Selection Sort, Insertion Sort, and Bubble Sort

Introduction to Sorting

Sorting is the process of arranging data elements in a specific order, usually ascending or descending. It is one of the most fundamental operations in computer science because many applications require data to be organized before processing.

Efficient sorting improves the performance of searching, data analysis, database management, and information retrieval systems. Many algorithms have been developed for sorting, each with different strengths and weaknesses.

Sorting algorithms can be classified based on their methodology, time complexity, memory usage, and stability. Selection Sort, Insertion Sort, and Bubble Sort are among the simplest sorting algorithms and are commonly taught as introductory sorting techniques.

Selection Sort

Selection Sort is a simple comparison-based sorting algorithm. It repeatedly finds the smallest element from the unsorted portion of the list and places it in its correct position in the sorted portion.

The algorithm divides the array into two parts:

1. Sorted portion
2. Unsorted portion

During each pass, the minimum element from the unsorted section is selected and swapped with the first unsorted element.

Although Selection Sort is easy to understand and implement, it is inefficient for large datasets because it always performs many comparisons regardless of the initial arrangement of data.

Working Principle of Selection Sort

The algorithm follows these steps:

1. Assume the first element is the minimum.
2. Compare it with all remaining elements.
3. Find the actual smallest element.

4. Swap it with the first position.
5. Move to the next position.
6. Repeat until the array becomes sorted.

Step-by-Step Process of Selection Sort

Suppose we have the array:

[29, 10, 14, 37, 13]

Pass 1

Current array:

[29, 10, 14, 37, 13]

Smallest element = 10

Swap 29 and 10

[10, 29, 14, 37, 13]

Sorted portion:

[10]

Pass 2

Remaining unsorted part:

[29, 14, 37, 13]

Smallest element = 13

Swap 29 and 13

[10, 13, 14, 37, 29]

Sorted portion:

[10, 13]

Pass 3

Remaining unsorted part:

[14, 37, 29]

Smallest element = 14

No swap required.

[10, 13, 14, 37, 29]

Pass 4

Remaining unsorted part:

[37, 29]

Smallest element = 29

Swap 37 and 29

[10, 13, 14, 29, 37]

Array is now sorted.

Pseudocode for Selection Sort

```
SELECTION-SORT(A)
for i = 0 to n-2
   minIndex = i
   for j = i+1 to n-1
      if A[j] < A[minIndex]
         minIndex = j
   swap(A[i], A[minIndex])
end
```

Time Complexity of Selection Sort

Case	Complexity

Best Case	$O(n^2)$
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

Explanation

Selection Sort always searches the entire unsorted portion to find the minimum element. Therefore, even if the array is already sorted, the same number of comparisons are performed.

Number of comparisons:

$(n-1)+(n-2)+(n-3)+...+1$

Which equals:

$n(n-1)/2$

Thus:

$O(n^2)$

Advantages of Selection Sort

- Easy to understand and implement.
- Requires very little additional memory.
- Performs fewer swaps than Bubble Sort.
- Suitable for small datasets.

Disadvantages of Selection Sort

- Slow for large datasets.
- Time complexity remains $O(n^2)$ even for sorted data.
- Not adaptive.
- Not suitable for real-time applications involving large amounts of data.

Insertion Sort

Insertion Sort works similarly to how people arrange playing cards in their hands. It builds the sorted array one element at a time by inserting each element into its proper position.

The algorithm assumes that the first element is already sorted and gradually inserts the remaining elements into the correct location within the sorted section.

Insertion Sort is considered efficient for small datasets and nearly sorted arrays.

Working Principle of Insertion Sort

The algorithm follows these steps:

1. Assume the first element is sorted.
2. Take the next element.
3. Compare it with elements in the sorted portion.
4. Shift larger elements one position to the right.
5. Insert the element in its correct position.
6. Repeat until all elements are processed.

Step-by-Step Process of Insertion Sort

Given:

[29, 10, 14, 37, 13]

Pass 1

Key = 10

Compare with 29

Shift 29 right

Insert 10

[10, 29, 14, 37, 13]

Pass 2

Key = 14

Compare with 29

Shift 29

Insert 14

[10, 14, 29, 37, 13]

Pass 3

Key = 37

Already larger than 29

No shifting required.

[10, 14, 29, 37, 13]

Pass 4

Key = 13

Compare with 37

Shift 37

Compare with 29

Shift 29

Compare with 14

Shift 14

Insert 13

[10, 13, 14, 29, 37]

Array becomes sorted.

Pseudocode for Insertion Sort

```
INSERTION-SORT(A)
for i = 1 to n-1
   key = A[i]
   j = i - 1
   while j >= 0 and A[j] > key
      A[j+1] = A[j]
```

```
        j = j - 1

    A[j+1] = key

end
```

Time Complexity of Insertion Sort

Case	Complexity
Best Case	O(n)
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

Best Case

Array already sorted:

[1,2,3,4,5]

Only one comparison per element.

