Sorting Algorithms

Data Structures and Algorithms

Masoud Hamad

State University of Zanzibar (SUZA)

2025/2026 Academic Year

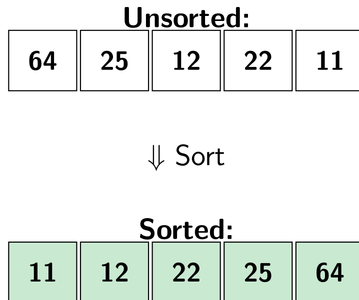
Outline

1. Introduction to Sorting
2. Bubble Sort
3. Selection Sort
4. Insertion Sort
5. Merge Sort
6. Quick Sort
7. Comparison of Sorting Algorithms
8. Summary

Why Sorting?

Sorting = arranging elements in a specific order (ascending or descending).

- Enables **binary search** ($O(\log n)$ vs $O(n)$)
- Makes data easier to **visualize and analyze**
- Speeds up **duplicate detection**
- Required for many algorithms (merge, median, etc.)
- Used in **databases**, **search engines**, **file systems**



Sorting Algorithm Properties

Time Complexity How many operations? $O(n^2)$ vs $O(n \log n)$

Space Complexity Extra memory needed? **In-place** $= O(1)$ extra

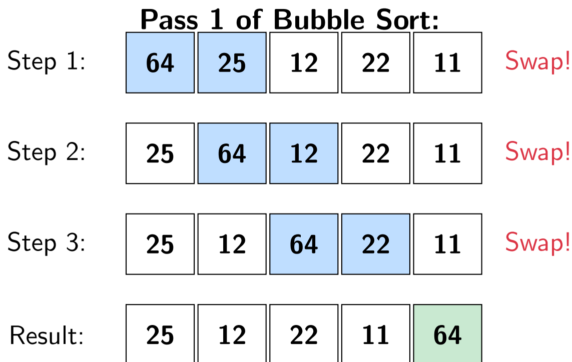
Stability Does it preserve order of equal elements?

Adaptivity Does it run faster on partially sorted data?

Algorithm	Best	Average	Worst	Stable?
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	No
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Yes
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	No

Bubble Sort – Idea

- Repeatedly **compare adjacent elements** and **swap** if out of order
- After each pass, the largest unsorted element "bubbles up" to its correct position
- Repeat until no swaps are needed



Bubble Sort – Code

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int swapped = 0;                    // optimization flag

        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap adjacent elements
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = 1;
            }
        }

        if (!swapped) break;  // array is already sorted
    }
}
```

Time Complexity

Best: $O(n)$ (already sorted)

Worst/Average: $O(n^2)$

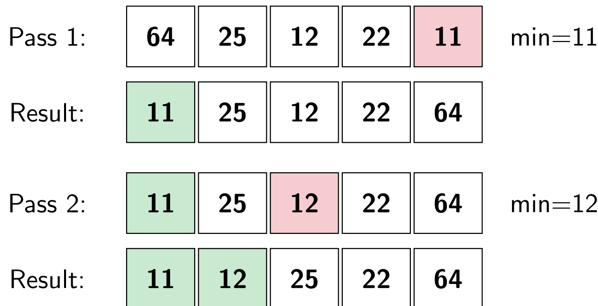
Properties

In-place: Yes ($O(1)$ extra)

Stable: Yes

Selection Sort – Idea

- Find the **minimum** element in the unsorted portion
- **Swap** it with the first unsorted element
- Move the boundary of sorted portion one step right
- Repeat until entire array is sorted



Selection Sort – Code

```
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int minIdx = i;

        // Find minimum in unsorted portion
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIdx])
                minIdx = j;
        }

        // Swap minimum with first unsorted element
        if (minIdx != i) {
            int temp = arr[i];
            arr[i] = arr[minIdx];
            arr[minIdx] = temp;
        }
    }
}
```

Complexity

Always $O(n^2)$ comparisons. Only $O(n)$ swaps (good when swaps are expensive). Not stable.

Insertion Sort – Idea

- Like sorting **playing cards** in your hand
- Take each element and **insert it into its correct position** in the sorted portion
- Shift larger elements to the right to make room

Initial:	12	11	13	5	6
key=11:	11	12	13	5	6
key=13:	11	12	13	5	6
key=5:	5	11	12	13	6
key=6:	5	6	11	12	13

Insertion Sort – Code

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

        // Shift elements greater than key to the right
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }

        arr[j + 1] = key;          // insert key at correct position
    }
}
```

