

1 Introduction and Algorithm Complexity

1.1 What is a Data Structure?

Data **structure** is a way of organizing and storing data on a computer so that it can be accessed and modified efficiently.

Different problems require different ways of organizing data.

Examples of data structures:

- Array
- Stack
- Queue
- Linked List
- Tree
- Hash Table

Real-life analogy:

- Books thrown randomly in a room → hard to find one.
- Books arranged on shelves by category → easy to find.

The way we organize data affects:

- Speed of access
- Memory usage
- Simplicity of the program

1.2 What is an Algorithm?

An **algorithm** is a sequence of well-defined steps used to solve a problem.

Characteristics of a good algorithm:

- Clear and unambiguous steps
- Finite number of steps
- Produces correct output
- Efficient in time and memory

Example: Find the maximum of three numbers

Steps:

1. Read three numbers: L, M, N .
2. Set $OLP = L$.
3. If $M > OLP$, set $OLP = M$.
4. If $N > OLP$, set $OLP = N$.
5. Print OLP .

1.3 Program vs Algorithm

- **Algorithm:** The idea or method to solve a problem.
- **Program:** The implementation of the algorithm in a programming language.

Two programs may:

- Solve the same problem
- Produce the same output

But:

- One may take 1 second
- The other may take 5 minutes

The difference is:

Algorithm efficiency

1.4 Why Efficiency Matters

Consider searching for a number in a list:

- If the list has 10 elements → fast
- If the list has 1,000,000 elements → much slower

In real systems:

- Search engines process billions of records
- Social networks store millions of users
- Online stores handle thousands of transactions per second

Efficient algorithms:

- Save time
- Save memory

- Reduce system cost

1.5 Measuring Algorithm Efficiency

We measure efficiency mainly by:

- Time complexity
- Space complexity

In this course, we focus mostly on:

Time complexity

We measure:

- Number of operations
- As input size increases

Example: Finding Maximum Value

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int n, maxVal, x;
6     cin >> n;
7     cin >> x;
8     maxVal=x;
9
10    for (int i = 1; i < n; i++) {
11        cin >> x;
12        if (x > maxVal)
13            maxVal = x;
14    }
15
16    cout << maxVal;
17    return 0;
18 }
```

Analysis:

- Loop runs $Q \rightarrow 1$ times.
- Each iteration does a comparison.

Total operations almost n

So the time complexity is:

$$R(Q)$$

1.6 Big-O Notation

Big-O notation describes how the running time of an algorithm grows as the input size increases.

It does NOT measure actual time in seconds. Instead, it measures the **growth rate**.

It focuses on:

- Growth rate
- Not exact time

Big-O notation represents the upper bound (worst-case scenario).

We use worst case because:

- It guarantees performance.
- It prepares for maximum input.
- It ensures scalability.

Common Time Complexities

Complexity	Name
$R(1)$	Constant time
$R(\log Q)$	Logarithmic time
$R(Q)$	Linear time
$R(Q \log Q)$	Linearithmic time
$R(Q^2)$	Quadratic time
$R(2^Q)$	Exponential time

Example Comparisons

Algorithm A:

$$S(Q) = Q$$

Algorithm B:

$$S(Q) = Q^2$$

If:

$$Q = 10$$

- A $\rightarrow$ 10 operations

- $B \rightarrow 100$ operations

If:

$$Q = 1000$$

- $A \rightarrow 1,000$ operations
- $B \rightarrow 1,000,000$ operations

Big difference!

More Big-O Examples

- $f(n) = 5n + 20$
 - Solution: $O(n)$
- $T(Q) = 7Q^2 + 3Q + 9$
 - Solution: $R(Q^2)$
- $T(Q) = Q^3 + Q^2 + Q$
 - Solution: $R(Q^3)$
- $T(Q) = Q \log Q + Q$
 - Solution: $R(Q \log Q)$
- $T(Q) = 500$
 - Solution: $R(1)$
- $T(Q) = 2^Q + Q^3$
 - Solution: $R(2^Q)$

When finding Big-O:

- Ignore constants.
- Ignore lower-order terms.
- Keep the fastest-growing term.

Example 1: Find the greater of two numbers.

```
1 int max_value(int a, int b)
2 {
3     if (a > b)
4         return a;
5     else
6         return b;
```

7 }

So:

- The function performs only one comparison.
- There is no loop.
- The number of operations does not depend on input size.

$$S(Q) = R(1)$$

Example 2: Find the maximum element in an array.

