



Lab 3

Problem 1

A university is ranking students based on their exam scores. The scores must be sorted in **descending order**, maintaining **stability** (students with the same score stay in their original order).

What is Stability in Sorting?

- **Definition:** A sorting algorithm is **stable** if it preserves the relative order of equal elements after sorting.
- **Example:**
 - **Unsorted:** `[("Alice", 85), ("Charlie", 85)]`
 - **Stable Sort Output:** `[("Alice", 85), ("Charlie", 85)]`  (Bob remains before Charlie)
 - **Unstable Sort Output:** `[("Charlie", 85), ("Alice", 85)]`  (Order changed, unfair in ranking systems)

You are given a list of students with their scores:

```
students = [("Alice", 85), ("Bob", 90), ("Charlie", 85), ("David", 92), ("Eve", 90)]
```

Explain **step by step** how **Insertion Sort** sorts this list.

1. **Describe each iteration** of Insertion Sort, showing comparisons, shifts, and insertions.
2. **Show the list's state** after each pass.

Answer

Step-by-Step Execution:

- **Pass 1 (Bob, 90) is inserted at the correct position**
 - Compare "Bob" (90) with "Alice" (85).
 - Since $90 > 85$, shift "Alice" and insert "Bob" before "Alice".

List after Pass 1: `[("Bob", 90), ("Alice", 85), ("Charlie", 85), ("David", 92), ("Eve", 90)]`

- **Pass 2 (Charlie, 85) is inserted at the correct position**

- Compare "Charlie" (85) with "Alice" (85).
- Since `85 == 85`, **no change** (stability is maintained).

List after Pass 2: `[("Bob", 90), ("Alice", 85), ("Charlie", 85), ("David", 92), ("Eve", 90)]`

- **Pass 3 (David, 92) is inserted at the correct position**
 - Compare "David" (92) with "Charlie" (85) → Shift "Charlie".
 - Compare "David" (92) with "Alice" (85) → Shift "Alice".
 - Compare "David" (92) with "Bob" (90) → Shift "Bob".
 - "David" is inserted at the beginning.

List after Pass 3: `[("David", 92), ("Bob", 90), ("Alice", 85), ("Charlie", 85), ("Eve", 90)]`

- **Pass 4 (Eve, 90) is inserted at the correct position**
 - Compare "Eve" (90) with "Charlie" (85) → Shift "Charlie".
 - Compare "Eve" (90) with "Alice" (85) → Shift "Alice".
 - Compare "Eve" (90) with "Bob" (90) → **No shift** (stability maintained).
 - "Eve" is placed **after "Bob" and before "Alice"**.

List after Pass 4 (Final Sorted List): `[("David", 92), ("Bob", 90), ("Eve", 90), ("Alice", 85), ("Charlie", 85)]`

Problem 2

You are managing an **e-commerce platform** and need to display the **top K cheapest products** to customers. Instead of sorting the entire product list, you only need to identify the **K lowest-priced items efficiently**.

You are given the following product prices:

`prices = [299, 150, 89, 199, 49, 120]`

For **K = 3**, the three cheapest products should be:

`[49, 89, 120]`

1. **Explain step by step** how Selection Sort is used to find the **K smallest prices** without sorting the entire list.

2. **Show the intermediate states** of the list after each iteration until the K smallest elements are selected.

Answer

Step-by-Step Execution Using Selection Sort

Selection Sort works by selecting the smallest element and swapping it to its correct position. Instead of sorting the entire list, we only run the first K iterations to find the K smallest elements.

- **Pass 1: Finding the 1st Smallest Price**
 - Scan the entire list to find the **smallest element**.
 - **Smallest value:** 49
 - Swap 49 with the first element (299).

Updated list after Pass 1: [49, 150, 89, 199, 299, 120]

- **Pass 2: Finding the 2nd Smallest Price**
 - Scan the remaining unsorted portion [150, 89, 199, 299, 120].
 - **Smallest value:** 89
 - Swap 89 with 150 (second element).

Updated list after Pass 2: [49, 89, 150, 199, 299, 120]

- **Pass 3: Finding the 3rd Smallest Price**
 - Scan the remaining unsorted portion [150, 199, 299, 120].
 - **Smallest value:** 120
 - Swap 120 with 150 (third element).

Updated list after Pass 3: [49, 89, 120, 199, 299, 150]

Final Selected Prices: [49, 89, 120]

We stop after $K = 3$ iterations instead of fully sorting the list, improving efficiency.

- Standard Selection Sort sorts the entire list ($O(n^2)$).
- By stopping after K iterations, we reduce unnecessary comparisons, improving efficiency to $O(nK)$.
- Useful for ranking scenarios where only a few top elements are needed.

