Time Complexity

Definition

Time complexity is a computational concept that quantifies the amount of time an algorithm takes to execute as a function of the size of the input (denoted as nn). It provides a way to estimate the efficiency of an algorithm, focusing on its growth rate rather than absolute execution time.

Explanation with Example

Why Time Complexity?
Rather than measuring execution time directly (which depends on the machine and environment), time complexity expresses performance in a machine-independent manner, using asymptotic notations like OO, Ω\Omega, and Θ\Theta.

Example Problem: Sum all elements in an array.

Input: A = [1, 2, 3, 4, 5]

Output: Sum = 15

Steps:

1. Initialize sum = 0.
2. Loop through each element of the array and add it to sum.

Analysis

- The loop iterates through nn elements, performing one addition per element.
- The total operations grow linearly with nn.

Time Complexity: O(n)O(n), where nn is the size of the array.

Common Time Complexities

1. **Constant Time (O(1)O(1))**:
   - Execution time is constant regardless of input size.
   - Example: Accessing an element in an array by index.
2. **Linear Time (O(n)O(n))**:
   - Execution time grows proportionally to input size.
   - Example: Summing elements in an array.
3. **Logarithmic Time (O(log n)O(\log n))**:
   - Execution time grows logarithmically with input size.
   - Example: Binary search.
4. **Quadratic Time (O(n2)O(n^2))**:

    - Execution time grows quadratically with input size.
    - Example: Nested loops, such as in Bubble Sort.
5. **Exponential Time (O(2n)O(2^n))**:
    - Execution time doubles with each additional input element.
    - Example: Solving the Traveling Salesman Problem using brute force.

Algorithm Example: Linear Search

Algorithm: Find an element in an unsorted array.

1. Start from the first element.
2. Compare it with the target.
3. Repeat until the target is found or the array ends.

C Code:

```
#include <stdio.h>


int linearSearch(int arr[], int n, int target) {
   for (int i = 0; i < n; i++) { // Line 1: Iteration (n times)
      if (arr[i] == target) {  // Line 2: Comparison (n times in worst case)
         return i;           // Line 3: Return
      }
   }
   return -1; // Element not found
}


int main() {
   int arr[] = {5, 3, 8, 6, 1};
   int n = sizeof(arr) / sizeof(arr[0]);
   int target = 6;
   int result = linearSearch(arr, n, target);
   if (result != -1)
      printf("Element found at index: %d\n", result);
   else
```

```
        printf("Element not found.\n");
    return 0;
}
```

Time Complexity Analysis

- **Best Case**: O(1)O(1) (First element is the target).
- **Worst Case**: O(n)O(n) (Target not present, or at the end).
- **Average Case**: O(n)O(n) (Target is somewhere in the middle).

Space Complexity

Definition

Space complexity measures the total amount of memory an algorithm uses in terms of:

1. **Fixed Part**: Memory used for variables, constants, and program instructions.
2. **Variable Part**: Memory required during execution, depending on the input size nn.

It is expressed as a function of the input size nn, and includes auxiliary space (temporary extra space) and input space (memory to hold inputs).

Explanation with Example

Why Space Complexity?
Efficient use of memory is critical, especially in environments with limited resources, such as embedded systems or large-scale data processing.

Example Problem: Reversing an array.

Input: A = [1, 2, 3, 4, 5]

Output: A = [5, 4, 3, 2, 1]

Method 1 (In-Place): Use two pointers to swap elements.

- Space Complexity: O(1)O(1) (No extra memory used).

Method 2 (Using Extra Array): Create a new array to store reversed elements.

- Space Complexity: O(n)O(n) (Memory proportional to input size).

Algorithm Example: In-Place Array Reversal

Algorithm:

1. Initialize two pointers, one at the start and one at the end.
2. Swap elements at these pointers.
3. Move pointers inward until they meet.

