Sorting Algorithms I

Bubble Sort, Selection Sort, and Insertion Sort

Sorting Algorithms I

Laboratory Session 12

Bubble Sort

Compare adjacent elements and swap them

Selection Sort

Find the minimum element and place it in position

Insertion Sort

Insert elements into the correct position

Objective: Understand the principles of the simplest sorting algorithms, their efficiency, and their application areas in C++ programming.

Why is sorting needed?

Sorting is one of the most fundamental operations in computer science and programming. It forms the basis for many algorithms and data structures.

Student Lists

Ordering students by last name for easy searching and organization of the educational process

Library Catalogs

Sorting books by alphabet, genre, or author for quick retrieval of necessary literature

File Systems

Organizing files by creation date, size, or type for efficient work

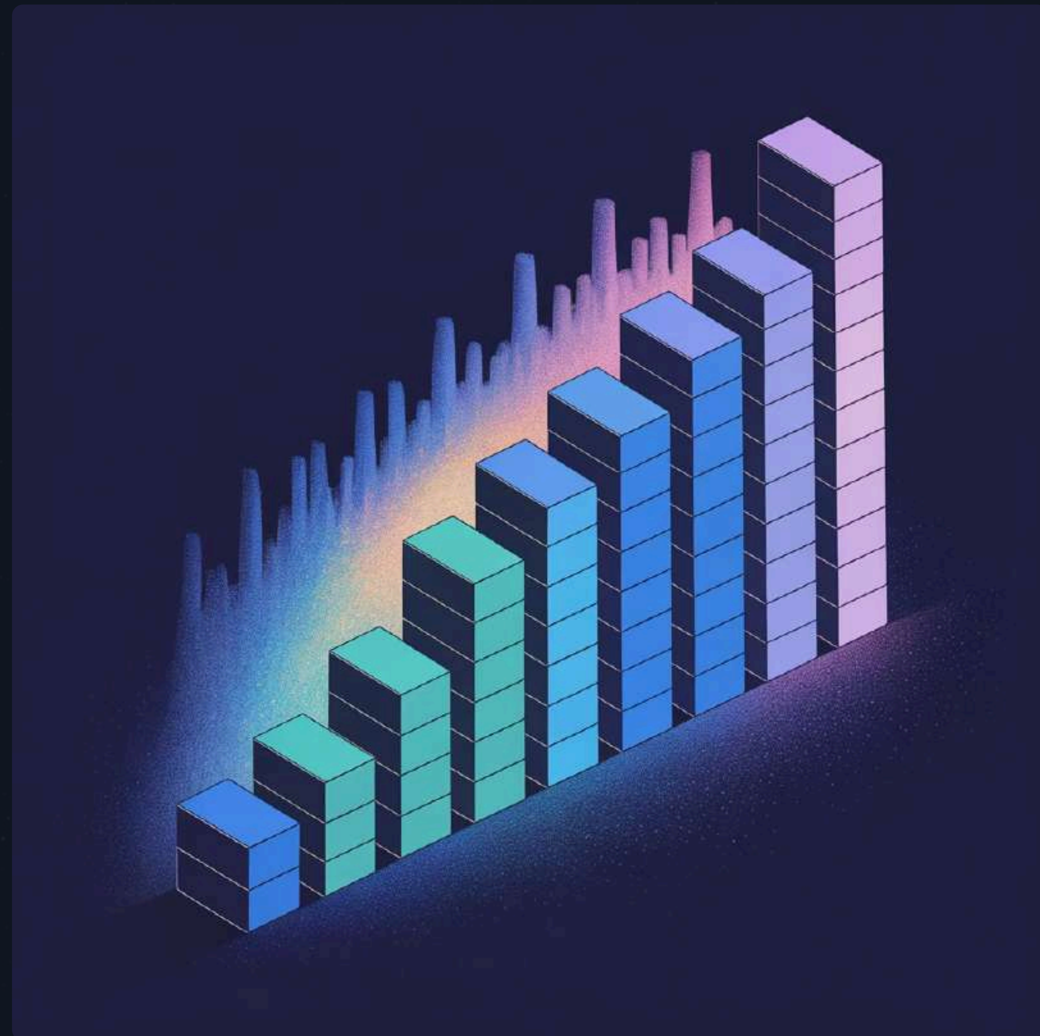
Fundamentals of Sorting

Input Data

An array of n elements of arbitrary type that needs to be ordered

Output Data

The same array, but with elements arranged in a specific order (ascending or descending)



Execution Time

How many operations are required for sorting

Memory

How much additional memory the algorithm uses

Stability

Whether the relative order of equal elements is preserved

🟢 Bubble Sort

Bubble Sort gets its name from the way elements "bubble up" to their correct positions, similar to air bubbles rising in water.

