm Efficiency

1. **Identify Recursive Structure**: Understand how the algorithm breaks down the problem into smaller sub-problems and recursively solves them.
2. **Recurrence Relation**: Define a recurrence relation that describes the time complexity of the algorithm in terms of the size of the input.
3. **Base Case**: Identify the base case(s) where the recursion stops and directly provides a solution without further recursion.
4. **Time Complexity**:
   - **Recurrence Relation**: Formulate a recurrence relation that expresses the time complexity T(n) of the algorithm, where n is the size of the input.
   - **Solve the Recurrence**: Solve the recurrence relation to determine the overall time complexity of the algorithm.
5. **Space Complexity**: Evaluate the space complexity, which includes the memory required to execute the algorithm. This typically involves analyzing the depth of recursion and any additional space used by data structures.

Example: Recursive Algorithm to Find Factorial

Here's an algorithm to find the factorial of a given number using recursion:

```
Algorithm Factorial(n):
    Input: A non-negative integer n
    Output: Factorial of n, denoted as n!

    1. if n == 0 or n == 1 then
    2.     return 1   // Base case: factorial of 0 or 1 is 1
    3. else
    4.     return n * Factorial(n-1)   // Recursive case: n! = n * (n-1)!
```

Analysis of Factorial Recursive Algorithm

Time Complexity:

- **Recurrence Relation**: The time complexity T(n) can be expressed as:

  T(n)=T(n−1)+O(1)T(n) = T(n-1) + O(1)T(n)=T(n−1)+O(1)

  Here, T(n−1)T(n-1)T(n−1) represents the time complexity of computing factorial for n−1n-1n−1, and O(1)O(1)O(1) accounts for the constant time operations (comparison and multiplication).

- **Base Case**: The recursion stops when n=0n = 0n=0 or n=1n = 1n=1, where the time complexity is constant O(1)O(1)O(1).
- **Solution of Recurrence**: Solving the recurrence:

  T(n)=T(n−1)+O(1)=T(n−2)+O(1)+O(1)=…=T(1)+O(1)+…+O(1)T(n) = T(n-1) + O(1) = T(n-2) + O(1) + O(1) = \ldots = T(1) + O(1) + \ldots + O(1)T(n)=T(n−1)+O(1)=T(n−2)+O(1)+O(1)=…=T(1)+O(1)+…+O(1)

  The recursion unwinds until the base case is reached, so T(n)=O(n)T(n) = O(n)T(n)=O(n).

Therefore, the time complexity of the factorial recursive algorithm is O(n)O(n)O(n), where nnn is the input number.

Space Complexity:

- **Depth of Recursion**: The space complexity is determined by the maximum depth of the recursion stack, which is O(n)O(n)O(n) in this case, because the algorithm will recurse nnn times before reaching the base case.
- **Additional Space**: Apart from the recursion stack, the algorithm uses a constant amount of additional space O(1)O(1)O(1) for storing local variables and parameters.

2.b. array and partitioning 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.

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.a. ChatGPT

The divide and conquer approach is a fundamental algorithmic technique that involves breaking down a problem into smaller sub-problems, solving each sub-problem recursively, and then combining the solutions to the sub-problems to solve the original problem. This approach has several advantages and disadvantages:

Advantages of Divide and Conquer:

1. **Efficiency**: Divide and conquer algorithms often have good time complexity because they reduce the problem size at each recursive step. This efficiency makes them suitable for large input sizes.
2. **Parallelism**: Many divide and conquer algorithms can be parallelized effectively. Since independent sub-problems are solved concurrently, this approach is well-suited for parallel computing environments.
3. **Simplicity**: The recursive nature of divide and conquer algorithms can lead to simpler and cleaner code compared to iterative solutions, especially for problems that are naturally divided into smaller parts.
4. **Modularity**: Divide and conquer encourages breaking down problems into smaller, manageable sub-problems. Each sub-problem can be solved independently and then combined with others, promoting modular design and easier debugging.
5. **Optimization Potential**: In many cases, divide and conquer algorithms can be optimized further by choosing appropriate strategies for dividing the problem and combining sub-solutions.
6. **Versatility**: This approach is applicable to a wide range of problems in various domains, including sorting (e.g., QuickSort, MergeSort), searching (e.g., Binary Search), and optimization (e.g., Strassen's Matrix Multiplication).

