DAA Assignment No. 2

❖ 2 Marks Questions

Q.1.) Define asymptotic notation. (Remember)

Ans :- Asymptotic notation is a mathematical tool used to describe the limiting behavior of an algorithm's time or space complexity as the input size grows infinitely large. It provides a way to classify algorithms based on their efficiency. The three main types of asymptotic notations are:

1. **Big-O (O)**: Upper bound, representing the worst-case complexity.
2. **Omega (Ω)**: Lower bound, representing the best-case complexity.
3. **Theta (Θ)**: Tight bound, representing the average-case complexity.

Q.2.) What is the importance of measuring input size in algorithm analysis? (Understand)

Ans :- Measuring input size in algorithm analysis is crucial because it directly impacts the algorithm's efficiency in terms of time and space complexity. Here's why it's important:

1. **Performance Estimation** – Understanding input size helps predict how an algorithm will perform as the problem scales, ensuring it can handle large datasets efficiently.
2. **Comparing Algorithms** – It allows fair comparison between different algorithms, helping choose the most optimal one for a given problem.
3. **Resource Management** – Knowing how input size affects performance helps allocate computational resources effectively, such as memory and processing power.
4. **Identifying Bottlenecks** – Helps in detecting performance limitations and improving algorithms to handle larger inputs efficiently.

5. **Real-World Applicability** – Ensures that an algorithm remains practical and usable in real-world applications where large inputs are common.

Q.3.) Differentiate between worst-case, best-case, and average-case complexities. (Understand)

Ans :- The worst-case, best-case, and average-case complexities describe an algorithm's efficiency under different input conditions. Here's how they differ:

1. **Worst-Case Complexity (O)**
    - Represents the maximum time an algorithm takes for any input of size *n*.
    - It provides an upper bound on runtime, ensuring the algorithm will not exceed this limit.
    - Used for performance guarantees, especially in critical systems.
    - Example: In **linear search**, if the element is not present, it takes **O(n)** time.
2. **Best-Case Complexity (Ω)**
    - Represents the minimum time the algorithm takes for the most favorable input.
    - It provides a lower bound but is less useful for real-world analysis.
    - Example: In **linear search**, if the element is found at the first position, it takes **Ω(1)** time.
3. **Average-Case Complexity (Θ)**
    - Represents the expected runtime over all possible inputs.
    - It gives a more realistic measure of an algorithm's efficiency.
    - Example: In **linear search**, if the element is randomly placed, the search takes **Θ(n/2) = Θ(n)** on average.

Q.4.) What is the significance of O-notation in algorithm analysis? (Understand)

Ans :- The **significance of O-notation (Big-O notation) in algorithm analysis** lies in its ability to describe the upper bound of an algorithm's time or space complexity, providing a worst-case performance guarantee. Here's why it is important:

1. **Measures Scalability** – Helps understand how an algorithm behaves as input size increases, ensuring it can handle large datasets efficiently.
2. **Simplifies Complexity Analysis** – Focuses on the dominant term, ignoring constants and lower-order terms, making comparisons easier.
3. **Allows Algorithm Comparison** – Enables fair comparison between different algorithms to determine which is more efficient for large inputs.
4. **Guides Optimization** – Helps developers identify bottlenecks and improve algorithms for better performance.
5. **Ensures Performance Predictability** – Provides a worst-case guarantee, which is essential for real-time and critical systems where performance limits must be known.

Q.5.) State the difference between recursive and non-recursive algorithms. (Remember)

Ans :- The difference between **recursive** and **non-recursive** algorithms is as follows:

Feature	Recursive Algorithm	Non-Recursive Algorithm
Definition	An algorithm that calls itself to solve a smaller subproblem.	An algorithm that solves the problem iteratively without self-calls.
Function Calls	Uses recursive function calls, increasing stack memory usage.	Uses loops and conditionals, avoiding extra function calls.
Memory Usage	Requires extra memory for recursive calls (stack frames).	Uses constant memory, making it more space-efficient.
Complexity	Can be easier to write and understand but may have higher time/memory overhead.	Often more efficient in terms of execution time and memory usage.
Example Use Cases	Tree traversal, Divide and Conquer algorithms (e.g., Merge Sort, Quick Sort).	Iterative solutions like loops (e.g., searching, sorting, iterative Fibonacci).