01

Array Traversal

Compare each pair of adjacent elements from beginning to end

02

Comparison and Swap

If elements are in the wrong order, swap their positions

03

Repetition

Repeat the process until the array is fully sorted

Example of Bubble Sort Operation

Let's consider the step-by-step execution of the algorithm on the array [5, 3, 4, 1]:

1. **Initial Array: [5, 3, 4, 1]**

   Starting with an unsorted array

2. **First Pass: [3, 4, 1, 5]**

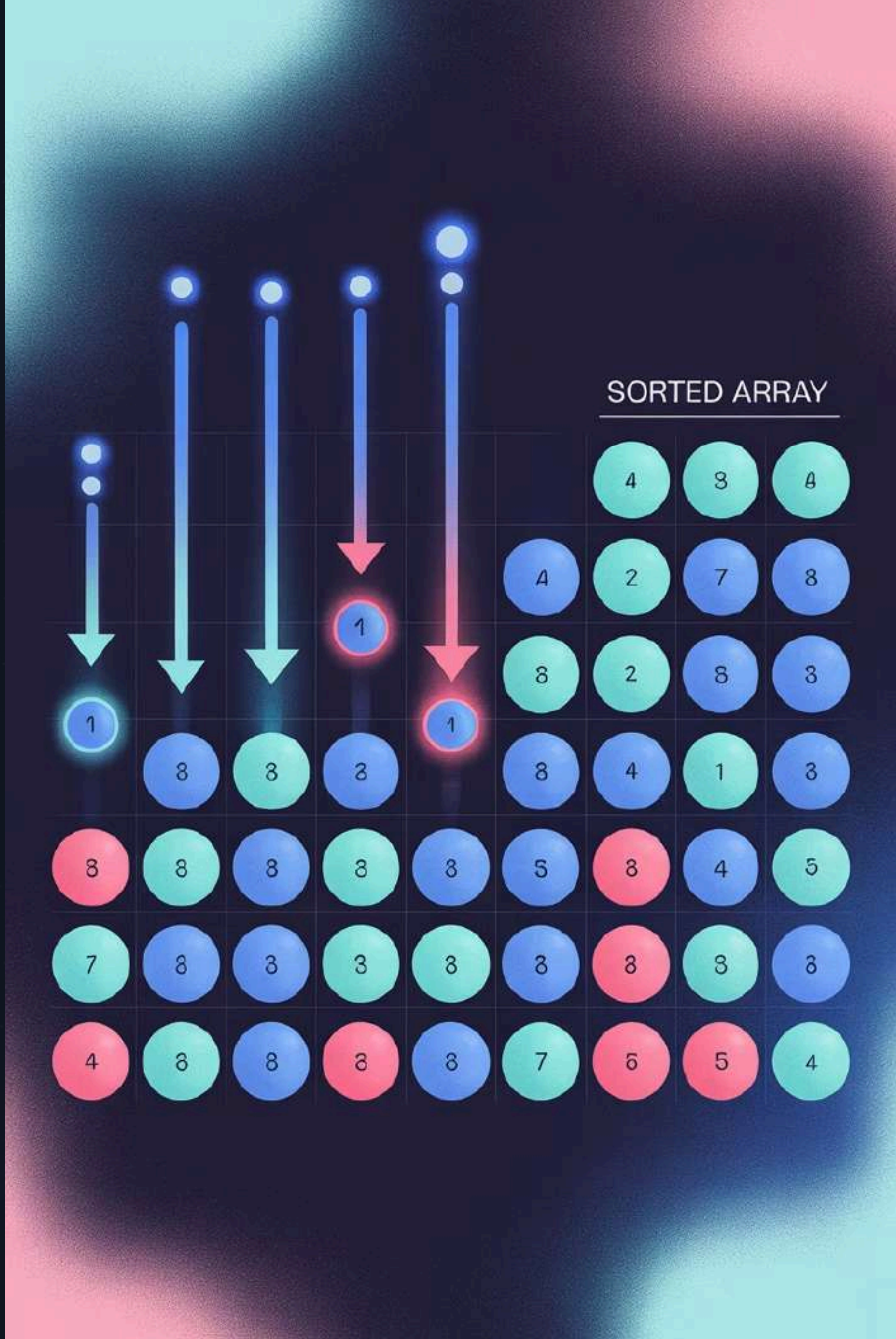
   5 "bubbles up" to the last position, comparing with each neighbor