Complexity:

O(n)

Worst Case

Array sorted in reverse order:

[5,4,3,2,1]

Every element must be shifted.

Complexity:

$O(n^2)$

Advantages of Insertion Sort

- Easy to implement.
- Efficient for small datasets.
- Adaptive to nearly sorted data.

- Stable sorting algorithm.
- Requires minimal memory.

Disadvantages of Insertion Sort

- Inefficient for large datasets.
- Worst-case complexity is $O(n^2)$.
- Excessive shifting operations for large arrays.

Bubble Sort

Bubble Sort repeatedly compares adjacent elements and swaps them if they are in the wrong order.

The largest element "bubbles" toward the end of the array after each pass, which is why the algorithm is called Bubble Sort.

Bubble Sort is one of the simplest sorting algorithms but is rarely used in practical applications due to poor efficiency.

Working Principle of Bubble Sort

1. Compare adjacent elements.
2. Swap them if they are out of order.
3. Continue through the array.
4. Repeat passes until no swaps occur.

Example of Bubble Sort

Initial array:

[5, 3, 8, 4]

Pass 1:

5 > 3 → Swap
[3,5,8,4]

8 > 4 → Swap
[3,5,4,8]

Pass 2:

5 > 4 → Swap
[3,4,5,8]

Array sorted.

Pseudocode for Bubble Sort

```
BUBBLE-SORT(A)

for i = 0 to n-2

   for j = 0 to n-i-2

      if A[j] > A[j+1]

         swap(A[j], A[j+1])

end
```

Time Complexity of Bubble Sort

Case	Complexity
Best Case	O(n)
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

Comparison of Selection Sort, Insertion Sort, and Bubble Sort

Feature	Selection Sort	Insertion Sort	Bubble Sort
Method	Select minimum element	Insert element into sorted part	Swap adjacent elements
Best Case	$O(n^2)$	O(n)	O(n)
Average Case	$O(n^2)$	$O(n^2)$	$O(n^2)$
Worst Case	$O(n^2)$	$O(n^2)$	$O(n^2)$
Stable	No	Yes	Yes
Adaptive	No	Yes	Yes
Number of Swaps	Fewest	Moderate	Highest
Extra Memory	O(1)	O(1)	O(1)
Suitable for Small Data	Yes	Yes	Yes

Suitable for Nearly Sorted Data	Poor	Excellent	Good

Comparison with Advanced Sorting Algorithms

Algorithm	Best Case	Average Case	Worst Case
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$
Merge Sort	O(n log n)	O(n log n)	O(n log n)
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$
Heap Sort	O(n log n)	O(n log n)	O(n log n)

Advanced algorithms such as Merge Sort, Quick Sort, and Heap Sort are significantly more efficient for large datasets. However, Selection Sort, Insertion Sort, and Bubble Sort remain important because they introduce fundamental sorting concepts and are easier to learn and implement.

1. Selection Sort C++ Implementation

Idea

Selection sort repeatedly finds the **minimum element** and places it at the correct position.

```
#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]);
   }
}

void printArray(int arr[], int n) {
```

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

int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr) / sizeof(arr[0]);

    selectionSort(arr, n);

    cout << "Sorted Array (Selection Sort): ";
    printArray(arr, n);

    return 0;
}
```

2. Insertion Sort C++ Implementation

Idea

Insertion sort builds the sorted array one element at a time by inserting each element into its correct position.

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

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;
    }
}

void printArray(int arr[], int n) {
```

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

int main() {
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);

    insertionSort(arr, n);

    cout << "Sorted Array (Insertion Sort): ";
    printArray(arr, n);

    return 0;}
```

3. Bubble Sort C++ Implementation

Idea

Bubble sort repeatedly swaps adjacent elements if they are in wrong order. Largest elements "bubble" to the end.

```
#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;
    }
}
```

```
void printArray(int arr[], int n) {
   for (int i = 0; i < n; i++)
      cout << arr[i] << " ";
   cout << endl;
}

int main() {
   int arr[] = {5, 1, 4, 2, 8};
   int n = sizeof(arr) / sizeof(arr[0]);

   bubbleSort(arr, n);

   cout << "Sorted Array (Bubble Sort): ";
   printArray(arr, n);

   return 0;
}
```

Summary

Selection Sort repeatedly selects the smallest element and places it in its correct position. It is simple but always performs $O(n^2)$ comparisons.

Insertion Sort builds a sorted portion of the array by inserting elements into their correct positions. It is efficient for small and nearly sorted datasets and has a best-case complexity of $O(n)$.

Bubble Sort repeatedly swaps adjacent elements until the array is sorted. While simple to understand, it generally performs more swaps than Selection Sort and is less efficient for large datasets.

Among these three algorithms, **Insertion Sort is usually considered the most practical and efficient for small or nearly sorted datasets**, while **Selection Sort minimizes swaps**, and **Bubble Sort is primarily used for educational purposes.**