Disadvantages of Divide and Conquer:

1. **Overhead**: There can be overhead associated with dividing the problem, managing recursive calls, and combining results. This overhead might make the algorithm less efficient for smaller problem sizes or when recursion depth is significant.
2. **Space Complexity**: Divide and conquer algorithms often require additional space for recursion stack and intermediate data structures. This space overhead can be a concern for problems with large input sizes.
3. **Difficulty in Partitioning**: In some cases, determining an effective strategy for dividing the problem into sub-problems can be complex. Poor partitioning strategies can lead to inefficiencies or incorrect solutions.
4. **Not Always Optimal**: While divide and conquer is efficient for many problems, it's not always the most optimal approach. Some problems might have specialized algorithms that outperform divide and conquer in terms of time or space complexity.
5. **Recursive Depth**: Deep recursion can lead to stack overflow errors, especially in environments with limited stack space or when handling extremely large input sizes.
6. **Not Always Stable**: Some divide and conquer algorithms may not preserve the order of equal elements (e.g., QuickSort), which is a consideration in certain applications.

3.b.Sort the following element using Merge Sort 10,5,7,6,1,7,8,3,2,9

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:

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

4.a. Asymptotic efficiency classes, also known as asymptotic notations, are used to describe the time or space complexity of algorithms in relation to the size of their input. Here are the basic asymptotic efficiency classes commonly used in algorithm analysis:

1. O-notation (Big O)

- **Definition**: Represents the upper bound of an algorithm's growth rate, indicating the worst-case scenario.
- **Usage**: It provides an upper limit on the rate of growth of the function describing the algorithm's complexity.
- **Example**: O(n)O(n)O(n), O(n2)O(n^2)O(n2), O(logn)O(\log n)O(logn)
- **Interpretation**: An algorithm with O(n)O(n)O(n) time complexity means that its execution time grows linearly with the size of the input nnn.

2. Ω-notation (Big Omega)

- **Definition**: Represents the lower bound of an algorithm's growth rate, indicating the best-case scenario.
- **Usage**: It provides a lower limit on the rate of growth of the function describing the algorithm's complexity.
- **Example**: Ω(n)\Omega(n)Ω(n), Ω(n2)\Omega(n^2)Ω(n2), Ω(logn)\Omega(\log n)Ω(logn)
- **Interpretation**: An algorithm with Ω(n2)\Omega(n^2)Ω(n2) time complexity means that its execution time grows at least quadratically with the size of the input nnn.

3. Θ-notation (Big Theta)

- **Definition**: Represents the tight bound of an algorithm's growth rate, indicating both the upper and lower bounds.
- **Usage**: It describes the exact asymptotic behavior of the algorithm's complexity.
- **Example**: Θ(n)\Theta(n)Θ(n), Θ(n2)\Theta(n^2)Θ(n2), Θ(logn)\Theta(\log n)Θ(logn)
- **Interpretation**: An algorithm with Θ(n)\Theta(n)Θ(n) time complexity means that its execution time grows linearly with the size of the input nnn, and it also implies that the worst-case and best-case scenarios have the same growth rate.

4. o-notation (Little o)

- **Definition**: Represents an upper bound that is not tight; it denotes a growth rate strictly less than the upper bound.
- **Usage**: It provides an upper limit on the rate of growth of the function, which is not asymptotically tight.
- **Example**: o(n)o(n)o(n), o(n2)o(n^2)o(n2), o(log⁡n)o(\log n)o(logn)
- **Interpretation**: An algorithm with o(n2)o(n^2)o(n2) time complexity means that its execution time grows slower than quadratic time with the size of the input nnn.

