Faculty of Engineering – Ain Shams University

Computer Engineering and Software Systems

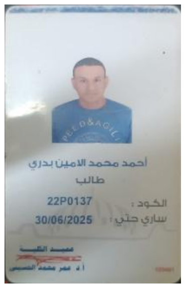
CSE331: Data Structure and Algorithms

Fall 2024

Submitted to

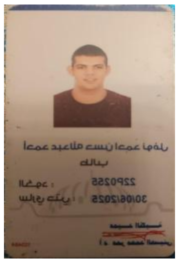
Dr. Hesham Farag

Submitted by


أحمد محمد الامين بدري
طالب
الكود : 22P0137
ساري حتي : 30/06/2025
عميد الكلية
أ.د عمر محمد الحسيني

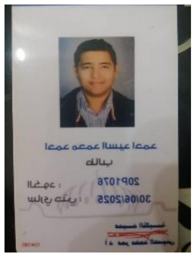
Student Name: **Ahmed Mohamed Alamin**

*Student ID (ASU):***22P0137**


أحمد عبدالله حسن عبدالله نوفل
طالب
الكود : 22P0255
ساري حتي : 30/06/2025

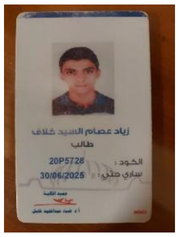
Student Name: **Ahmed Abdallah Hassan Abdallah Nofull**

Student ID (ASU): **22P0255**


أحمد محمد السيد أحمد
طالب
الكود : 20P1076
ساري حتي : 30/06/2025

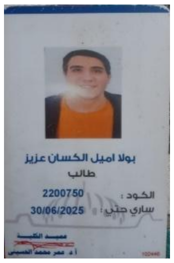
Student Name: **Ahmed Mohamed El Sayed Ahmed**

Student ID (ASU): **20P1076**


زياد عصام السيد خلاف
طالب
الكود : 20P5728
ساري حتي : 30/06/2025
عميد الكلية

Student Name: **Zeyad Essam El Sayed**

Student ID (ASU): **20P5728**


بولا اميل الكسان عزيز
طالب
الكود : 2200750
ساري حتي : 30/06/2025
عميد الكلية
أ.د عمر محمد الحسيني

Student Name: **Paula Emil Alexan Aziz**

Student ID (ASU): **2200750**

Introduction

First, we have used python 3 as a programming language. Our main goal is to plot the graph of steps of many important sorting algorithms like insertion sort, merge sort, heap sort, counting sort, quick Sort, bubble and radix Sort. Moreover, we will show the notation of each algorithm, compare between steps of these algorithms to know which is quicker and efficient. By making such comparison, the uses and advantages of each algorithm will be clearly displayed but first we need to discuss each algorithm and know how each of them works.

Sorting Algorithms

1- Insertion sort

Insertion Sort is a simple sorting algorithm that works similarly to how we sort playing cards in our hands. The array is virtually split into a sorted and an unsorted part, and the values from the unsorted part are picked and placed in the correct position in the sorted part.

How It Works

1. Start with the second element (index 1) in the array. Assume the first element (index 0) is already sorted.
2. Compare the current element to the elements before it in the sorted part of the array.
3. Shift all elements larger than the current element one position to the right.
4. Insert the current element into its correct position.

5. Repeat steps 2–4 for all remaining elements in the unsorted part of the array.

Steps in Action

Suppose we have an array

23, 1, 10, 5, 2

Step 1: First element (23) is considered sorted.

- Step 2: Next, take 1.
    - Compare 1 with 23, shift 23 to the right to become
    - 1, 23, 10, 5, 2

- Step 3: Take 10.
- Compare 10 with 23, shift 23 to the right, and then compare 10 with 1, 1 is smaller then keep it to become
- 1, 10, 23, 5, 2

