DAA Unit 1

1. Algorithm

- **Definition**: An algorithm is a step-by-step procedure or a set of instructions designed to perform a specific task or solve a problem.
- **Characteristics**:
  - **Finiteness**: The algorithm must terminate after a finite number of steps.
  - **Definiteness**: Each step must be precisely defined.
  - **Input**: Should have well-defined inputs.
  - **Output**: Should produce a clearly defined output.
  - **Effectiveness**: Each step must be simple enough to be performed in a finite time.

---

2. Performance Analysis

- **Objective**: To evaluate how efficiently an algorithm performs with respect to resource usage, mainly **time** and **space**.
- Two key measures of performance are:

2.1 Space Complexity

- **Definition**: The amount of memory required by an algorithm to run to completion.
- **Components**:
  - **Fixed part**: The memory space required to store the inputs, outputs, and constants.
  - **Variable part**: Memory required for dynamic allocation during execution (e.g., function call stacks, recursion).

2.2 Time Complexity

- **Definition**: The amount of time taken by an algorithm to run as a function of the input size, denoted by n.
- **Components**:
  - **Best-case**: The minimum time an algorithm takes for any input.
  - **Worst-case**: The maximum time an algorithm takes for any input.
  - **Average-case**: The expected time over all possible inputs.

---

3. Asymptotic Notations

Asymptotic notations provide a way to express the growth of an algorithm's time or space complexity as the input size tends to infinity.

3.1 Big O Notation (O)

- **Definition**: Describes the **upper bound** of the algorithm's growth rate. It represents the worst-case scenario.
- **Example**: If an algorithm has time complexity $O(n^2)$, its running time increases quadratically as the input size grows.

3.2 Omega Notation (Ω)

- **Definition**: Describes the **lower bound** of the algorithm's growth rate. It represents the best-case scenario.
- **Example**: An algorithm with time complexity $\Omega(n)$ will take at least linear time in the best case.

3.3 Theta Notation (Θ)

- **Definition**: Describes the **tight bound** of the algorithm's growth rate. It applies when the algorithm has the same growth rate in both the best and worst cases.
- **Example**: If an algorithm has time complexity $\Theta(n^2)$, it will always have quadratic time complexity, regardless of the input.

3.4 Little O Notation (o)

- **Definition**: Describes a growth rate that is strictly **less than** the upper bound given by Big O. It indicates a loose upper bound.
- **Example**: If the time complexity is $o(n^2)$, the algorithm's growth rate is strictly smaller than n^2.

---

1. Divide and Conquer: General Method

- **Definition**:
  - A powerful algorithm design paradigm where a problem is broken down into smaller sub-problems, each sub-problem is solved recursively, and then their solutions are combined to solve the original problem.
- **Steps**:
  1. **Divide**: Break the problem into smaller, independent sub-problems.
  2. **Conquer**: Solve the sub-problems recursively. If the sub-problem is small enough, solve it directly.
  3. **Combine**: Merge the solutions of the sub-problems to form the solution to the original problem.

- **Advantages**:
  - Makes complex problems more manageable by breaking them into simpler parts.
  - Recursive solutions often easier to conceptualize and implement.
- **Time Complexity**: Often characterized using recurrence relations. For example, the time complexity of many divide and conquer algorithms can be expressed using the recurrence $T(n) = aT(n/b) + O(n^d)$, which can be solved using the **Master Theorem**.

---

2. Applications of Divide and Conquer

2.1 Binary Search

- **Problem**: Searching for an element in a sorted array.
- **Divide and Conquer Approach**:
  - **Divide**: Split the array into two halves.
  - **Conquer**: Recursively search in the half where the target might exist.
  - **Combine**: No need to combine, as only one half is relevant at any point.
- **Time Complexity**: $O(\log n)$, as the problem size is halved at each step.

Pseudo-algorithm:

```
BinarySearch(arr, target, low, high)
  if low > high
    return -1  // target not found
  mid = (low + high) // 2
  if arr[mid] == target
    return mid  // target found
  else if arr[mid] > target
    return BinarySearch(arr, target, low, mid - 1)
  else
    return BinarySearch(arr, target, mid + 1, high)
```

2.2 Quick Sort

- **Problem**: Sorting an array of elements.
- **Divide and Conquer Approach**:

    - **Divide**: Pick a pivot element and partition the array into two subarrays: elements less than the pivot and elements greater than the pivot.
    - **Conquer**: Recursively quicksort the two subarrays.
    - **Combine**: The sorted subarrays are concatenated with the pivot in the middle.
- **Time Complexity**:
    - **Average case**: $O(n \log n)$.
    - **Worst case**: $O(n^2)$ (when the pivot is always the smallest or largest element).

Pseudo-algorithm:

```
QuickSort(arr, low, high)
  if low < high
    pivot = Partition(arr, low, high)
    QuickSort(arr, low, pivot - 1)  // Recursively sort left half
    QuickSort(arr, pivot + 1, high)  // Recursively sort right half

Partition(arr, low, high)
  pivot = arr[high]  // Choose the last element as pivot
  i = low - 1  // Index for the smaller element
  for j = low to high - 1
    if arr[j] <= pivot
      i = i + 1
      Swap(arr[i], arr[j])
  Swap(arr[i + 1], arr[high])
  return i + 1
```

2.3 Merge Sort

- **Problem**: Sorting an array of elements.
- **Divide and Conquer Approach**:
    - **Divide**: Split the array into two halves.
    - **Conquer**: Recursively sort the two halves.
    - **Combine**: Merge the two sorted halves to produce the final sorted array.
- **Time Complexity**: $O(n \log n)$ in all cases, since the array is always split in half and merging takes linear time.

Pseudo-algorithm:

```
MergeSort(arr, low, high)
  if low < high
    mid = (low + high) // 2
    MergeSort(arr, low, mid)  // Recursively sort the left half
    MergeSort(arr, mid + 1, high)  // Recursively sort the right
half
    Merge(arr, low, mid, high)  // Merge the sorted halves

Merge(arr, low, mid, high)
  n1 = mid - low + 1
  n2 = high - mid
  Create temp arrays L[1..n1] and R[1..n2]
  Copy data to temp arrays L[] and R[]
  Merge the temp arrays back into arr[low..high]
```

2.4 Strassen's Matrix Multiplication

- **Problem**: Multiplying two matrices.
- **Divide and Conquer Approach**:
  - **Divide**: Break the matrices into 4 smaller sub-matrices.
  - **Conquer**: Use 7 recursive matrix multiplications (instead of 8 in standard methods).
  - **Combine**: Use addition and subtraction to combine the sub-matrices.
- **Time Complexity**: $O(n^{\log_2 7}) \approx O(n^{2.81})$, faster than the traditional $O(n^3)$ for large matrices.

Pseudo-algorithm:

```
Strassen(A, B)
  if A and B are 1x1 matrices
    return A * B
  else
    Divide A and B into 4 sub-matrices
    M1 = Strassen(A11 + A22, B11 + B22)
    M2 = Strassen(A21 + A22, B11)
    M3 = Strassen(A11, B12 - B22)
    M4 = Strassen(A22, B21 - B11)
    M5 = Strassen(A11 + A12, B22)
    M6 = Strassen(A21 - A11, B11 + B12)
    M7 = Strassen(A12 - A22, B21 + B22)

    Combine the results to form matrix C:
    C11 = M1 + M4 - M5 + M7
    C12 = M3 + M5
    C21 = M2 + M4
    C22 = M1 - M2 + M3 + M6
    return C
```