3. **Second Pass: [3, 1, 4, 5]**

   4 finds its place, 5 is already in position

4. **Third Pass: [1, 3, 4, 5]**

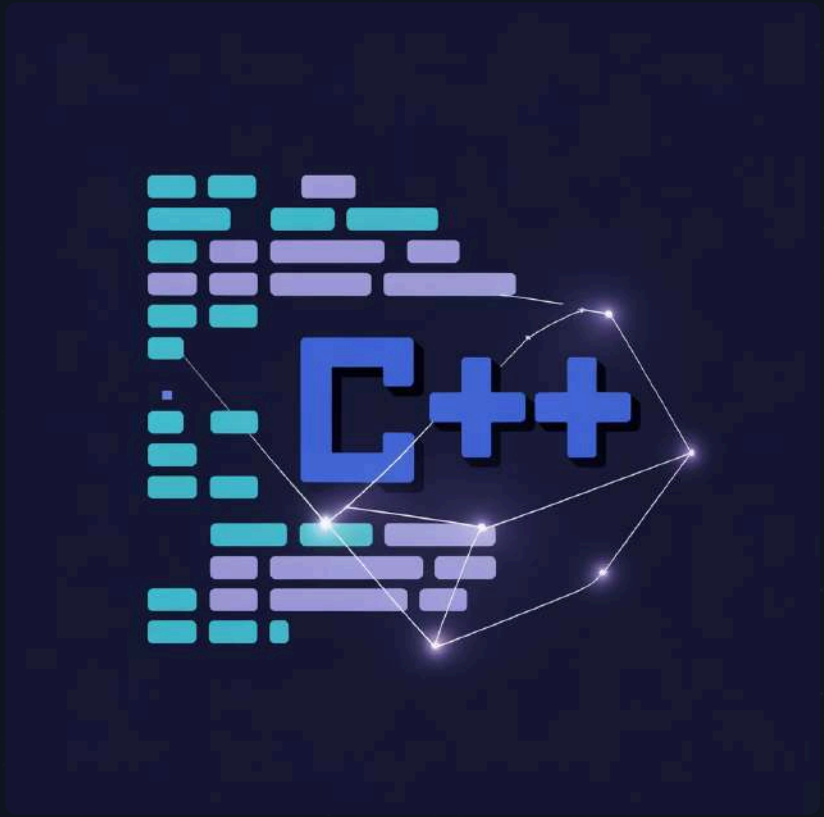
   The array is completely sorted



Bubble Sort Implementation in C++

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}
```

The outer loop determines the number of passes, and the inner loop determines the comparisons in each pass. The `swap()` function exchanges two array elements.



Optimization: A flag can be added for an early exit if no swaps occurred during a pass - the array is already sorted.

Analysis of Bubble Sort Complexity

$O(n^2)$

Worst Case

Array sorted in reverse order

$O(n^2)$

Average Case

Elements are in random order

$O(n)$

Best Case

Array is already sorted (with optimization)

$O(1)$

Memory

Algorithm sorts "in place"

Bubble sort performs up to n(n-1)/2 comparisons and swaps in the worst case, resulting in quadratic time complexity.