Problem 3

A **restaurant review platform** wants to display customer ratings **in ascending order** to help users easily find the lowest-rated restaurants. However, the ratings are currently **unsorted**, and the platform is using **Bubble Sort** to process them.

You are given the following **restaurant ratings (out of 5 stars)**:

```
ratings = [4.5, 3.2, 5.0, 2.8, 4.0, 3.8]
```

After sorting, the ratings should appear as:

```
[2.8, 3.2, 3.8, 4.0, 4.5, 5.0]
```

1. **Explain step by step** how Bubble Sort sorts this list, showing each full pass and the swaps that occur.
2. How can **Bubble Sort be optimized** for a nearly sorted list? Analyze the time complexity.

Answer

Standard Bubble Sort Execution

Bubble Sort works by repeatedly **swapping adjacent elements** if they are in the wrong order. This continues until the list is fully sorted.

Pass 1: Move the largest element to the end

- Compare 4.5 and 3.2 → **Swap** → [3.2, 4.5, 5.0, 2.8, 4.0, 3.8]
- Compare 4.5 and 5.0 → No swap
- Compare 5.0 and 2.8 → **Swap** → [3.2, 4.5, 2.8, 5.0, 4.0, 3.8]
- Compare 5.0 and 4.0 → **Swap** → [3.2, 4.5, 2.8, 4.0, 5.0, 3.8]
- Compare 5.0 and 3.8 → **Swap** → [3.2, 4.5, 2.8, 4.0, 3.8, 5.0]

State after Pass 1: [3.2, 4.5, 2.8, 4.0, 3.8, 5.0]

Pass 2: Move the second-largest element to its correct position

- Compare 3.2 and 4.5 → No swap
- Compare 4.5 and 2.8 → **Swap** → [3.2, 2.8, 4.5, 4.0, 3.8, 5.0]
- Compare 4.5 and 4.0 → **Swap** → [3.2, 2.8, 4.0, 4.5, 3.8, 5.0]

- Compare 4.5 and 3.8 → **Swap** → [3.2, 2.8, 4.0, 3.8, 4.5, 5.0]

State after Pass 2: [3.2, 2.8, 4.0, 3.8, 4.5, 5.0]

Pass 3: Move the third-largest element to its correct position

- Compare 3.2 and 2.8 → **Swap** → [2.8, 3.2, 4.0, 3.8, 4.5, 5.0]
- Compare 3.2 and 4.0 → No swap
- Compare 4.0 and 3.8 → **Swap** → [2.8, 3.2, 3.8, 4.0, 4.5, 5.0]

State after Pass 3: [2.8, 3.2, 3.8, 4.0, 4.5, 5.0]

At this point, the list is already sorted, but **Bubble Sort will continue checking** for two more passes, even though no swaps are needed.

Pass 4: Move the fourth-largest element to its correct position

- Compare 2.8 and 3.2 → No swap
- Compare 3.2 and 3.8 → No swap

State after Pass 4: [2.8, 3.2, 3.8, 4.0, 4.5, 5.0]

Pass 5: Move the fifth-largest element to its correct position

- Compare 2.8 and 3.2 → No swap

State after Pass 5: [2.8, 3.2, 3.8, 4.0, 4.5, 5.0]

Optimized Bubble Sort (Stopping Early)

- **Issue with Standard Bubble Sort:** Runs $n-1$ passes even if already sorted, causing unnecessary comparisons.
- **Optimization:** Use a swapped flag:
 - Set `swapped = False` before each pass.
 - If any swap occurs, set `swapped = True`.
 - If no swaps occur, stop early (list is sorted).
- **Efficiency Gain:**
 - Prevents redundant iterations, reducing passes.
 - Best-case complexity improves from $O(n^2)$ to $O(n)$.
 - Ideal for sorted or nearly sorted lists, making Bubble Sort more efficient.

Problem 4

Different sorting algorithms perform **differently** depending on the structure of the input data. Your task is to **analyze and compare** the behavior of sorting algorithms when applied to different types of input.

Given the following input cases:

1. **Nearly sorted array:** [1, 2, 3, 4, 6, 5, 7, 8, 9, 10]
2. **Completely reversed array:** [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
3. **Array with many duplicate elements:** [4, 2, 2, 8, 3, 3, 3, 7, 4, 2]

Which **sorting algorithm (Bubble Sort, Selection Sort, Insertion Sort, Merge Sort)** performs **best and worst** for each input type? Explain why.

Answer

Case 1: Nearly Sorted Array

Example: [1, 2, 3, 4, 6, 5, 7, 8, 9, 10]

Best Algorithm: Insertion Sort

- Insertion Sort is highly efficient when the array is nearly sorted.
- **Few swaps needed:** It only moves elements that are out of place. In this case, only 6 and 5 need to be adjusted, minimizing the number of operations.
- **Reaches $O(n)$ complexity:** In the best case (fully sorted), each element is compared once and no swaps are required.

