C++ Sorting Algorithms

Implementation and Visualization

Mr. Gullo

January 4, 2026

Learning Objectives

After this presentation, you will:

- Understand five different sorting algorithms

Learning Objectives

After this presentation, you will:

- Understand five different sorting algorithms
- Be able to implement each sorting algorithm in C++

Learning Objectives

After this presentation, you will:

- Understand five different sorting algorithms
- Be able to implement each sorting algorithm in C++
- Know the advantages and disadvantages of each method

Learning Objectives

After this presentation, you will:

- Understand five different sorting algorithms
- Be able to implement each sorting algorithm in C++
- Know the advantages and disadvantages of each method
- Recognize the time complexity of different algorithms

Helper Function: swap()

Why We Need This

Sorting requires swapping elements. We'll build our own swap function.

```
void swap(int &a, int &b) {
    int temp = a;    // Save a's value
    a = b;           // Overwrite a with b
    b = temp;        // Put saved value in b
}
```

Helper Function: swap()

Why We Need This

Sorting requires swapping elements. We'll build our own swap function.

```
void swap(int &a, int &b) {
    int temp = a;    // Save a's value
    a = b;           // Overwrite a with b
    b = temp;        // Put saved value in b
}
```

The & Means Pass-by-Reference

Without &, changes stay inside the function.
With &, we modify the **original** variables.

Initial Array

Numbers to Sort

Our example will use these 9 numbers: [7, 2, 9, 4, 5, 8, 3, 6, 10]

Bubble Sort: The Code

Algorithm Description

- Compares adjacent elements and swaps if needed

Bubble Sort: The Code

Algorithm Description

- Compares adjacent elements and swaps if needed
- Largest element "bubbles" to the end each pass

Bubble Sort: The Code

Algorithm Description

- Compares adjacent elements and swaps if needed
- Largest element ”bubbles” to the end each pass

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

Bubble Sort: Trace Through

Let's Trace: arr = [5, 3, 8, 1]

Pass 1: Compare adjacent, swap if needed

,8,1

$\rightarrow$ [**3,5**,8,1] $\rightarrow$ [3,5,8,1] $\rightarrow$ [3,5,**1,8**]

Bubble Sort: Trace Through

Let's Trace: arr = [5, 3, 8, 1]

Pass 1: Compare adjacent, swap if needed

,8,1

→ [**3,5**,8,1] → [3,5,8,1] → [3,5,**1,8**]

Pass 2: 8 is in place, work on first 3

,1,8

→ [3,5,1,8] → [3,**1,5**,8]

Bubble Sort: Trace Through

Let's Trace: arr = [5, 3, 8, 1]

Pass 1: Compare adjacent, swap if needed
,8,1

$\rightarrow$ [**3**,**5**,8,1] $\rightarrow$ [3,5,8,1] $\rightarrow$ [3,5,**1**,**8**]

Pass 2: 8 is in place, work on first 3
,1,8

$\rightarrow$ [3,5,1,8] $\rightarrow$ [3,**1**,**5**,8]

Pass 3: [3,1,5,8] $\rightarrow$ [**1**,**3**,5,8] ✓

Bubble Sort Animation

6 5 3 1 8 7 2 4

Watch

Adjacent elements swap until largest "bubbles" to the end.

Selection Sort: The Code

Algorithm Description

- Find minimum in unsorted region

Selection Sort: The Code

Algorithm Description

- Find minimum in unsorted region
- Swap it to the front of unsorted region

Selection Sort: The Code

Algorithm Description

- Find minimum in unsorted region
- Swap it to the front of unsorted region

```c
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_idx = i;                     // Assume first is min
        for (int j = i+1; j < n; j++) // Search rest
            if (arr[j] < arr[min_idx])
                min_idx = j;                 // Found smaller
        swap(arr[min_idx], arr[i]);      // Move min to front
    }
}
```

Selection Sort: Trace Through

Let's Trace: arr = [5, 3, 8, 1]

i=0: Find min in [5,3,8,1] → min_idx=3 (value 1)
Swap arr[3] with arr[0]: [**1**,3,8,**5**]

Selection Sort: Trace Through

Let's Trace: arr = [5, 3, 8, 1]

i=0: Find min in [5,3,8,1] → min_idx=3 (value 1)
Swap arr[3] with arr[0]: [**1**,3,8,**5**]

i=1: Find min in [3,8,5] → min_idx=1 (value 3)
Swap arr[1] with arr[1]: [1,**3**,8,5] (no change)