🟡 Selection Sort

Selection sort works by finding the minimum (or maximum) element and placing it in the correct position.

1

Find Minimum

Find the smallest element in the unsorted portion

2

Swap

Swap it with the first element of the unsorted portion

3

Shift Boundary

Increase the sorted portion by one element

Real-life analogy: Imagine you select the shortest person from a group and place them first in a row, then from the remaining, you again select the shortest and place them second, and so on.

Example of Selection Sort in Action

Let's trace the algorithm's execution on the same array [5, 3, 4, 1]:

Step 1: [5, 3, 4, 1]

Minimum = 1, swap with the first element

Step 2: [1, 3, 4, 5]

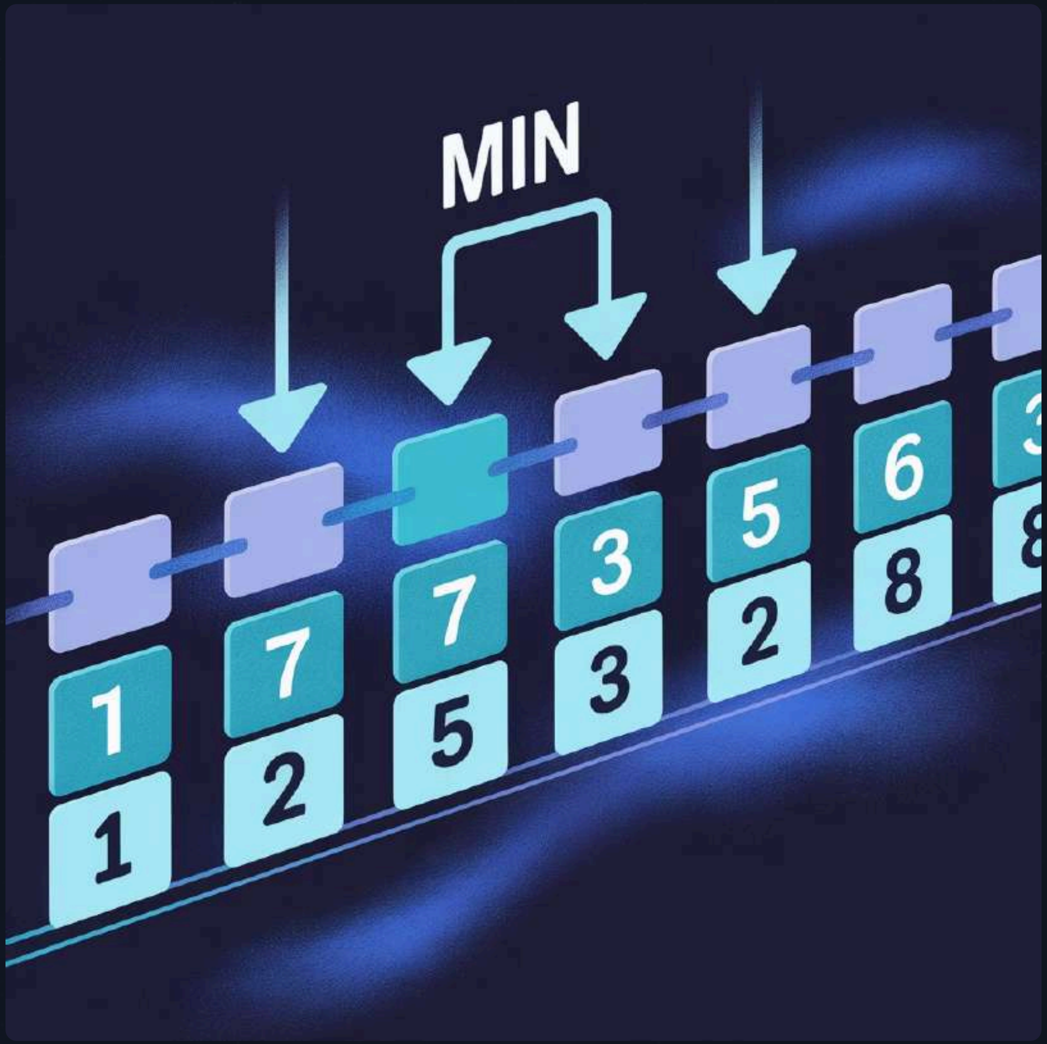
Minimum among [3, 4, 5] = 3, already in place

Step 3: [1, 3, 4, 5]

Minimum among [4, 5] = 4, already in place

Result: [1, 3, 4, 5]

Array is fully sorted



Selection Sort Implementation in C++

```
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]);
    }
}
```

Outer Loop

Iterates through positions to place minimum elements

Inner Loop

Finds the minimum element in the unsorted part

Swap

Places the found minimum in its correct position

Analysis of Selection Sort Complexity

Selection sort has stable time complexity in all cases:

Worst Case	Average Case	Best Case
$O(n^2)$	$O(n^2)$	$O(n^2)$
Always requires finding the minimum	Number of comparisons does not depend on data	Even in a sorted array, comparisons are needed

Feature: Selection sort performs exactly n-1 swaps regardless of input data, which can be useful for expensive permutation operations.

🔵 Insertion Sort

Insertion sort works similarly to how we sort playing cards in our hands – we pick up a card and insert it into the correct position among the already sorted ones.

Partitioning

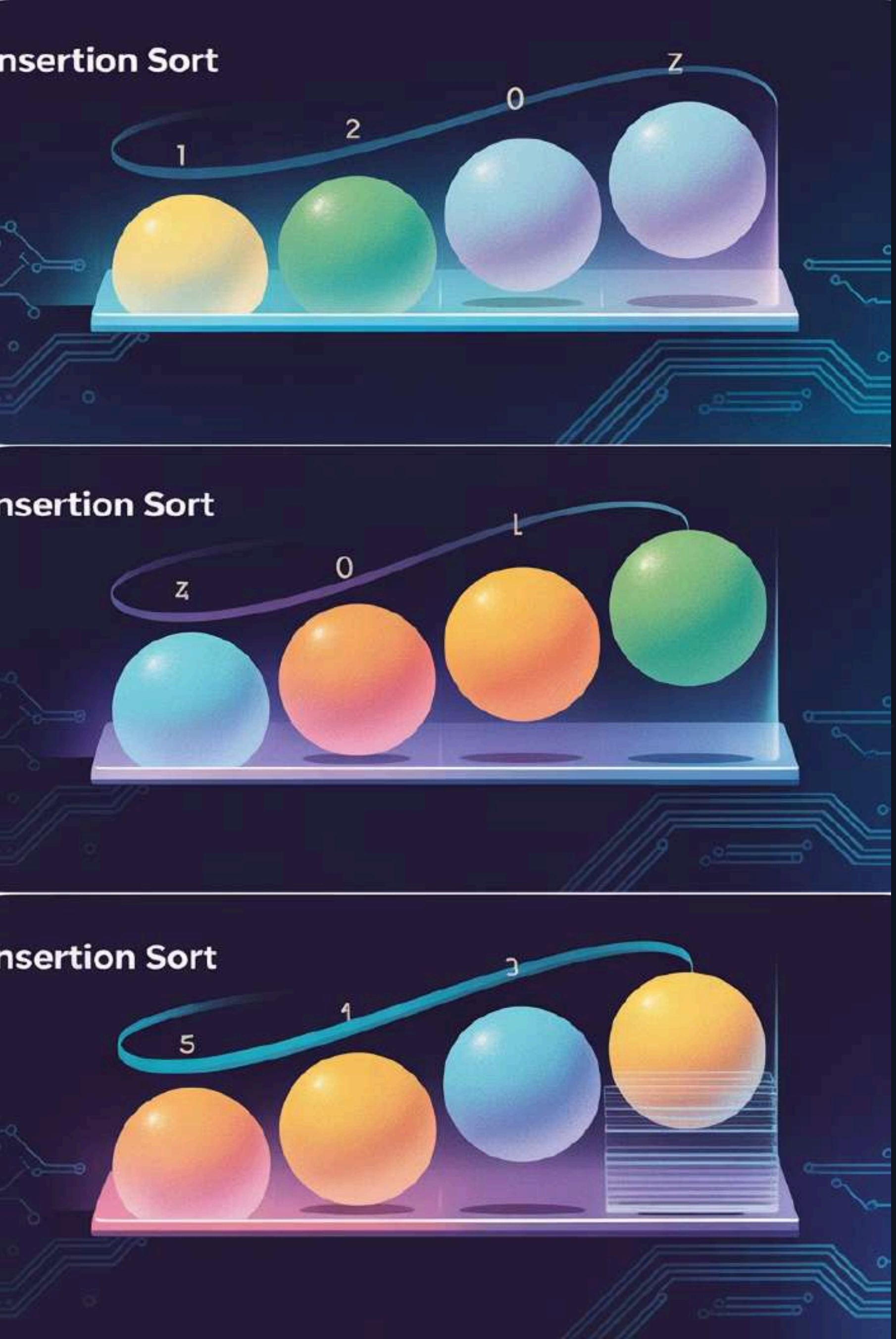
The array is logically divided into sorted and unsorted parts

Extraction

Take the first element from the unsorted part

Insertion

Find the correct place in the sorted part and insert the element



Example of Insertion Sort in Action

Let's consider the algorithm's execution on the array [5, 3, 4, 1]:

Start: [5 | 3, 4, 1]

The first element is considered sorted

Insert 3: [3, 5 | 4, 1]

3 is inserted before 5

Insert 4: [3, 4, 5 | 1]

4 is inserted between 3 and 5

Insert 1: [1, 3, 4, 5 |]

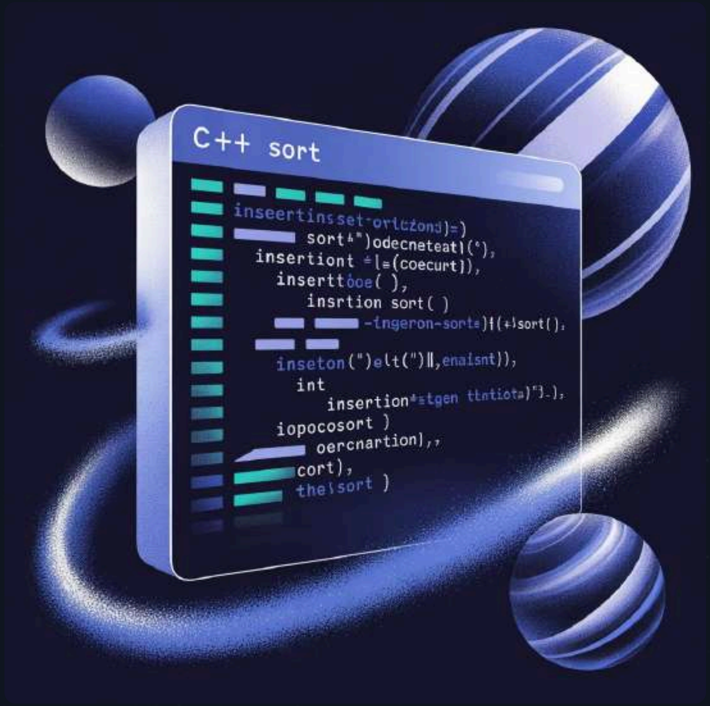
1 is inserted at the beginning, the array is sorted

Insertion Sort Implementation in C++