```

1 int findMax(int arr[], int n)
2 {
3     int maxVal = arr[0];
4
5     for (int i = 1; i < n; i++)
6     {
7         if (arr[i] > maxVal)
8             maxVal = arr[i];
9     }
10
11     return maxVal;
12 }
```

So:

- The loop runs n times.
- Each iteration performs one comparison.
- The number of operations increases linearly with n.

$$S(Q) = R(Q)$$

Example 3: Print all pairs of elements in an array.

```

1 void printPairs(int arr[], int n)
2 {
3     for (int i = 0; i < n; i++)
4     {
5         for (int j = 0; j < n; j++)
6         {
7             cout << arr[i] << "," << arr[j] << endl;
```

```

8         }
9     }
10 }
```

So:

- The outer loop runs n times.
- The inner loop also runs n times.
- For each outer iteration, the inner loop executes n times.
- Total operations: $n \times n = n^2$.

$$S(Q) = R(Q^2)$$

Summary

Complexity	Code Pattern	Example
$O(1)$	No loops	Compare two numbers
$O(n)$	One loop	Find max in array
$O(n^2)$	Nested loops	Print all pairs / Bubble sort

Time complexity according to Big O

n	$R(1)$	$R(\log Q)$	$R(Q)$	$R(Q \log Q)$	$R(Q^2)$	$R(Q^3)$	$R(2^Q)$	$R(Q!)$
5	1	2.32	5	11.61	25	125	32	120
10	1	3.32	10	33.22	100	1000	1024	3,628,800
100	1	6.64	100	664.39	10000	1000000	1.27e+30	9.33e+157
1000	1	9.97	1000	9965.78	1000000	1000000000	1.07e+301	Too large
10000	1	13.29	10000	132877.12	100000000	1000000000000	Too large	Too large

9. Types of Complexity Cases

Best Case:

- Minimum time required

Worst Case:

- Maximum time required

Average Case:

- Expected time for random input

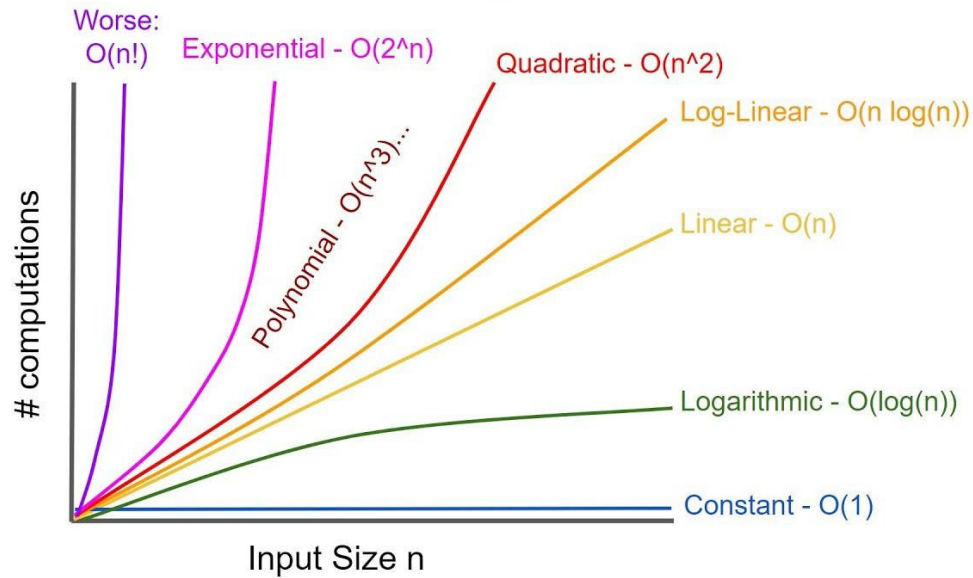


Figure 1: **Growth Rates of Common Time Complexities:**

We focus on **worst-case complexity**.

Important Note

When calculating time complexity:

- Count how many times loops run.
- Multiply nested loops.
- Add separate blocks.
- Division or doubling usually means logarithmic complexity.

Exercises

1. What is a data structure?
2. What is an algorithm?
3. What is the difference between an algorithm and a program?
4. What does Big-O measure?
5. Which is faster: $R(Q)$ or $R(Q^2)$ for large Q ?