**Day 1 – Introduction & Sorting Basics**

# Analysis of Algorithms

**Analysis of Algorithms** is a fundamental aspect of computer science that involves evaluating performance of algorithms and programs(check the efficiency of program). Efficiency is measured in terms of **time**and**space**.

### **Why Do We Analyze Algorithms?**

There can be multiple algorithms to solve the same problem.  
We analyze algorithms to answer questions like:

* Which algorithm is **faster**?
* Which algorithm uses **less memory**?
* Which algorithm is **better for large inputs**?  
  Goal: To choose the most **efficient** algorithm.

# **Problem 1**: **Given a number n, find the sum of first n natural numbers (1 + 2 + 3 + … + n).**

# **Approach 1: Iterative Method (Loop)**

int sum = 0;

for(int i = 1; i <= n; i++) {

sum += i;

}

# **Approach 2: Formula Method (Mathematical)**



int sum = (n \* (n + 1)) / 2;

# **Approach 3: Recursion**

int sumRec(int n) {

if (n == 0) return 0;

return n + sumRec(n - 1);

}

# **Problem 2:** Given a number n, find its factorial (n! = n × (n-1) × … × 1).

### Approach 1: Iterative Method (Loop)

int fact = 1;

for(int i = 1; i <= n; i++) {

fact \*= i;

}

### Approach 2: Recursion

int factorial(int n) {

if (n == 0 || n == 1) return 1;

return n \* factorial(n - 1);

}

# **Problem 3:** Print the first n terms of the Fibonacci series (0, 1, 1, 2, 3, 5, …).

### Approach 1: Iterative Method (Loop)

int a = 0, b = 1, next;

for(int i = 1; i <= n; i++) {

printf("%d ", a);

next = a + b;

a = b;

b = next;

}

### Approach 2: Recursion

int fib(int n) {

if (n == 0) return 0;

if (n == 1) return 1;

return fib(n - 1) + fib(n - 2);

}

**Analysis:**

* Iterative → O(n) time, O(1) space Best practical
* Recursive → O(2^n) time, O(n) space (inefficient for large n)
* Formula → O(1) time, but floating-point precision issues

### ****Types of Analysis of algorithm****

**Asymptotic Notation**   
Asymptotic Notation is a **mathematical tool (concept)** used in computer science to describe the **time complexity** and **space complexity** of (programs) algorithms.

**Types of Asymptotic Notations**



| **Notation** | **Symbol** | **Meaning** | **Goal** |
| --- | --- | --- | --- |
| Big O | O() | Worst case | Upper Bound |
|  |  |  |  |
| Omega | Ω() | Best case | Lower Bound |
| Theta | Θ() | Average/Exact case | Tight Bound |

* O(1) → Constant time
* O(n) → Linear time
* O(n^2) → Quadratic time
* O(log n) → Logarithmic time etc.

**1. Best Case (Ω Notation)**

* **Definition:** The minimum time taken by an Algorithm for the most favorable input.
* It is the **lower bound(**minimum time**)** of execution time.
* **Example:** Linear Search: If the element is at the **first index**, only 1 comparison → Ω(1).
* **Why useful?**
  + Shows the fastest the Algorithm can possibly run.
  + But not reliable in practice (inputs may not always be favorable).

**2. Worst Case (O Notation)**

* **Definition:** The maximum time taken by an Algorithm for the most unfavorable input.
* It is the **upper bound(**maximum time**)** of execution time.
* **Example:** Linear Search: If the element is **last** or not present at all → requires n comparisons → O(n).
* **Why useful?**
  + Guarantees performance **in all cases**.
  + Commonly used in competitive programming & interviews.

**3. Average Case (Θ Notation)**

* **Definition:** The expected time taken by the Algorithm for a **random input** distribution.
* It gives a **tight bound** (both upper & lower).
* **Example:**
  + Linear Search:
    - Best Case: 1 comparison.
    - Worst Case: n comparisons.
    - Average Case: Element found in middle position ≈ n/2 comparisons → Θ(n).
* **Why useful?**
  + Reflects **practical performance** when input is neither best nor worst.

**Time Complexity  
Time Complexity** is a measure of the amount of time an algorithm takes to run, depending on the size of the input n   
(num of step or num of operations). The **time taken by an Algorithm** depends on the **number of operations** it performs, not the actual seconds/minutes (since that depends on CPU, compiler, etc.).

**Execution Time vs Time Complexity**

| **Aspect** | **Execution Time** | **Time Complexity** |
| --- | --- | --- |
| **Definition** | The **actual running time** of a program measured in seconds, milliseconds, etc. | A **mathematical measure** of how the running time of an Algorithm grows with input size n. |
| **Depends on** | Hardware (CPU speed, RAM), compiler, operating system, programming language, machine load. | The **Algorithm’s design** (logic, number of operations). |
| **Unit** | Seconds, milliseconds, nanoseconds. | Asymptotic notation: **O, Ω, Θ**. |
| **Example (Linear Search, n=1,000,000)** | Might take **50 ms** on a fast computer, **200 ms** on a slow one. | Always described as **O(n)** regardless of machine. |
| **Consistency** | Varies from one system to another. | Same across all systems (machine-independent). |
| **Purpose** | Used for **real performance testing**. | Used for **theoretical analysis & comparison** of Algorithms. |
| **Nature** | Practical, system-dependent. | Abstract, system-independent. |

**How to Calculate Time Complexity?**

1. **Look at loops** – How many times does each loop run?
2. **Nested loops** – Multiply their ranges.
3. **Function calls** – Consider recursion depth.
4. **Ignore constants** – Only highest-order term matters.
5. **Ignore hardware-specific operations** – Only analyze based on input size.

**Example 1: Simple Loop (Linear)**

for (int i = 1; i < =n; i++) {

printf("%d ", i);

}

Runs n times  
Time Complexity: **O(n)**

**Example 2: Nested Loop (Quadratic)**

for (int i = 1; i < =n; i++) {

for (int j = 1; j < =n; j++) {

printf("%d ", i + j);

}

}

Outer loop runs n times  
Inner loop runs n times per outer iteration  
Time Complexity: **O(n²)**

**Example 3: Triple Nested Loop**

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

for (int k = 0; k < n; k++) {

printf("Work\n");

}

}

}

Time Complexity: **O(n³)**

**Example 4: Two Separate Loops**

for (int i = 1; i < =n; i++) {

printf("%d ", i);

}

for (int j = 1; j < =n; j++) {

printf("%d ", j);

}

Time Complexity: O(n + n) = **O(2n) ~O(n) ignore multiple factor**

**Example 5 (i): Logarithmic Loop**

Whenever you see a loop where i is **multiplied/divided** by a number every iteration (like i = i\*2 or i = i/2), think of **logarithmic complexity**.

for (int i = 1; i < n; i = i \* 2) {

printf("%d ", i);

}

* This means the value of i **doubles** every time the loop runs.
* So the sequence of i is:
* i = 1, 2, 4, 8, 16, 32, 64, ...

// n=32

int n = 32;

for (int i = 1; i < n; i = i \* 2) {

printf("%d ", i);

}

| **Iteration** | **i value** |
| --- | --- |
| 1 | 1 |
| 2 | 2 |
| 3 | 4 |
| 4 | 8 |
| 5 | 16 |
| 6 | 32 (loop exits since i = n) |

The loop runs **5 times** before i = 32.

log₂(32) = 5  
 25=32

**Example 5 (ii):**

int n = 200;

for (int i = 1; i < n; i = i \* 3) {

printf("%d\n", i);

}

* i is multiplied by 3 each time.
* Sequence: 1, 3, 9, 27, 81, 243 (stops before i ≥ 200)
* Loop condition: i < n
* **Time Complexity: O(log n)**

**Example 6: Linear + Logarithmic**

for (int i = 1; i < =n; i++) {

for (int j = 1; j < n; j = j \* 2) {

printf("%d %d\n", i, j);

}

}

Outer loop: n times  
Inner loop: log n times  
Time Complexity: **O(n log n)**

**Example 7: Constant Time**

int a = 10;

int b = 20;

int sum = a + b;

printf("%d", sum);

Time Complexity: **O(1)**

**Example 8(i): Recursive Function**

void printCount(int n) {

if (n == 0)

return;

printCount(n - 1); // Recursive call

printf("%d ", n); // Print after recursion

}

**Dry Run for n = 3:**

printCount(3)

→ printCount(2)

→ printCount(1)

→ printCount(0) → return

→ printf(1)

→ printf(2)

→ printf(3)

**Output:**

1 2 3

T(n) = T(n - 1) + O(1)  
**Time Complexity: O(n)**

**Example 8(ii): Print in Reverse**

void printReverse(int n) {

if (n == 0)

return;

printf("%d ", n); // Print before recursion

printReverse(n - 1);

}

Still n calls → each prints once  
Time Complexity: **O(n)**

**Example 8(iii): Fibonacci Function**

int fib(int n) {

if (n <= 1)

return n;

return fib(n-1) + fib(n-2);

}

**Recursive Tree for fib(5)**

fib(5)

/ \

fib(4) fib(3)

/ \ / \

fib(3) fib(2) fib(2) fib(1)

