Analysis of Algorithms

LAB 1

Lecturer: Miss Ayesha Majid Ali

DEPARTMENT OF COMPUTER SCIENCE

RIPHAH SCHOOL OF COMPUTING AND INNOVATION

LAB MANUAL: ASYMPTOTIC ANALYSIS

The aim of this lab is to:

- Understand the concept of asymptotic notations.
- Learn how to analyze algorithm efficiency using time complexity.
- Compare different algorithms using Big-O, Big-Ω, and Big-Θ.
- Implement simple programs to measure runtime performance.

2. Theory

2.1 What is Asymptotic Analysis?

Asymptotic analysis is a method of describing the **running time of an algorithm** as the **input size (n)** grows towards infinity.
It helps us focus on the **growth rate**, not the exact number of operations.

It tells us:

How does the runtime of an algorithm increase when input size increases?

2.2 Primitive Operations

When analyzing code, we count basic operations like:

- Assignments (x = y)
- Comparisons (if (a < b))
- Arithmetic operations (a + b, a * b)
- Function calls (swap(a, b))
- Return statements (return)

Each primitive operation is considered to take constant time (O(1)).

2.3 Why We Ignore Constants

When comparing algorithms, we only care about **relative growth**.

Example:
If Algorithm A takes 6n + 6 operations and Algorithm B takes $2n^2$,
then for large n, the quadratic term dominates — constants become negligible.

3. Asymptotic Notations

3.1 Big O (O) — Upper Bound

- Represents the **worst-case** growth rate.
- Tells how fast the **runtime increases at most**.

Definition:

$f(n)=O(g(n))$ if there exist $c>0, n0>0$ such that $f(n) \le c \cdot g(n)$ for all $n \ge n0$

✅ **Example:**

$$2n+10 \le 12n \Rightarrow 2n+10=O(n)$$

3.2 Big Omega (Ω) — Lower Bound

- Represents the **best-case** growth rate.
- Describes the **minimum time** the algorithm will take.

Definition:

$$f(n)=\Omega(g(n)) \text{ if } f(n) \ge c \cdot g(n) \text{ for all } n \ge n0$$

Example:

$$3n^2+5n+2 \ge 3n^2 \Rightarrow f(n)=\Omega(n^2)$$

3.3 Big Theta (Θ) — Tight Bound

- Represents both **upper and lower bounds**.
- Means the algorithm grows at the **same rate** as g(n).

Definition:

$f(n)=\Theta(g(n))$ if $f(n)=O(g(n))$ and $f(n)=\Omega(g(n))$

Example:

$5n^2+4n+1=\Theta(n^2)$

4. Step-by-Step Proof Examples

Example 1 — Linear

Prove $3n+5=O(n)$

For n≥1

3n+5≤3n+5n=8n

c=8,n0=1

Example 2 — Quadratic

Prove $2n^2 + 3n + 10 = O(n^2)$

$$2n^2 + 3n + 10 \le 2n^2 + 3n^2 + 10n^2 = 15n^2$$

✅ $c = 15, n_0 = 1$

Example 3 — Cubic

Prove $4n^3 + 2n^2 \ldots (n^3)$

Ask ChatGPT

$$4n^3 + 2n^2 + n + 5 \le 4n^3 + 2n^3 + n^3 + 5n^3 = 12n^3$$

✅ $c = 12, n_0 = 1$

5. Coding Examples

Example 1: Counting Operations

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

int main() {
    int n = 5;
    int count = 0;

    for (int i = 0; i < n; i++) {
        count++;        // Operation 1
    }

    cout << "Total operations: " << count << endl;
    // Complexity: O(n)
    return 0;
}
```

Example 2: Nested Loops (O(n²))

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

int main() {
    int n = 4, count = 0;
```

```cpp
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            count++;
        }
    }
    cout << "Total operations: " << count << endl;
    // Complexity: O(n²)
    return 0;
}
```

Example 3: Logarithmic Loop (O(log n))

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

int main() {
    int n = 64, count = 0;

    while (n > 1) {
        n /= 2;
        count++;
    }

    cout << "Total steps: " << count << endl;
    // Complexity: O(log n)
    return 0;
}
```

6. Common Time Complexities

Complexity	Example Algorithm	Growth Type
O(1)	Accessing array element	Constant
O(log n)	Binary Search	Logarithmic
O(n)	Linear Search	Linear
O(n log n)	Merge Sort	Linearithmic
$O(n^2)$	Bubble Sort	Quadratic
$O(n^3)$	Matrix Multiplication	Cubic
$O(2^n)$	Recursion (subset)	Exponential
O(n!)	Traveling Salesman	Factorial

7. Experiment Tasks

Task 1: Linear Search

1. Implement a simple linear search algorithm.
2. Count how many comparisons your program makes.
3. Observe how this number changes as the array size increases.
4. Determine whether the algorithm is O(n).

Concept Recap

Before coding, understand:

- Linear Search means checking each element one by one until the target value is found (or the array ends).
- If the array has *n* elements, in the worst case, you might check all *n* elements.
- Hence, we expect the time complexity to be O(n) — meaning the time grows *linearly* with the input size.

Try the following array sizes

Array Size (n)	**Comparisons (Worst Case)**	**Observation**
10		**Comparisons ?**
100		**Comparisons ?**
500		**Comparisons ?**
1,000		**Comparisons ?**
5,000		**Comparisons ?**

Task 2:

Binary Search and Time Complexity Analysis

1. Implement Binary Search on a sorted array.
2. Count how many comparisons are made while searching.
3. Observe how comparisons grow as array size (*n*) increases.
4. Verify that the algorithm follows O(log n) complexity.

Concept Recap

- Binary Search works only on sorted arrays.
- It repeatedly divides the search space in half:
    1. Compare the middle element with the target.
    2. If equal → found!
    3. If target < middle → search in left half.
    4. If target > middle → search in right half.
- Each step cuts the problem size by half → giving a logarithmic growth pattern.

Array Size (n)	**Comparisons (Worst Case)**	**$\log_2(n)$ (Approx.)**
10		
100		
1,000		
5,000		
10,000		

Task 3: Compare Linear and Binary Search.

Task 4: Nested Loops

- Write a program with two nested loops.
- Count operations for increasing input sizes.
- Verify $O(n^2)$ behaviour.