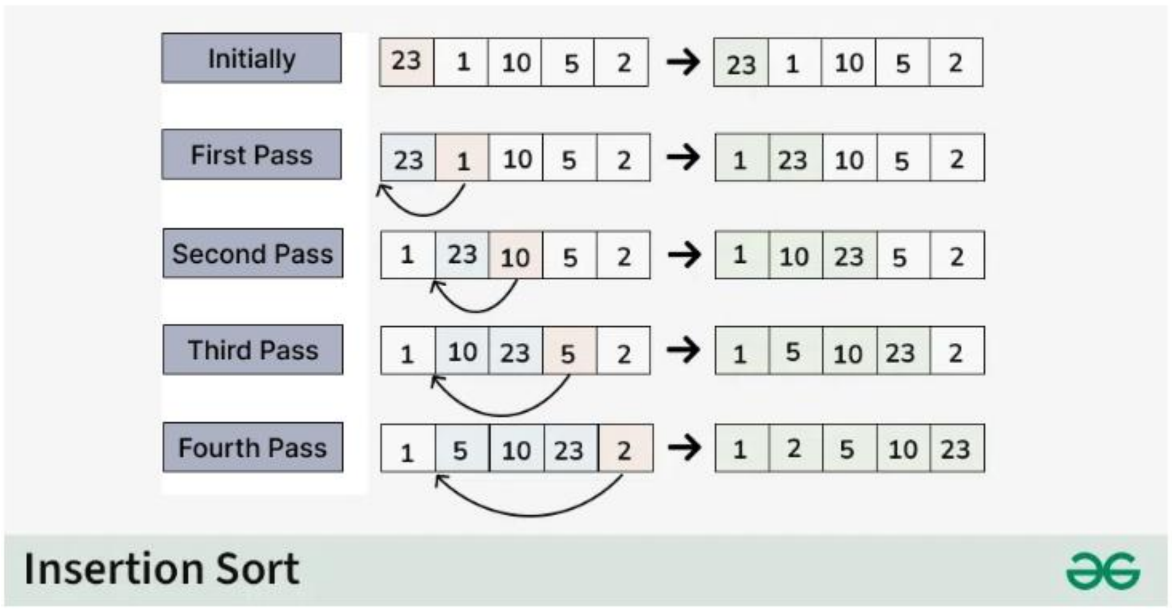
- Step 4: Take 5.
- Compare 5 with 23, shift 23 to the right, then compare 5 with 10, shift 10 to the right, and then compare 5 with 1, 1 is smaller then keep it to become

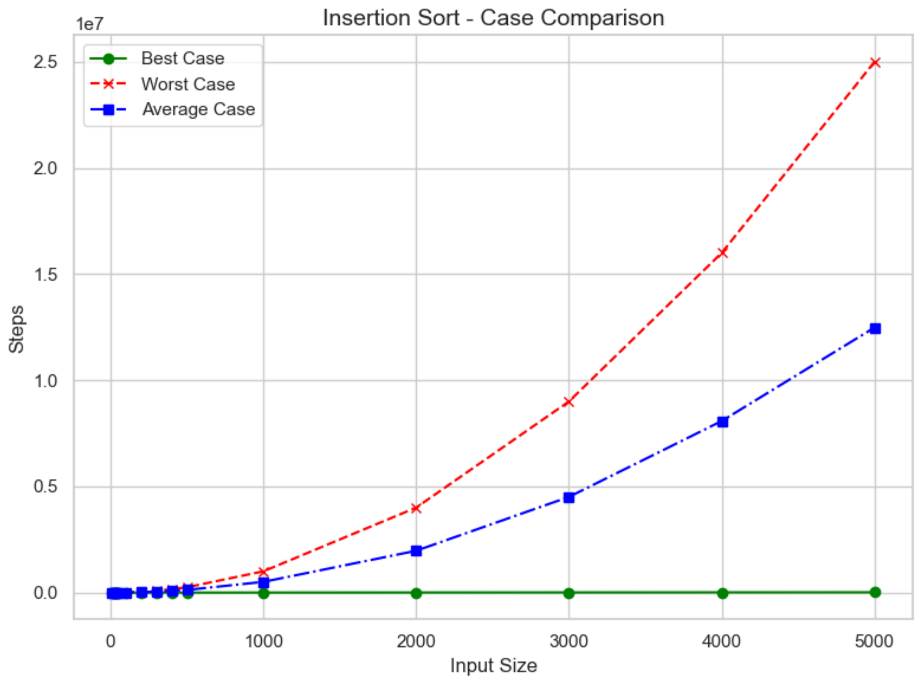
- 1, 5, 10, 23, 2

Step 5: Take 2.

- Compare 2 with 23, shift 23 to the right, then compare 2 with 10, shift 10 to the right, then compare 2 with 5, shift 5 to the right, and then compare 2 with 1, 1 is smaller so it remains the same

  Result:
  1, 2, 5, 10, 23





2- Merge sort

Merge Sort is a divide-and-conquer sorting algorithm. It recursively divides the array into two halves until each subarray has only one element (which is considered sorted). Then, it merges these subarrays back together in sorted order.

Steps for Merge Sort

Given Array:

6,5,12,10,9,1

Step 1: Divide the Array

1. Split the array into two halves:
   - Left: 6, 5, 12
   - Right: 10, 9, 1

Step 2: Recursively Divide the Subarrays

For the Left Subarray 6, 5, 12

1. Split into:
   - Left: 6 (single element, already sorted).
   - Right: 5, 12
2. Split 5, 12into:
   - Left: 5 (sorted).
   - Right: 12(sorted).