/ \ / \ / \

fib(2) fib(1)fib(1)fib(0)fib(1)fib(0)

/ \

fib(1) fib(0)

Total calls: **15 function calls**

| **n** | **fib(n) Calls** | **Approximate Count** |
| --- | --- | --- |
| 1 | 1 | 1 |
| 2 | 3 | ~3 |
| 3 | 5 | ~5 |
| 4 | 9 | ~9 |
| 5 | 15 | ~15 |
| 10 | 177 | ~177 |
| 20 | ~21,891 | ~2²⁰ = 1 million+ |
| 30 | ~2^30 | Over 1 **billion** calls! |

| **Function Type** | **Example** | **Time Complexity** |
| --- | --- | --- |
| Single linear recursion | printCount(n) | O(n) |
| Tail or head recursion | printReverse(n) | O(n) |
| Double recursive calls | badCount(n) (calls twice) | O(2ⁿ) |

## What is Space Complexity?

**Space Complexity** refers to the **amount of memory (space)** an algorithm uses during its execution(run), in terms of input size n.

1. **Input space** – memory needed to store the input.
2. **Auxiliary space** – temporary variables, function calls, stacks, arrays, etc.
3. **Call stack size** – in recursion.

## Why is Space Complexity Important?

* Prevents memory overflows on large datasets.
* Helps optimize for **low-memory systems** (like embedded systems).
* Important in **recursion, dynamic programming, and parallel algorithms**.

## How to Calculate Space Complexity?

### 1. Identify variables used:

Count any extra **arrays, lists, hash tables, strings, temp variables**, etc.

### 2. Watch recursion:

Each recursive call takes **stack space**.

### 3. Look at nested/loop-allocated space:

If memory is allocated **inside a loop**, it could add up.

### 4. Ignore constants:

Just like time complexity, ignore fixed-size variables.

### Example 1: Constant Space

int a = 10;

int b = 20;

int sum = a + b;

printf("%d", sum);

* Just **three variables** (a, b, sum)
* These are **fixed in size**, no matter what n is.
* Whether n = 1 or n = 10,000, the memory used stays the same.  
  **Space Complexity = O(1)** (constant)

### Example 2: Constant Space

int a = 0, b = 1, c;

for (int i = 2; i <= n; i++) {

c = a + b;

a = b;

b = c;

}

Space: **O(1)**

**Example 3: Linear Array**

int arr[n];

for (int i = 0; i < n; i++) {

arr[i] = i \* 2;

}

* You declared an **array arr of size n**.
* That means memory is allocated for **n integers**.
* As n increases, the memory used **grows linearly**.  
  **Space Complexity = O(n)** because the **size of the array depends on input n**.

### Example 4: Nested Array (2D Matrix)

int arr[n][n];

* Space: **O(n²)**
* Explanation: 2D matrix with n × n space.

### Example 5: Recursion Stack (Linear)

void print(int n) {

if (n == 0) return;

print(n - 1);

printf("%d ", n);

}  
So for n = 3, the stack looks like this:

Call stack (top to bottom):

print(3)

print(2)

print(1)

print(0) → returns first

**Space Complexity = O(n)**

### Example 6: Binary Recursion (Exponential)

int fib(int n) {

if (n <= 1) return n;

return fib(n-1) + fib(n-2);

}

* Space: **O(n)**
* Explanation: Max depth of call stack = n, even though time is O(2ⁿ).

**4. Sorting Basics**

Sorting = arranging data in **ascending or descending order**.

**Why sorting is important?**

* Makes **searching faster** (Binary Search works only on sorted data).
* Used in **databases, data processing, ranking systems, machine learning**.

**5. Lab Work**

**Sorting in Array**

Sorting means arranging elements in a particular order (ascending or descending).  
Example: arranging **exam marks from highest to lowest**.

* **Time Complexity:**
  + Bubble Sort: O(n²)
  + Quick Sort / Merge Sort: O(n log n)
* **Space Complexity:** Depends (O(1) for bubble sort, O(log n) for quick sort recursion).

**1. Selection Sort**Selection Sort is a simple sorting algorithm that repeatedly selects the **smallest (or largest)** element from the unsorted part of the array and places it at the correct position in the sorted part.

**Example**

Array: **[64, 25, 12, 22, 11]**

* Pass 1: Smallest = 11 → Swap with 64 → [11, 25, 12, 22, 64]
* Pass 2: Smallest = 12 → Swap with 25 → [11, 12, 25, 22, 64]
* Pass 3: Smallest = 22 → Swap with 25 → [11, 12, 22, 25, 64]
* Pass 4: Smallest = 25 → Swap with 25 → [11, 12, 22, 25, 64]  
  Sorted.

