UST
Faculty of Engineering
First Year [Sem2]

Data Structure and Algorithms

Lecture 2

Introduction to Algorithm Analysis

Algorithm

- An **algorithm** is a finite sequence of well-defined instructions used to solve a problem or perform a specific task.
- A good algorithm should satisfy several essential characteristics.

Algorithm Characteristics

1. Input

- An algorithm accepts **one or more input values**.
- Inputs are provided before or during execution.

Example:

- Input: Two numbers (5 and 8)
- Task: Find their sum

2. Output

- An algorithm produces **at least one output**.
- The output is the result of processing the input.

Example:

- Input: Two numbers (5 and 8)
- output: sum=13

3. Definiteness (Unambiguity)

- Every instruction must be **clear, precise, and unambiguous**.
- Each step should have only one meaning.

☑ Good Instruction

- Multiply A by B.

❌ Poor Instruction

- Process the numbers.
- (The word *process* is unclear.)

4. Finiteness

- The algorithm must **terminate after a finite number of steps**.
- It should never run forever.

☑ **Example**

- Repeat 10 times.
- Stop.

❌ **Example**

- Repeat forever.

5. Effectiveness

- Every instruction must be **simple and executable**.
- Each step should be completed in a reasonable amount of time.

Example

- Read a number.
- Add 1.
- Display the result.
- These are simple operations that a computer can perform.

6. Correctness

- The algorithm should produce the **correct output** for every valid input.

Example

- Input: 8, 4
- Output: 12
- If the algorithm outputs **15**, it is incorrect.

7. Generality

- The algorithm should solve **all instances of a problem**, not just one specific case.

Example

- Algorithm for addition:

```
Input: Any two numbers
Output: Their sum
```

- Works for:

```
2 + 3
100 + 250
7.5 + 9.3
```

8. Efficiency

- The algorithm should use **minimum execution time** and **minimum memory**.
- Efficient algorithms perform better on large datasets.

Example

- Searching:
- Linear Search → **O(n)**
- Binary Search → **O(log n)** (more efficient for sorted data)

Summary Table

Characteristic	Description	Example
Input	Accepts zero or more inputs	Two numbers
Output	Produces at least one result	Sum of numbers
Definiteness	Steps are clear and unambiguous	"Add A and B"
Finiteness	Ends after a finite number of steps	Loop 10 times
Effectiveness	Steps are simple and executable	Add, Compare, Print
Correctness	Produces correct results	5 + 3 = 8
Generality	Solves all valid instances	Works for any numbers
Efficiency	Uses minimal time and memory	Binary Search

Time Complexity

Time complexity is a measure of how the running time of an algorithm grows as the size of the input increases.

It describes the relationship between the input size and the number of basic operations an algorithm performs, rather than the actual execution time in seconds.

Definition:

- **Time complexity** is the amount of computational work an algorithm performs as a function of the input size, usually denoted by n.

Why Do We Study Time Complexity?

- Different computers have different processors, memory sizes, and operating systems. Therefore, measuring an algorithm by the actual execution time (e.g., 2 seconds or 5 seconds) is unreliable.

Instead, computer scientists analyze:

- The **number of operations** performed.
- How the number of operations increases as the input size (**n**) grows.
- This provides a hardware-independent way to compare algorithms.

Example 1: Constant Time – O(1)

```
</>  C++

int x = 10;
cout << x;
```

The program executes only one operation regardless of the input size.

Input Size (n)	Operations
10	1
100	1
1,000	1

Time Complexity: O(1) (Constant Time)

Example 2: Linear Time – O(n)

```C++
for (int i = 0; i < n; i++)
{
    cout << i << " ";
}
```

- The loop executes n times.

n	Operations
5	5
10	10
100	100
1000	1000

Time Complexity: O(n)

Real-Life Example

- Imagine searching for a student's name in a class list.
- **10 students** → search is quick.
- **100 students** → takes longer.
- **1,000 students** → takes much longer.
- The larger the input size, the more work the algorithm performs.

How Do We Measure Time Complexity?

We count the number of basic operations, such as:

- Comparisons (<, >, ==)
- Assignments (=)
- Arithmetic operations (+, -, *, /)
- Loop iterations
- Function calls (when appropriate)
- The exact execution time in seconds is not measured because it depends on the hardware.

Common Time Complexities

Complexity	Name	Performance
O(1)	Constant	Excellent
O(log n)	Logarithmic	Very Fast
O(n)	Linear	Good
O(n log n)	Linearithmic	Efficient
$O(n^2)$	Quadratic	Slow for large inputs
$O(n^3)$	Cubic	Very Slow
$O(2^n)$	Exponential	Extremely Slow

Space Complexity

- **Space complexity** is a measure of the amount of memory an algorithm requires during its execution as the input size increases.
- It describes how memory usage grows with the size of the input, usually denoted by n.

Definition:

- **Space complexity** is the total amount of memory used by an algorithm during execution as a function of the input size (n).
- Unlike **time complexity**, which measures the number of operations, **space complexity measures memory consumption**.

Components of Space Complexity

Space complexity consists of two main parts:

1. Input Space

- The memory required to **store the input data** provided to the algorithm.

Example:

```C++
int arr[100];
```

The array itself occupies memory for **100 integers**. This is the **input space**.

2. Auxiliary Space

- The **extra memory** used by the algorithm during execution, excluding the input data.

Examples include:

- Temporary variables
- Additional arrays
- Stacks
- Queues
- Recursive function calls

Formula

$$\text{Space Complexity} = \text{Input Space} + \text{Auxiliary Space}$$

Example 1: Constant Space — O(1)

```cpp
int sum = 0;

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

n	Memory Used
10	2 variables
100	2 variables
1000	2 variables

Memory Used

- Variable sum
- Variable i
- Only two variables are used regardless of the value of n.

Space Complexity: O(1) (Constant Space)

Example 2: Linear Space — O(n)

```cpp
int arr[n];

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

n	Memory Required
10	10 integers
100	100 integers
1000	1000 integers

- The algorithm creates an array of size **n**.
- Memory increases depend on the input size.

Space Complexity: O(n)

Real-Life Example

- Imagine packing books into a backpack.
- A **small backpack** represents **low space complexity**.
- A **large backpack** represents **high space complexity**.
- If you only carry one notebook regardless of the task, your memory usage is **constant (O(1))**.
- If you need one notebook for every student in the class, memory usage grows with the number of students, giving **O(n)**.

Common Space Complexities

Complexity	Name	Memory Growth
O(1)	Constant	Fixed amount of memory
O(log n)	Logarithmic	Grows very slowly
O(n)	Linear	Memory grows proportionally with input size
O(n log n)	Linearithmic	Moderate memory growth
$O(n^2)$	Quadratic	Large memory requirement
$O(2^n)$	Exponential	Extremely high memory usage

Time Complexity vs. Space Complexity

Feature	Time Complexity	Space Complexity
Measures	Running time	Memory usage
Focus	Number of operations	Amount of memory
Unit	Growth of execution time	Growth of memory consumption
Objective	Faster algorithms	Memory-efficient algorithms