Selection Sort: Trace Through

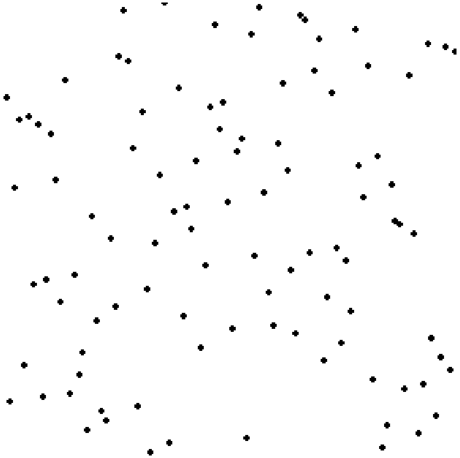
Let's Trace: arr = [5, 3, 8, 1]

i=0: Find min in [5,3,8,1] → min_idx=3 (value 1)
Swap arr[3] with arr[0]: [**1**,3,8,**5**]

i=1: Find min in [3,8,5] → min_idx=1 (value 3)
Swap arr[1] with arr[1]: [1,**3**,8,5] (no change)

i=2: Find min in [8,5] → min_idx=3 (value 5)
Swap arr[3] with arr[2]: [1,3,**5**,**8**] ✓

Selection Sort Animation



Watch

Find the minimum, swap to front. Repeat for remaining unsorted portion.

Insertion Sort: The Code

Algorithm Description

- Take element, slide it left until correct position

Insertion Sort: The Code

Algorithm Description

- Take element, slide it left until correct position
- Like sorting cards in your hand

Insertion Sort: The Code

Algorithm Description

- Take element, slide it left until correct position
- Like sorting cards in your hand

```
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];          // Element to
           insert
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];   // Shift right
            j--;
        }
        arr[j + 1] = key;          // Insert in gap
    }
}
```

Insertion Sort: Trace Through

Let's Trace: arr = [5, 3, 8, 1]

i=1: key=3, compare with 5, shift 5 right
[5,$\underline{5}$,8,1] $\rightarrow$ insert key: [**3**,5,8,1]

Insertion Sort: Trace Through

Let's Trace: arr = [5, 3, 8, 1]

i=1: key=3, compare with 5, shift 5 right
[5,<u>5</u>,8,1] $\rightarrow$ insert key: [**3**,5,8,1]

i=2: key=8, compare with 5, 8¿5 so stop
[3,5,**8**,1] (no shifts needed)

Insertion Sort: Trace Through

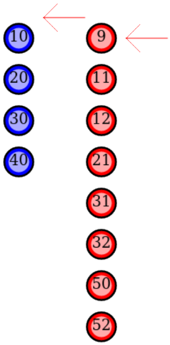
Let's Trace: arr = [5, 3, 8, 1]

i=1: key=3, compare with 5, shift 5 right
[5,<u>5</u>,8,1] $\rightarrow$ insert key: [**3**,5,8,1]

i=2: key=8, compare with 5, 8¿5 so stop
[3,5,**8**,1] (no shifts needed)

i=3: key=1, shift 8,5,3 all right
[3,5,8,<u>8</u>] $\rightarrow$ [3,5,<u>5</u>,8] $\rightarrow$ [3,<u>3</u>,5,8] $\rightarrow$ [**1**,3,5,8] ✓

Insertion Sort Animation



Merge Sort: The Concept

Algorithm Description

- Divide and conquer approach

Merge Sort: The Concept

Algorithm Description

- Divide and conquer approach
- Splits array in half recursively

Merge Sort: The Concept

Algorithm Description

- Divide and conquer approach
- Splits array in half recursively
- Merges sorted halves back together

Merge Sort: The Concept

Algorithm Description

- Divide and conquer approach
- Splits array in half recursively
- Merges sorted halves back together

Time complexity: O(n log n) - consistent performance

Merge Sort: The Merge Function

```c
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int L[n1], R[n2];  // Temp arrays

    for (int i = 0; i < n1; i++) L[i] = arr[left + i];
    for (int j = 0; j < n2; j++) R[j] = arr[mid + 1 +
        j];

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2)
        arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}
```

Merge Sort: The Recursive Function

```
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid);        // Sort left half
        mergeSort(arr, mid + 1, right); // Sort right half
        merge(arr, left, mid, right);    // Merge sorted halves
    }
}
```

Merge Sort: The Recursive Function

```
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid);        // Sort left
            half
        mergeSort(arr, mid + 1, right); // Sort right
            half
        merge(arr, left, mid, right);   // Merge
            sorted halves
    }
}
```

Let's Trace: arr = [7, 2, 9, 4]

Split: [7,2] and [9,4] $\rightarrow$ [2,7] and [4,9] $\rightarrow$ [2,4,7,9]

Merge Sort Animation

Watch

Split, sort, merge. The "divide and conquer" strategy in action.