**Time Complexity**

* Best Case: **O(n²)**
* Average Case: **O(n²)**
* Worst Case: **O(n²)**  
  ➡ Reason: Always checks every element for minimum.

**Space Complexity**

* **O(1)** → In-place algorithm.

#include <iostream>

using namespace std;

void selectionSort(int arr[], int n) {

for(int i = 0; i < n-1; i++) {

int minIndex = i;

for(int j = i+1; j < n; j++) {

if(arr[j] < arr[minIndex])

minIndex = j;

}

swap(arr[i], arr[minIndex]);

}

}

int main() {

int arr[] = {64, 25, 12, 22, 11};

int n = 5;

selectionSort(arr, n);

cout << "Sorted array: ";

for(int i = 0; i < n; i++)

cout << arr[i] << " ";

return 0;

}

**2. Bubble Sort**Bubble Sort is a simple sorting algorithm that repeatedly compares **adjacent elements** and swaps them if they are in the wrong order.  
➡ The largest element "bubbles up" to the last position after each pass.

**Example**

Array: **[5, 1, 4, 2, 8]**

* Pass 1: [1, 4, 2, 5, 8]
* Pass 2: [1, 2, 4, 5, 8]
* Pass 3: [1, 2, 4, 5, 8] (already sorted, stops early).  
  Sorted.

**Time Complexity**

* Best Case: **O(n)** (when array already sorted, with optimized version).
* Average Case: **O(n²)**
* Worst Case: **O(n²)**

**Space Complexity**

* **O(1)** → In-place sorting.

**Stability**

* **Stable** (order of equal elements preserved).

#include <iostream>

using namespace std;

void bubbleSort(int arr[], int n) {

for(int i = 0; i < n-1; i++) {

bool swapped = false;

for(int j = 0; j < n-i-1; j++) {

if(arr[j] > arr[j+1]) {

swap(arr[j], arr[j+1]);

swapped = true;

}

}

if(!swapped) break; // optimization

}

}

int main() {

int arr[] = {5, 1, 4, 2, 8};

int n = 5;

bubbleSort(arr, n);

cout << "Sorted array: ";

for(int i = 0; i < n; i++)

cout << arr[i] << " ";

return 0;

}

**3. Insertion Sort**  
Insertion Sort builds the final sorted array one item at a time.  
It picks an element from the unsorted part and repeatedly **swaps it backward** into its correct position (like bubbling it into the right place).  
This version uses **nested for loops with swapping** instead of shifting.

**Working Principle**

* Consider the first element as already sorted.
* Take the next element and compare it with its previous element.
* If it is smaller, swap them.
* Keep swapping backward until it reaches its correct position.
* Repeat this process for all elements until the array is sorted.

Array: [12, 11, 13, 5, 6]

* **Pass 1 (i = 1, element = 11):**  
  Compare 11 with 12 → swap → [11, 12, 13, 5, 6]
* **Pass 2 (i = 2, element = 13):**  
  Compare 13 with 12 → already greater → no swap → [11, 12, 13, 5, 6]
* **Pass 3 (i = 3, element = 5):**  
  Compare 5 with 13 → swap → [11, 12, 5, 13, 6]  
  Compare 5 with 12 → swap → [11, 5, 12, 13, 6]  
  Compare 5 with 11 → swap → [5, 11, 12, 13, 6]
* **Pass 4 (i = 4, element = 6):**  
  Compare 6 with 13 → swap → [5, 11, 12, 6, 13]  
  Compare 6 with 12 → swap → [5, 11, 6, 12, 13]  
  Compare 6 with 11 → swap → [5, 6, 11, 12, 13]  
  Stop (correct position reached).

Final Sorted Array = [5, 6, 11, 12, 13]

**Time Complexity**

* **Best Case (Already Sorted):** O(n)
* **Average Case:** O(n²)
* **Worst Case (Reverse Sorted):** O(n²)

**Space Complexity**

* O(1) → In-place sorting

**Stability**

* **Stable**, because equal elements do not get swapped unnecessarily.

#include <iostream>

using namespace std;

void insertionSort(int arr[], int n) {

for (int i = 1; i < n; i++) {

for (int j = i; j > 0; j--) {

if (arr[j] < arr[j - 1]) {

// Swap adjacent elements

swap(arr[j], arr[j - 1]);

} else {

// Already in correct place → stop

break;

}

}

}

}

int main() {

int arr[] = {12, 11, 13, 5, 6};

int n = 5;

insertionSort(arr, n);

cout << "Sorted array: ";

for (int i = 0; i < n; i++)

cout << arr[i] << " ";

return 0;

}