Worst Algorithm: Selection Sort

- Selection Sort **always** performs $O(n^2)$ comparisons, regardless of input order.
- **Inefficient for nearly sorted input:** Even if only one element is misplaced, it still scans the entire list in every iteration.

Case 2: Completely Reversed Array

Example: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

Best Algorithm: Merge Sort

- Merge Sort performs $O(n \log n)$ operations consistently, regardless of whether the array is sorted or reversed.
- **Efficient for worst-case scenarios:** Unlike Insertion Sort and Bubble Sort, which perform poorly on reversed data, Merge Sort divides and processes subarrays efficiently.

Worst Algorithm: Bubble Sort/Insertion Sort

- Every element must be swapped into its correct position, leading to $O(n^2)$ swaps.
- **Highly inefficient:** The maximum number of swaps and comparisons occur when the list is in descending order.

Case 3: Array with Many Duplicates

Example: [4, 2, 2, 8, 3, 3, 3, 7, 4, 2]

Best Algorithm: Merge Sort

- Merge Sort maintains $O(n \log n)$ efficiency and does not depend on unique values.
- **Efficiently handles repeated elements:** Sorting performance remains consistent, even when duplicate values are present.

Worst Algorithm: Selection Sort

- Selection Sort **compares every element $O(n^2)$ times**, making it inefficient when sorting a dataset with repeated values.
- **Unnecessary operations:** Even with duplicates, the algorithm does not improve its performance by recognizing similar values.

Problem 5

An e-commerce platform processes thousands of orders daily. Each order contains a **unique order ID** and a **total purchase amount**. To generate financial reports efficiently, the orders need to be **sorted in ascending order by total purchase amount** before analysis.

Given a list of **unsorted orders**, you must choose and explain the steps of a **sorting algorithm with $O(n \log n)$ complexity** to efficiently organize the data. **Describe step by step** how the sorting algorithm processes the list.

Answer

Using Merge Sort

- This sorting algorithm follows a **divide-and-conquer** approach.
- The list is **recursively divided into smaller sublists** until each contains only one element.
- Then, the sublists are **merged back together in sorted order**.

Example: orders = [320, 150, 450, 280, 500, 200, 100]

The goal is to **sort these orders in ascending order** using a **sorting algorithm with $O(n \log n)$ complexity**.

Step-by-Step Sorting Execution

1. Splitting the List (Divide Phase)

The list is **split into two halves repeatedly** until each sublist contains only one element.

Original list: [320, 150, 450, 280, 500, 200, 100]

First Split

Left: [320, 150, 450, 280]

Right: [500, 200, 100]

Second Split

Left (split again): [320, 150] | [450, 280]

Right (split again): [500, 200] | [100]

Final Split into Single Elements

[320] [150] [450] [280] [500] [200] [100]

2. Merging the Sorted Sublists (Conquer Phase)

After dividing, the algorithm **merges sublists back together in sorted order**.

Merging the smallest elements first

[320] and [150] → [150, 320]

[450] and [280] → [280, 450]

[500] and [200] → [200, 500]

Merging the next level of sublists

[150, 320] and [280, 450] → [150, 280, 320, 450]

[200, 500] and [100] → [100, 200, 500]

Final Merge (Sorted List)

[150, 280, 320, 450] and [100, 200, 500] → [100, 150, 200, 280, 320, 450, 500]

Final Sorted Order:

[100, 150, 200, 280, 320, 450, 500]

Problem 6

You are developing a digital photo management system. Users can add new photos to their albums in two ways:

1. **A few new photos** are added to an already sorted album.
2. **A large batch of photos** is imported at once, often in random order.

Which sorting algorithm would be most suitable for **each scenario**, and why? Consider factors such as **time complexity, efficiency, and adaptability** to different input conditions.

Answer

Scenario 1: A Few New Photos Added to an Already Sorted Album

Best Algorithm: Insertion Sort

- **Efficient for nearly sorted data:** Insertion Sort performs in $O(n)$ time when only a few elements are out of place.
- It minimizes unnecessary comparisons and swaps.
- It doesn't require extra space, making it efficient for maintaining the sorted structure.

Scenario 2: A Large Batch of Photos Imported at Once (Unsorted Data)

Best Algorithm: Merge Sort

- **Handles large, unordered datasets efficiently:** Merge Sort guarantees $O(n \log n)$ time complexity in all cases.

- If photos have timestamps, Merge Sort maintains their original order. Without this, photos with identical timestamps could be **rearranged unpredictably**, disrupting the intended order in albums or galleries.