5. ω-notation (Little Omega)

- **Definition**: Represents a lower bound that is not tight; it denotes a growth rate strictly greater than the lower bound.
- **Usage**: It provides a lower limit on the rate of growth of the function, which is not asymptotically tight.
- **Example**: ω(n)\omega(n)ω(n), ω(n2)\omega(n^2)ω(n2), ω(log⁡n)\omega(\log n)ω(logn)
- **Interpretation**: An algorithm with ω(log⁡n)\omega(\log n)ω(logn) time complexity means that its execution time grows faster than logarithmic time with the size of the input nnn.

4.b. **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:

```
plaintext
Copy code
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:

```
plaintext
Copy code
Quick Sort:
1. Choose pivot 'I' (last character) for "VTUBELAGAVI".
2. Partition: "AEGAVIVTUBL".
3. Recursively sort "AEGAVI" and "UBL".
4. Final sorted string: "AAEGILTVUB".
```

Sorted Output:

- **Merge Sort**: "AAEGILTVUB"
- **Quick Sort**: "AAEGILTVUB"

5.a. General Plan for Analyzing Time Efficiency of Recursive Algorithms:

Analyzing 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 analyzing the time efficiency of recursive algorithms:

1. **Identify Recursive Structure**: Understand how the recursive algorithm divides the problem into smaller sub-problems 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.
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:

Here's a simplified version of the recursive algorithm for solving the Tower of Hanoi problem:

```
plaintext
Copy code
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 analyze 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

   - The 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.
   - The +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):
   - By 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.

5.b. Quick Sort Algorithm Explanation:

Quick Sort is a divide-and-conquer sorting algorithm that works as follows:

1. **Choose a Pivot**: Select an element from the array as the pivot. There are different strategies for choosing the pivot; commonly used ones include selecting the first element, the last element, or a random element.
2. **Partitioning**: Rearrange the array so that all elements smaller than the pivot come before it, and all elements greater than the pivot come after it. After partitioning, the pivot is in its correct position.
3. **Recursively Apply**: Recursively apply Quick Sort to the sub-arrays formed by partitioning until the entire array is sorted.
4. **Combine Results**: No further action is needed as the array is already sorted after partitioning.

Tracing the Dataset [1, 1, 9, 9, 5, 5, 6, 6] with Quick Sort:

Let's trace the sorting process step by step for the dataset [1, 1, 9, 9, 5, 5, 6, 6]:

1. **Initial Array**: [1, 1, 9, 9, 5, 5, 6, 6]
2. **Choose Pivot**: Let's choose the last element as the pivot. In this case, the last element is 6.
3. **Partitioning**:
    - Rearrange elements so that all elements less than the pivot (6) come before it, and all elements greater than the pivot come after it.

    After partitioning:

    - [1, 1, 5, 5, 6, 6, 9, 9]
    - The pivot element (6) is now in its correct position between the smaller and larger elements.
4. **Recursive Calls**:
    - Now, recursively apply Quick Sort to the left and right sub-arrays.

    Left sub-array: [1, 1, 5, 5] Right sub-array: [9, 9]

5. **Partitioning and Recursive Calls**:
    - For the left sub-array [1, 1, 5, 5]:
        - Choose the last element (5) as the pivot.
        - Partitioning: After partitioning, the array remains unchanged as [1, 1, 5, 5] since all elements are already in order.
        - No further recursion needed for this sub-array.
    - For the right sub-array [9, 9]:
        - Choose the last element (9) as the pivot.
        - Partitioning: After partitioning, the array remains unchanged as [9, 9] since all elements are already in order.
        - No further recursion needed for this sub-array.
6. **Combine Results**:
    - After all recursive calls and combining results, the sorted array is [1, 1, 5, 5, 6, 6, 9, 9].

Final Sorted Array:

The final sorted array using Quick Sort for the dataset [1, 1, 9, 9, 5, 5, 6, 6] is [1, 1, 5, 5, 6, 6, 9, 9].