Example of a Recursive Algorithm (Factorial Calculation)

```cpp
#include <iostream>
using namespace std;

// Recursive function to calculate factorial
int factorial(int n) {
    if (n == 0 || n == 1)
        return 1; // Base case
    return n * factorial(n - 1); // Recursive call
}

int main() {
    int num = 5;
    cout << "Factorial of " << num << " is " << factorial(num) << endl;
    return 0;
}
```

Output:

```
Factorial of 5 is 120
```

Example of a Non-Recursive Algorithm (Factorial Calculation)

```cpp
#include <iostream>
using namespace std;

// Iterative function to calculate factorial
int factorial(int n) {
    int result = 1;
    for (int i = 2; i <= n; i++) {
        result *= i;
    }
    return result;
}

int main() {
    int num = 5;
    cout << "Factorial of " << num << " is " << factorial(num) << endl;
    return 0;
}
```

Output :

```
Factorial of 5 is 120
```

❖ 10 Marks Questions

Q.1.) Explain different units for measuring running time in algorithm analysis. (Understand)

Ans :- **Different Units for Measuring Running Time in Algorithm Analysis**

The running time of an algorithm is measured using various units depending on the level of abstraction. These units help in evaluating and comparing algorithm efficiency.

1. Machine-Dependent Units

These units depend on hardware, processor speed, and system architecture.

a) Actual Time (Seconds, Milliseconds, etc.)

- Measures the **real execution time** taken by an algorithm.
- **Example:** If an algorithm runs in 5 milliseconds (ms), it means it took **5 ms** to complete execution.
- **Limitation:** Varies based on machine specifications, making comparisons difficult.

b) Number of CPU Cycles

- Measures the **number of clock cycles** an algorithm takes to execute.
- **Formula:**

$$\text{Time} = \frac{\text{Number of cycles}}{\text{Clock speed (Hz)}}$$

- **Limitation:** Depends on processor speed, cache memory, and system load.

2. Machine-Independent Units

These units abstract hardware dependencies, making algorithm analysis more general.

a) Basic Operations Count

- Counts the number of fundamental operations (e.g., comparisons, assignments, additions).

- **Example:** In linear search, the worst case requires **n** comparisons.

b) Input Size (n)

- Represents how the execution time grows as the **size of input (n) increases**.

- Common measures:
  - **n (linear complexity)**
  - **n^2 (quadratic complexity)**
  - **log n (logarithmic complexity)**

c) Asymptotic Notation

- Uses mathematical notations to describe growth rate:
  - **Big-O (O)** → Upper bound (worst case)
  - **Big-Omega (Ω)** → Lower bound (best case)
  - **Big-Theta (Θ)** → Tight bound (average case)
- **Example:**
  - **Bubble Sort:** $O(n^2)$ in worst case.
  - **Merge Sort:** $O(n \log n)$ in worst case.

Q.2.) Discuss the different orders of growth with suitable examples. (Apply)

Ans :- **Different Orders of Growth in Algorithm Analysis**

The **order of growth** describes how an algorithm's running time increases as the input size nnn grows. It helps classify algorithms based on their efficiency. Below are the most common growth rates with examples.

1. Constant Time – $O(1)$

- The execution time remains the same regardless of input size.
- **Example: Accessing an element in an array**

```cpp
int getElement(int arr[], int index) {
    return arr[index]; // Always takes constant time
}
```

- **Use case:** Hash table lookups, accessing array elements.

2. Logarithmic Time – $O(\log n)$

- The execution time grows **logarithmically**, meaning it increases slowly as input size grows.
- **Example: Binary Search**

```cpp
int binarySearch(int arr[], int low, int high, int key) {
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == key) return mid;
        else if (arr[mid] < key) low = mid + 1;
        else high = mid - 1;
    }
    return -1;
}
```

- **Use case:** Searching in sorted arrays, tree traversals.

3. Linear Time – $O(n)$

- The execution time grows **proportionally** to the input size.
- **Example: Linear Search**

```cpp
int linearSearch(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) return i;
    }
    return -1;
}
```

- **Use case:** Iterating over arrays, simple loops.

4. Linearithmic Time – $O(n \log n)$

- More efficient than $O(n^2)$, common in divide and conquer algorithms.
- **Example: Merge Sort**

```cpp
void mergeSort(int arr[], int left, int right) {
    if (left >= right) return;
    int mid = left + (right - left) / 2;
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);
    merge(arr, left, mid, right); // Merge step
}
```

- **Use case:** Efficient sorting algorithms (Merge Sort, Heap Sort).

5. Quadratic Time – $O(n^2)$

- Execution time **grows quadratically**, often caused by nested loops.
- **Example: Bubble Sort**

```cpp
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) swap(arr[j], arr[j + 1]);
        }
    }
}
```

- **Use case:** Brute-force algorithms, basic sorting methods.

6. Cubic Time – $O(n^3)$

- Execution time **grows cubically**, often found in algorithms with **three nested loops**.
- **Example: Matrix Multiplication**

```cpp
void matrixMultiply(int A[][N], int B[][N], int C[][N], int N) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            C[i][j] = 0;
            for (int k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

- **Use case:** Computational problems in physics, AI, graphics.

7. Exponential Time – $O(2^n)$

- The execution time **doubles** with each increase in input size.
- **Example: Fibonacci Sequence (Recursive)**

```cpp
int fibonacci(int n) {
    if (n <= 1) return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

- **Use case:** Combinatorial problems, brute-force solutions.

8. Factorial Time – $O(n!)$

- The execution time **grows factorially**, making it impractical for large inputs.
- **Example: Generating all permutations**

```cpp
void permute(string str, int l, int r) {
    if (l == r) cout << str << endl;
    else {
        for (int i = l; i <= r; i++) {
            swap(str[l], str[i]);
            permute(str, l + 1, r);
            swap(str[l], str[i]); // Backtrack
        }
    }
}
```

- **Use case:** Traveling Salesman Problem, brute-force optimization.

Q.3.) Compare and contrast O-notation, Ω-notation, and θ-notation with real-world examples. (Analyze)

Ans :- **Comparison of O-Notation, Ω-Notation, and Θ-Notation**

Asymptotic notations describe the **time complexity** of algorithms by providing upper, lower, and tight bounds. Here's a detailed comparison:

1. Big-O Notation (O-Notation) – Upper Bound (Worst Case)

- Represents the **maximum time an algorithm can take** for an input size nnn.
- Used to analyze **worst-case** performance.
- **Example:** Sorting a randomly ordered array using **Bubble Sort** (O(n2)O(n^2)O(n2)).

Real-World Example:

- **Traffic Jam Estimation:** If a city estimates **at most 2 hours** for a route during rush hour, then **O(2 hours)** represents the **worst case** time.

Example in Code (Linear Search - Worst Case: O(n))

```
int linearSearch(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) return i; // Found
    }
    return -1; // Worst case: Element not found (O(n))
}
```

2. Omega Notation (Ω-Notation) – Lower Bound (Best Case)

- Represents the **minimum time an algorithm will take** in the best case.
- Used to analyze **best-case** performance.
- **Example:** Sorting an **already sorted** array using **Insertion Sort** (Ω(n)Ω(n)Ω(n)).

Real-World Example:

- **Minimum Delivery Time:** If a package **takes at least 30 minutes** for delivery in ideal conditions, then **Ω(30 min)** represents the **best case** scenario.

Example in Code (Linear Search - Best Case: Ω(1))

```
int linearSearch(int arr[], int n, int key) {
    if (arr[0] == key) return 0; // Best case: Found at first position (Ω(1))
    for (int i = 1; i < n; i++) {
        if (arr[i] == key) return i;
    }
    return -1;
}
```

3. Theta Notation (Θ-Notation) – Tight Bound (Average Case)

- Represents **both the upper and lower bounds** (average-case performance).
- It means the algorithm runs in **precisely this complexity**.
- **Example:** Merge Sort always runs in **Θ(nlog⁡n)Θ(n \log n)Θ(nlogn)** regardless of input order.

Real-World Example:

- **Fuel Consumption in a Car:** If a car consistently uses fuel at **Θ(5L per 100 km)**, it means the consumption will be around that amount regardless of driving conditions.

Example in Code (Merge Sort - Always Θ(n log n))

```
void mergeSort(int arr[], int left, int right) {
    if (left >= right) return;
    int mid = left + (right - left) / 2;
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);
    merge(arr, left, mid, right); // Merge step takes Θ(n)
}
```

Q.4.) Perform a mathematical analysis of a recursive algorithm and derive its time complexity. (Apply & Analyze)

Ans :- **Mathematical Analysis of a Recursive Algorithm and Deriving Its Time Complexity**

To analyze the time complexity of a recursive algorithm, we follow these steps:

1. **Define the recurrence relation** based on the recursive calls.
2. **Solve the recurrence relation** using standard methods (Substitution, Recurrence Tree, or Master Theorem).
3. **Derive the final time complexity** in Big-O notation.