```
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
```

The `key` variable stores the current element to be inserted, and the `while` loop shifts larger elements to the right until the correct position is found.



Key Idea: Shift elements larger than the current one to the right, creating space for insertion.

Analysis of Insertion Sort Complexity

$O(n^2)$

Worst Case

Array sorted in reverse order

$O(n^2)$

Average Case

Randomly ordered data

$O(n)$

Best Case

Array already sorted or nearly sorted

Advantage: Insertion sort is very efficient for nearly sorted arrays, as it requires a minimal number of comparisons and swaps.

Comparison of Three Algorithms

Algorithm	Worst Case	Best Case	Stability	Applicability
Bubble Sort	$O(n^2)$	$O(n)$	Yes	Educational Examples
Selection Sort	$O(n^2)$	$O(n^2)$	No	Small Data Sets
Insertion Sort	$O(n^2)$	$O(n)$	Yes	Nearly Sorted Data

Each algorithm has its own characteristics and application areas. The choice depends on data characteristics and performance requirements.

Practical Conclusions

Quadratic Complexity

All three algorithms have a time complexity of $O(n^2)$, making them inefficient for large arrays (more than 10,000 elements)

Educational Value

These algorithms are ideal for understanding the basic principles of sorting, comparisons, and element swaps

Practical Application

Used for small arrays (up to 50 elements) or as components of more complex hybrid sorting algorithms

Modern C++ libraries (e.g., std::sort) employ more efficient algorithms, but understanding simple sorting methods remains fundamental for any programmer.

Reinforcement Questions

Question 1

Why does insertion sort perform faster than bubble sort on nearly sorted data?

Question 2

What is the fundamental difference between finding the minimum in selection sort and swapping adjacent elements in bubble sort?

Question 3

Which of the algorithms discussed are stable, and what does this mean in practice?

Hint: Consider the number of comparisons and swaps in each algorithm, as well as how they handle already ordered data segments.

Additional Tasks

1

Basic Implementation

Implement all three sorting algorithms in C++ and test them on arrays of different sizes and content.

2

Performance Measurement

Compare the execution time of the algorithms on arrays of 1000 and 10000 elements, and plot time vs. size graphs.

3

Optimization

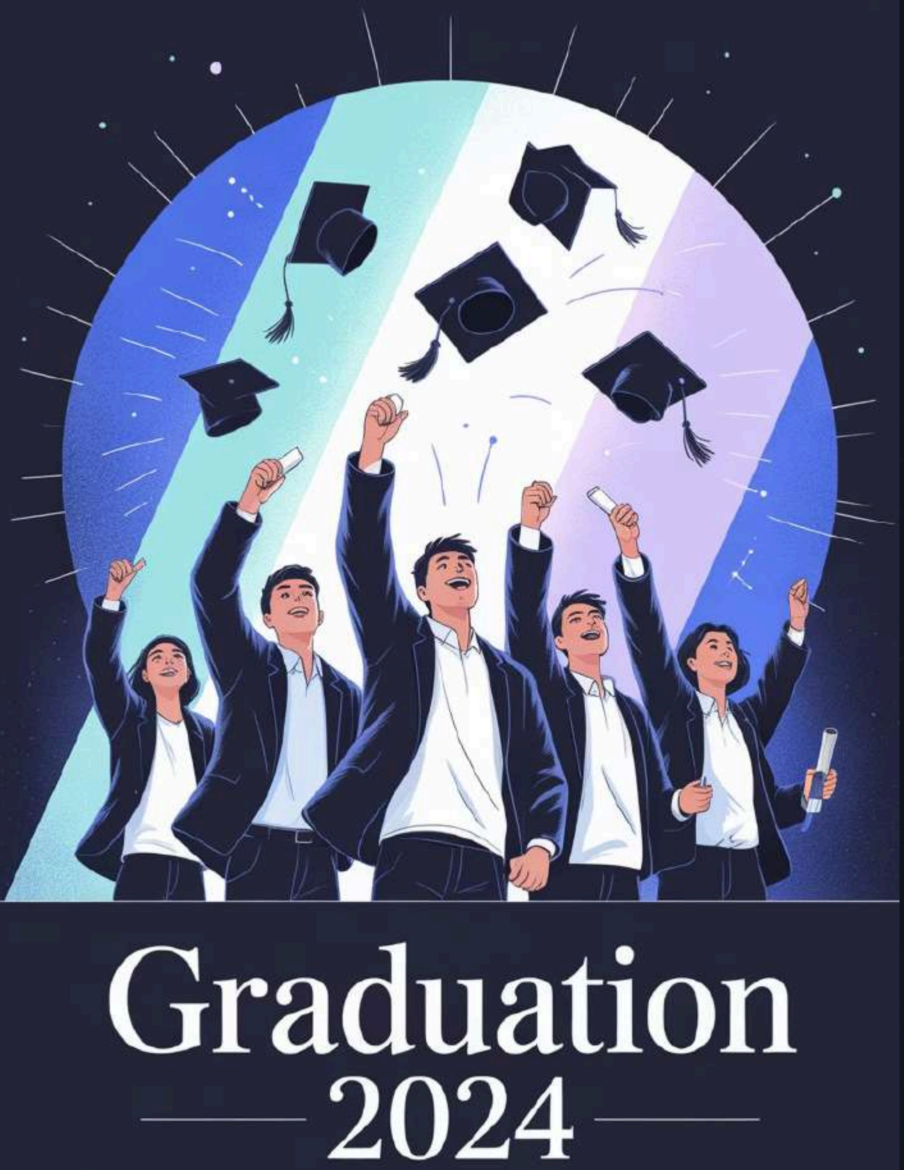
Add an early exit flag optimization to bubble sort and measure the performance improvement.

4

Stability Analysis

Create an array with duplicate elements and check which algorithms preserve the original order of equal elements.

These tasks will help you better understand the features of each algorithm and gain practical experience in their application.



Thank you for your attention!

Learned the Basics

Three classic sorting algorithms and their characteristics

Understood Complexity

Time and space characteristics of algorithms

Ready for Practice

Implementation and testing of algorithms in C++

> *"Simplicity is the ultimate sophistication."*
> Studying simple sorting algorithms lays a strong foundation for understanding more complex data processing methods.

Next Topic: Fast Sorting Algorithms (Quick Sort, Merge Sort, Heap Sort)