Key Points about Quick Sort:

- **Efficiency**: Quick Sort typically has an average-case time complexity of O(nlog⁡n)O(n \log n)O(nlogn) and worst-case complexity of O(n2)O(n^2)O(n2), depending on the choice of pivot.
- **Partitioning**: The partitioning step rearranges elements around a pivot, making Quick Sort an in-place sorting algorithm (with O(log⁡n)O(\log n)O(logn) extra space for recursion).
- **Randomization**: Randomizing the pivot choice or using median-of-three can improve its performance and mitigate worst-case scenarios.

6.a. Analysis:

a. **What does this algorithm compute?**

- The algorithm computes the sum of squares of the first `n` natural numbers. That is, it computes S=12+22+32+…+n2S = 1^2 + 2^2 + 3^2 + \ldots + n^2S=12+22+32+…+n2.

b. **What is its basic operation?**

- The basic operation in this algorithm is the addition `S ← S + i * i`, where `i * i` represents squaring the current value of `i`.

c. **How many times is the basic operation executed?**

- The basic operation `S ← S + i * i` is executed `n` times, once for each iteration of the loop from `i = 1` to `i = n`.

d. **What is the efficiency class of this algorithm?**

- To determine the efficiency class, we consider the total number of basic operations executed. Since the loop runs from `i = 1` to `i = n`, the total number of operations is the sum of squares of the first `n` natural numbers: ∑i=1ni2=n(n+1)(2n+1)6\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}∑i=1ni2=6n(n+1)(2n+1) Therefore, the time complexity of the algorithm `Mystery(n)` is O(n3)O(n^3)O(n3) because the dominant term in the expression n(n+1)(2n+1)6\frac{n(n+1)(2n+1)}{6}6n(n+1)(2n+1) is n3n^3n3.

e. **Suggest an improvement, or a better algorithm altogether, and indicate its efficiency class.**

- An improvement to compute the sum of squares more efficiently can be achieved using the formula:

  S=n(n+1)(2n+1)6S = \frac{n(n + 1)(2n + 1)}{6}S=6n(n+1)(2n+1)

This formula computes the sum of squares directly without needing to iterate through each number up to `n`.

- **Efficiency Class of Improved Algorithm**: The time complexity of computing the sum of squares using the formula is O(1)O(1)O(1). This is constant time complexity because the calculation involves only a fixed number of arithmetic operations regardless of the size of `n`.

6.b. Analysis:

1. **Logarithmic Function log⁡(n)\log(n)log(n)**:
   - The logarithmic function log⁡(n)\log(n)log(n) grows very slowly compared to polynomial functions like n2n^2n2.
   - In Big O notation, log⁡(n)\log(n)log(n) denotes the logarithm base 2 (commonly used in algorithm analysis unless specified otherwise).
2. **Quadratic Function n2n^2n2**:
   - The quadratic function n2n^2n2 grows much faster than log⁡(n)\log(n)log(n).
   - For large values of nnn, n2n^2n2 increases much more rapidly compared to log⁡(n)\log(n)log(n).

Comparison:

- **Behavior as nnn Increases**:
  - log⁡(n)\log(n)log(n):
    - Grows very slowly. For example, log⁡(1000)≈3\log(1000) \approx 3log(1000)≈3 and log⁡(1,000,000)≈6\log(1,000,000) \approx 6log(1,000,000)≈6.
    - It increases logarithmically, meaning the rate of growth diminishes as nnn increases.
  - n2n^2n2:
    - Grows rapidly. For example, 1002=10,000100^2 = 10,0001002=10,000 and 10002=1,000,0001000^2 = 1,000,00010002=1,000,000.
    - It increases quadratically, meaning the rate of growth accelerates as nnn increases.
- **Order of Growth Comparison**:
  - In terms of Big O notation:
    - log⁡(n)\log(n)log(n) has an order of growth O(log⁡n)O(\log n)O(logn).
    - n2n^2n2 has an order of growth O(n2)O(n^2)O(n2).