For the Right Subarray 10, 9, 1

1. Split into:
   - Left: 10 (sorted).
   - Right: 9, 1.

2. Split 9, 1 into:
    - Left: 9 (sorted).
    - Right: 1 (sorted).

Step 3: Merge Subarrays

Merge for Left Subarray:

1. Merge 5 and 12:
    - Compare 5 and 12 → result: 5, 12
2. Merge 6 and 5, 12:
    - Compare 6 with 5 → pick 5.
    - Compare 6 with 12 → pick 6.
    - Add 12 → result: 5, 6, 12

Merge for Right Subarray:

1. Merge 9 and 1:
    - Compare 9 and 1 → result: 1, 9
2. Merge 10 and 1, 9:
    - Compare 10 with 1 → pick 1.
    - Compare 10 with 9 → pick 9.

- Add 10 → result: 1, 9, 10

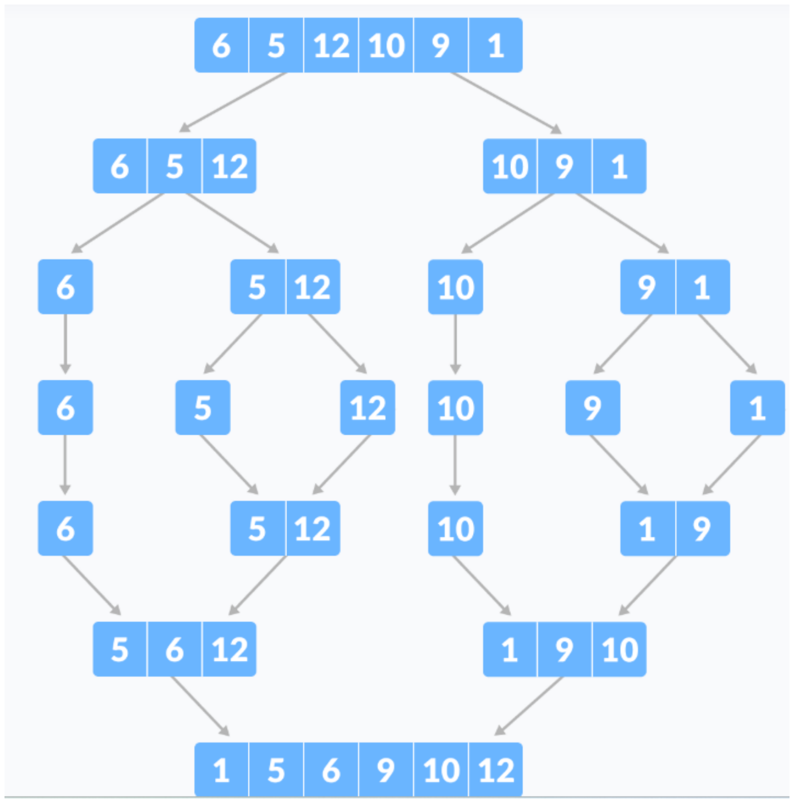
Step 4: Merge Final Subarrays

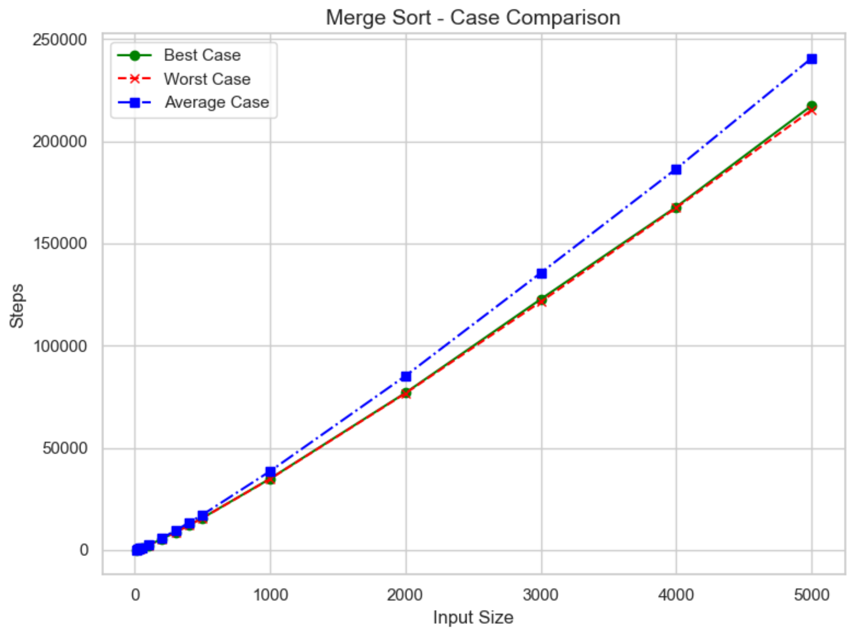
Merge 5, 6, 12 and 1, 9, 10:

1. