## Time Complexity

**Time complexity** gives an estimation of the running time of an algorithm. It represents how the runtime of an algorithm grows as the size of the input increases. This helps in predicting the performance and scalability of an algorithm.

**Common Time Complexities:**

1. **O(1) - Constant Time :**
   - The runtime does not depend on the input size.
   - Example: Accessing a specific element in an array.

2. **O(log n) - Logarithmic Time :**
   - The runtime grows logarithmically as the input size increases.
   - Example: Binary search in a sorted array.

3. **O(n) - Linear Time :**
   - The runtime grows linearly with the input size.
   - Example: Iterating through an array.

4. **O(n log n) - Linearithmic Time :**
   - The runtime grows faster than linear but slower than quadratic.
   - Example: Efficient sorting algorithms like merge sort and quicksort.

5. **O(n^2) - Quadratic Time :**
   - The runtime grows quadratically with the input size.
   - Example: Simple sorting algorithms like bubble sort and insertion sort.

6. **O(2^n) - Exponential Time :**
   - The runtime grows exponentially with the input size.
   - Example: Solving the traveling salesman problem using brute force.

7. **O(n!) - Factorial Time :**
   - The runtime grows factorially with the input size.
   - Example: Solving the traveling salesman problem with all permutations.

## Space Complexity

**Space complexity** measures the amount of memory an algorithm uses relative to the input size. It helps in understanding the memory requirements and optimizing them.

**Common Space Complexities:**

1. **O(1) - Constant Space :**
   - The memory usage does not depend on the input size.
   - Example: Using a fixed number of variables.

2. **O(n) - Linear Space :**
   - The memory usage grows linearly with the input size.
   - Example: Storing input elements in an array.

3. **O(log n) - Logarithmic Space :**
   - The memory usage grows logarithmically with the input size.
   - Example: Recursive algorithms like binary search, which use stack space proportional to the logarithm of the input size.

4. **O(n^2) - Quadratic Space :**
   - The memory usage grows quadratically with the input size.
   - Example: Using a 2D array to store a matrix of size n x n.

**Examples**
**Time Complexity Example:**
Consider a function that sums all elements in an array:

```
int sum(int arr[], int n) {
    int total = 0;
    for(int i = 0; i < n; i++) {
        total += arr[i];
    }
    return total;
}
```

- **Time Complexity : O(n) because the function iterates over all n elements in the array.**

**Space Complexity Example**:
Consider a function that creates a new array with doubled values:

```
int* doubleArray(int arr[], int n) {
    int* doubled = new int[n];
```

```
    for(int i = 0; i < n; i++) {
        doubled[i] = arr[i] * 2;
    }
    return doubled;
}
```

- **Space Complexity : O(n) because it allocates a new array of size n.**

## Approaches to Solve a DSA Problem

1. **Brute Force Approach :**
   - **Definition :** A straightforward method of solving a problem by trying all possible solutions.
   - **Example :** For finding the maximum element in an array, the brute force approach would be to iterate through all elements and keep track of the maximum.
   - **Time Complexity :** Often inefficient with higher time complexity, such as *O(n^2)* or worse.
   - **Space Complexity :** Generally uses more memory, depending on the method used.

2. **Optimizing the Solution :**
   - **Definition :** Improving the brute force solution by eliminating unnecessary work or using better data structures.
   - **Example :** Using a hash map to find two numbers in an array that sum up to a given value, instead of checking all pairs.
   - **Time Complexity :** Reduced compared to brute force, such as reducing *O(n^2)* to *O(n log n)* or O(n).
   - **Space Complexity :** Can vary; sometimes more space is used for faster computation.

3. **Final Efficient Solution :**
   - **Definition :** The most optimized version of the algorithm, often using advanced techniques or data structures.
   - **Example :** Using a sliding window or dynamic programming for problems like longest substring without repeating characters or finding the maximum sum subarray.
   - **Time Complexity:** As low as possible, often O(n) for linear solutions.
   - **Space Complexity:** Balanced to ensure minimal memory usage while maintaining efficiency.

### Example: Finding the Maximum Subarray Sum (Kadane's Algorithm)

***Problem: Given an array of integers, find the contiguous subarray (containing at least one number) which has the largest sum.***

**Brute Force Approach:**
- **Idea:** Compute the sum of all possible subarrays and track the maximum.
- **Time Complexity:** O(n^3)
- **Space Complexity:** O(1)
- **Code:**

```
    int maxSubArraySum(int arr[], int n) {
     int max_sum = INT_MIN;
     for (int i = 0; i < n; i++) {
         for (int j = i; j < n; j++) {
             int sum = 0;
             for (int k = i; k <= j; k++) {
                 sum += arr[k];
             }
             max_sum = max(max_sum, sum);
         }
     }
     return max_sum;
 }
```

**Optimized Approach:**
- **Idea:** Use prefix sums to reduce the number of sum calculations.
- **Time Complexity:** O(n^2)
- **Space Complexity:** O(n) (for prefix sums)
- **Code:**

```
 int maxSubArraySum(int arr[], int n) {
     int max_sum = INT_MIN;
     int prefix_sum[n + 1];
     prefix_sum[0] = 0;
     for (int i = 0; i < n; i++) {
         prefix_sum[i + 1] = prefix_sum[i] + arr[i];
     }
     for (int i = 0; i < n; i++) {
         for (int j = i; j < n; j++) {
```

```
            int sum = prefix_sum[j + 1] - prefix_sum[i];
            max_sum = max(max_sum, sum);
        }
    }
    return max_sum;
}
```