Quick Sort: The Concept

Algorithm Description

- Choose a pivot element

Quick Sort: The Concept

Algorithm Description

- Choose a pivot element
- Partition: smaller elements left, larger elements right

Quick Sort: The Concept

Algorithm Description

- Choose a pivot element
- Partition: smaller elements left, larger elements right
- Recursively sort the partitions

Quick Sort: The Concept

Algorithm Description

- Choose a pivot element
- Partition: smaller elements left, larger elements right
- Recursively sort the partitions

Time complexity: $O(n \log n)$ average, $O(n^2)$ worst case

Quick Sort: The Partition Function

```cpp
int partition(int arr[], int low, int high) {
    int pivot = arr[high];  // Choose last element as pivot
    int i = low - 1;        // Index of smaller element

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;  // Return pivot's final position
}
```

Quick Sort: The Partition Function

```
int partition(int arr[], int low, int high) {
    int pivot = arr[high];  // Choose last element as pivot
    int i = low - 1;        // Index of smaller element

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;  // Return pivot's final position
}
```

Note: Lomuto vs Hoare

This is **Lomuto partition** (simpler). **Hoare partition** uses two pointers

Quick Sort: The Recursive Function

```
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);    // Sort left of pivot
        quickSort(arr, pi + 1, high);  // Sort right of pivot
    }
}
```

Quick Sort: The Recursive Function

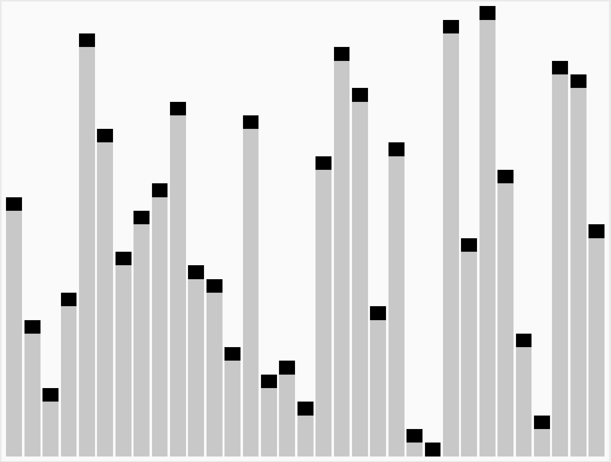
```
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);    // Sort left of pivot
        quickSort(arr, pi + 1, high);  // Sort right of pivot
    }
}
```

Let's Trace: arr = [7, 2, 9, 4]

Pivot=4: [2] 4 [7,9] → [2] 4 [7] 9 → [2,4,7,9]

Quick Sort Animation



Watch

Pick pivot, partition, recurse. Generally the fastest in practice.

Time Complexity Comparison

Time Complexity

- Bubble Sort: $O(n^2)$

Time Complexity Comparison

Time Complexity

- Bubble Sort: $O(n^2)$
- Selection Sort: $O(n^2)$

Time Complexity

- Bubble Sort: $O(n^2)$
- Selection Sort: $O(n^2)$
- Insertion Sort: $O(n^2)$

Time Complexity Comparison

Time Complexity

- Bubble Sort: $O(n^2)$
- Selection Sort: $O(n^2)$
- Insertion Sort: $O(n^2)$
- Quick Sort: $O(n \log n)$ average, $O(n^2)$ worst case

Time Complexity Comparison

Time Complexity

- Bubble Sort: $O(n^2)$
- Selection Sort: $O(n^2)$
- Insertion Sort: $O(n^2)$
- Quick Sort: $O(n \log n)$ average, $O(n^2)$ worst case
- Merge Sort: $O(n \log n)$

Summary

Key Points

- Bubble Sort: Simple but inefficient

Summary

Key Points

- Bubble Sort: Simple but inefficient
- Selection Sort: Simple and performs well on small lists

Summary

Key Points

- Bubble Sort: Simple but inefficient
- Selection Sort: Simple and performs well on small lists
- Insertion Sort: Efficient for small data sets

Summary

Key Points

- Bubble Sort: Simple but inefficient
- Selection Sort: Simple and performs well on small lists
- Insertion Sort: Efficient for small data sets
- Quick Sort: Generally the fastest in practice

Summary

Key Points

- Bubble Sort: Simple but inefficient
- Selection Sort: Simple and performs well on small lists
- Insertion Sort: Efficient for small data sets
- Quick Sort: Generally the fastest in practice
- Merge Sort: Consistent performance but requires extra space