C Code:

```
#include <stdio.h>

void reverseArray(int arr[], int n) {
   int start = 0, end = n - 1;
   while (start < end) {
      int temp = arr[start]; // Swap elements
      arr[start] = arr[end];
      arr[end] = temp;
      start++;
      end--;
   }
}

int main() {
   int arr[] = {1, 2, 3, 4, 5};
   int n = sizeof(arr) / sizeof(arr[0]);
   reverseArray(arr, n);
   printf("Reversed Array: ");
   for (int i = 0; i < n; i++) {
      printf("%d ", arr[i]);
   }
   return 0;
}
```

Space Complexity Analysis

- **Input Space**: O(n)O(n) (Array of size nn).
- **Auxiliary Space**: O(1)O(1) (Single temporary variable temp used for swapping).

- **Total Space Complexity**: O(n)O(n) (Dominated by input).

Common Space Complexities

1. **Constant Space (O(1)O(1))**: No additional memory is required.
    - Example: In-place algorithms like array reversal.
2. **Linear Space (O(n)O(n))**: Memory grows linearly with input size.
    - Example: Storing results in an auxiliary array.
3. **Logarithmic Space (O(log⁡n)O(\log n))**: Used in algorithms like recursive binary search.
    - Example: Space required by recursion stack.
4. **Quadratic Space (O(n2)O(n^2))**: Used for large multidimensional data.
    - Example: Adjacency matrix in graph representation.

Comparison of Time and Space Complexity

- Time complexity measures *speed*, while space complexity measures *memory*.
- Optimization often involves a trade-off, known as the **time-space trade-off**.

Real-World Application

1. **Embedded Systems**: Limited memory resources require O(1)O(1) space algorithms.
2. **Big Data Processing**: Large datasets demand efficient memory usage.
3. **Cloud Computing**: Space-efficient algorithms reduce cost in pay-per-use storage systems.

---

1. Fundamentals of Algorithm (Line Count, Operation Count)

Definition

An algorithm is a systematic, logical approach to solving a computational problem, written in terms of well-defined steps. Its efficiency is evaluated by measuring the resource utilization, primarily **time complexity** (operations performed) and **space complexity** (memory used).

Line Count measures the number of lines contributing to the execution of the algorithm.
Operation Count tracks the number of computational steps, such as assignments, comparisons, or arithmetic operations.

Explanation with Example

Problem: Find the maximum element in an array.

Input: A = [4, 7, 1, 9, 3]
Output: Max = 9

Steps:

1. Assume the first element as the maximum (max = A[0]).
2. Compare max with each subsequent element.
3. Update max if a larger element is found.

Line Count: Every significant instruction counts as a line, e.g., initialization, iteration, comparison.

Operation Count:

- Comparison: Performed once for each element after the first.
- Assignment: Performed when max is updated.

Algorithm

1. Initialize max to the first element of the array.
2. Loop through the array from the second element to the last.
    - Compare the current element with max.
    - If the current element is greater, update max.
3. Return max.

C Code

```
#include <stdio.h>


int findMax(int arr[], int n) {
   int max = arr[0]; // Line 1: Initialization
   for (int i = 1; i < n; i++) { // Line 2: Iteration
      if (arr[i] > max) { // Line 3: Comparison
         max = arr[i];   // Line 4: Assignment
      }
   }
   return max; // Line 5: Return
}
```

```
int main() {
    int arr[] = {4, 7, 1, 9, 3};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Maximum element: %d\n", findMax(arr, n));
    return 0;
}
```

Analysis

Line Count:

- Line 1: Initialization.
- Line 2: Loop.
- Line 3: Comparison (inside loop).
- Line 4: Assignment (conditional inside loop).
- Line 5: Return.

Operation Count:
For an array of size n:

- Comparisons: (n-1)
- Assignments: At most (n-1) in the worst case.

Time Complexity: O(n) (Linear Time).
Space Complexity: O(1) (Constant Space).

Real-World Application

- **Data Analytics**: Identifying maximum sales in a dataset.
- **Game Development**: Finding the highest score.
- **IoT Systems**: Selecting the maximum sensor reading in real-time monitoring.