Best Case: $O(n)$

Already sorted array – inner loop never executes

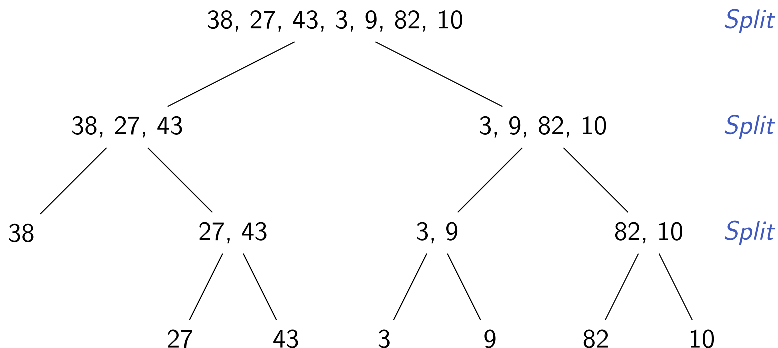
Worst Case: $O(n^2)$

Reverse sorted – every element shifted to front

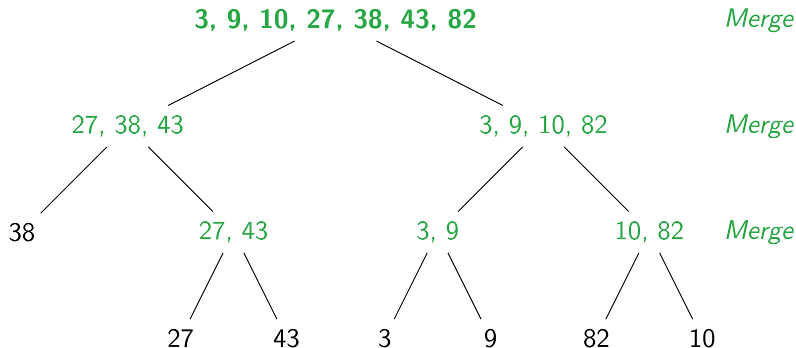
When to use Insertion Sort?

Merge Sort – Divide and Conquer

1. **Divide**: Split the array into two halves
2. **Conquer**: Recursively sort each half
3. **Combine**: Merge the two sorted halves



Merge Sort – Merge Phase



Key Insight

Merging two sorted arrays of size $n/2$ takes $O(n)$ time.
There are $O(\log n)$ levels of splitting.
Total: $O(n \log n)$ – **guaranteed, in all cases!**

Merge Sort – Code

```
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1, n2 = right - mid;
    int L[n1], R[n2];
    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) {
        if (L[i] <= R[j]) arr[k++] = L[i++];
        else               arr[k++] = R[j++];
    }
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}

void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
```

Quick Sort – Idea

1. Choose a **pivot** element
2. **Partition**: rearrange so that:
   - Elements $<$ pivot go to the **left**
   - Elements $>$ pivot go to the **right**
   - Pivot is in its **final position**
3. Recursively sort left and right partitions

Before:	**10**	**7**	**8**	**9**	**1**	**5**	pivot=5
After:	**1**	**5**	**8**	**9**	**10**	**7**	

< 5 (left partition: 1) > 5 (right partition: 8, 9, 10, 7)

Quick Sort – Partition

```
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] and arr[j]
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }

    // Place pivot in correct position
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;

    return i + 1;  // return pivot index
}
```

Quick Sort – Main 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
    }
}

// Usage:
// quickSort(arr, 0, n - 1);
```

Average Case: $O(n \log n)$

Pivot divides array roughly in half each time

Worst Case: $O(n^2)$

Pivot is always min or max (sorted/reverse input)

In Practice

Quick Sort is often the **fastest** sorting algorithm due to good cache behavior and small

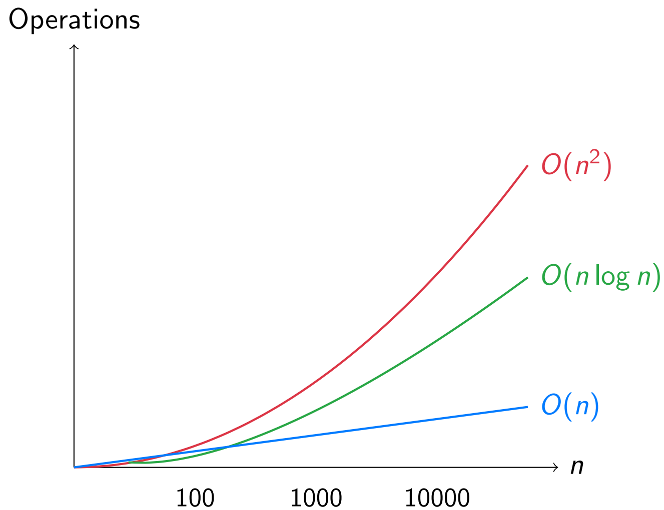
Comparison Summary

Algorithm	**Best**	**Avg**	**Worst**	**Space**	**Stable**	**In-place**
Bubble	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Yes
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No	Yes
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Yes
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes	No
Quick	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	No	Yes

Which to use?

- **Small arrays** ($n < 20$): Insertion Sort
- **Guaranteed** $O(n \log n)$: Merge Sort
- **General purpose, fastest in practice**: Quick Sort
- **Nearly sorted data**: Insertion Sort
- **Memory constrained**: Quick Sort or Insertion Sort

$O(n^2)$ vs $O(n \log n)$ – Visual Comparison



Practical Impact

Summary

- **Bubble Sort**: Compare adjacent, swap – simple but slow $O(n^2)$
- **Selection Sort**: Find min, place at front – always $O(n^2)$ comparisons
- **Insertion Sort**: Insert into sorted portion – great for small/nearly sorted data
- **Merge Sort**: Divide, sort halves, merge – guaranteed $O(n \log n)$, needs extra space
- **Quick Sort**: Partition around pivot – fastest in practice, $O(n \log n)$ average

What's Next?

Practice: Lab 5 (Sorting), Exercises, Homework 9 & 10
Code: `dsacode/sorting_algorithms.c`

Questions?

Try the benchmark: `dsacode/sorting_algorithms.c` (option 9)