**Final Efficient Solution (Kadane's Algorithm):**
- **Idea:** Use dynamic programming to keep track of the maximum subarray ending at each position.
- **Time Complexity:** O(n)
- **Space Complexity:** O(1)
- **Code:**

```
int maxSubArraySum(int arr[], int n) {
    int max_sum = arr[0];
    int current_sum = arr[0];
    for (int i = 1; i < n; i++) {
        current_sum = max(arr[i], current_sum + arr[i]);
        max_sum = max(max_sum, current_sum);
    }
    return max_sum;
}
```

**Big O Notation (Ob)**
- **Definition:** Describes an upper bound on the time complexity of an algorithm. It gives the worst-case scenario.
- **Purpose:** To ensure the algorithm won't run any slower than the given complexity.
- **Example:** If an algorithm runs in O(n^2) time, it means that, in the worst case, the running time will grow at most quadratically with the input size.

**Omega Notation (Ω)**
- **Definition:** Describes a lower bound on the time complexity of an algorithm. It gives the best-case scenario.
- **Purpose:** To ensure that the algorithm runs at least as fast as the given complexity.
- **Example:** If an algorithm runs in Ω(n) time, it means that, in the best case, the running time will grow at least linearly with the input size.

**Theta Notation (Θ)**

- **Definition:** Describes a tight bound on the time complexity of an algorithm. It combines both the upper and lower bounds, indicating the algorithm's time complexity for all input sizes.
- **Purpose:** To precisely define the running time of an algorithm.
- **Example:** If an algorithm runs in Θ(n log n) time, it means that the running time will grow logarithmically with the input size and no faster or slower.

**Relationships Among the Notations**
- **Big O (O): f(n) = O(g(n))** implies that for large input sizes, f(n) will not exceed g(n) up to a constant factor.
- **Omega (Ω): f(n) = Ω(g(n))** implies that for large input sizes, f(n) will not be less than g(n) up to a constant factor.
- **Theta (Θ): f(n) = Θ(g(n))**) implies that for large input sizes, f(n) will be both O(g(n)) and Ω(g(n)), meaning it grows asymptotically the same as g(n).

**Examples**

**Big O Notation Example**
```
void exampleFunction(int n) {
    for(int i = 0; i < n; i++) {
        // Some constant time operations
    }
}
```

- **Time Complexity:** O(n) because the loop runs `n` times, and the operations inside the loop are constant time.

**Omega Notation Example**
```
void exampleFunction(int n) {
    if (n % 2 == 0) {
        // Some constant time operations
    } else {
        for(int i = 0; i < n; i++) {
            // Some constant time operations
        }
    }
}
```

- **Time Complexity:** Ω(1) because in the best case (when `n` is even), the function runs in constant time.

**Theta Notation Example**

```
void exampleFunction(int n) {
    for(int i = 0; i < n; i++) {
        // Some constant time operations
    }
}
```

- **Time Complexity:** Θ(n) because the loop runs `n` times, and this is both the upper and lower bound of the time complexity.

**Space Complexity Analysis**

Similar to time complexity, we use Big O, Omega, and Theta notations to describe space complexity.

**Big O Space Complexity Example**

```
void exampleFunction(int n) {
    int* arr = new int[n];
    // Some operations
    delete[] arr;
}
```

- **Space Complexity:** O(n) because an array of size `n` is allocated.

**Omega Space Complexity Example**

```
void exampleFunction(int n) {
    if (n > 0) {
        int* arr = new int[1];
        // Some operations
        delete[] arr;
    }
}
```

- **Space Complexity:** Ω(1) because in the best case, a constant amount of space is used.

**Theta Space Complexity Example**

```
void exampleFunction(int n) {
    int* arr = new int[n];
    // Some operations
    delete[] arr;
}
```

- **Space Complexity:** Θ(n) because an array of size `n` is allocated, which is both the upper and lower bound of the space complexity.

**Summary**

- **Big O (O):** Upper bound on complexity; worst-case scenario.
- **Omega (Ω):** Lower bound on complexity; best-case scenario.
- **Theta (Θ):** Tight bound on complexity; average-case scenario.

## Sorting Algorithms

| Algorithm | Θ (Average Case) | O (Worst Case) | Ω (Best Case) | Space Complexity |
|---|---|---|---|---|
| **Bubble sort** | **Θ(n^2)** | **O(n^2)** | **Ω(n)** | **O(1)** |
| **Selection sort** | **Θ(n^2)** | **O(n^2)** | **Ω(n^2)** | **O(1)** |
| **Insertion sort** | **Θ(n^2)** | **O(n^2)** | **Ω(n)** | **O(1)** |
| **Quick sort** | **Θ(n log n)** | **O(n^2)** | **Ω(n log n)** | **O(log n)** |
| **Merge sort** | **Θ(n log n)** | **O(n log n)** | **Ω(n log n)** | **O(n)** |

## Searching Algorithms

| Algorithm | Θ (Average Case) | O (Worst Case) | Ω (Best Case) | Space Complexity |
|---|---|---|---|---|
| **Linear Search** | **Θ(n)** | **O(n)** | **Ω(1)** | **O(1)** |
| **Binary Search** | **Θ(log n)** | **O(log n)** | **Ω(1)** | **O(1)** |