Example: Recursive Fibonacci Algorithm

Consider the classic **recursive Fibonacci function**:

```cpp
int fibonacci(int n) {
    if (n <= 1) return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Step 1: Define the Recurrence Relation

- The function makes **two recursive calls** for each input n.
- The base cases ($n = 0, n = 1$) take **O(1)** time.
- The recurrence relation is:

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

Step 2: Solve the Recurrence Relation

Approach 1: Recurrence Tree Method

Expanding the recurrence:

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

$$T(n-1) = T(n-2) + T(n-3) + O(1)$$

$$T(n-2) = T(n-3) + T(n-4) + O(1)$$

This forms an **exponential tree**, where each level doubles the number of calls.

The number of recursive calls follows the Fibonacci sequence, leading to:

$$T(n) = O(2^n)$$

Thus, the **time complexity of recursive Fibonacci** is $O(2^n)$, which is **exponential growth**.

Example 2: Recursive Merge Sort Algorithm

Consider the recursive **Merge Sort** algorithm:

```cpp
void mergeSort(int arr[], int left, int right) {
    if (left >= right) return;
    int mid = left + (right - left) / 2;
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);
    merge(arr, left, mid, right); // Merging step
}
```

Step 1: Define the Recurrence Relation

- Merge Sort **divides the array** into **two halves** at each step.
- Each recursive call sorts **n/2** elements.
- The merging step takes **O(n)** time.
- The recurrence relation is:

$$T(n) = 2T(n/2) + O(n)$$

Step 2: Solve the Recurrence Relation

Approach 1: Recurrence Tree

Expanding the recurrence:

$$T(n) = 2T(n/2) + O(n)$$

$$T(n/2) = 2T(n/4) + O(n/2)$$

$$T(n/4) = 2T(n/8) + O(n/4)$$

This process continues until we reach **T(1) = O(1)**.

- The depth of the tree is $\log_2(n)$.
- Each level performs **O(n)** work.
- The total time complexity is:

$$O(n \log n)$$

Thus, the **time complexity of Merge Sort** is $O(n \log n)$.

Q.5.) Solve the recurrence relation and find the time complexity using Master Theorem.

a. T(n)=7T(n/2)+18n2

b. T(n)= 2T(n/2)n^3

Ans :-

Master Theorem Overview

The **Master Theorem** is a tool used to solve recurrence relations of the form:

$$T(n) = aT(n/b) + f(n)$$

Where:

- $a \geq 1$ is the number of subproblems,
- $b > 1$ is the factor by which the problem size is reduced in each subproblem,
- $f(n)$ is the non-recursive work done at each level (the merging step).

The **Master Theorem** compares $f(n)$ with $n^{\log_b a}$ and provides three cases:

1. If $f(n) = O(n^d)$ where $d < \log_b a$, then $T(n) = O(n^{\log_b a})$.
2. If $f(n) = \Theta(n^d)$ where $d = \log_b a$, then $T(n) = \Theta(n^d \log n)$.
3. If $f(n) = \Omega(n^d)$ where $d > \log_b a$, and if $af(n/b) \leq kf(n)$ for some constant $k < 1$, then $T(n) = \Theta(f(n))$.

Problem (a): $T(n) = 7T(n/2) + 18n^2$

Step 1: Identify Parameters

We have the recurrence in the form $T(n) = aT(n/b) + f(n)$:

- $a = 7$
- $b = 2$
- $f(n) = 18n^2$

Step 2: Calculate $\log_b a$

$$\log_b a = \log_2 7$$

Using logarithmic approximation:

$$\log_2 7 \approx 2.807$$

Step 3: Compare $f(n)$ with $n^{\log_b a}$

- $f(n) = 18n^2$ is $O(n^2)$.
- $n^{\log_b a} = n^{2.807}$.

Here, $f(n)$ is $O(n^2)$, and we observe that $2 < 2.807$. So, we fall into **Case 1** of the Master Theorem, where $f(n) = O(n^d)$ and $d < \log_b a$.

Step 4: Apply the Master Theorem

Since $d = 2$ and $\log_b a \approx 2.807$, the recurrence falls under **Case 1**, so the time complexity is:

$$T(n) = O(n^{\log_b a}) = O(n^{2.807})$$

Problem (b): $T(n) = 2T(n/2) + n^3$

Step 1: Identify Parameters

We have the recurrence in the form $T(n) = aT(n/b) + f(n)$:

- $a = 2$
- $b = 2$
- $f(n) = n^3$

Step 2: Calculate $\log_b a$

$$\log_b a = \log_2 2 = 1$$

Step 3: Compare $f(n)$ with $n^{\log_b a}$

- $f(n) = n^3$ is $O(n^3)$.
- $n^{\log_b a} = n^1$.

Here, $f(n)$ is $O(n^3)$, and $3 > 1$. So, we fall into **Case 3** of the Master Theorem, where $f(n) = \Omega(n^d)$ and $d > \log_b a$.

Step 4: Apply the Master Theorem

Since $f(n) = n^3$ and $d = 3$, which is greater than $\log_b a = 1$, we apply **Case 3**. Therefore, the time complexity is:

$$T(n) = \Theta(f(n)) = \Theta(n^3)$$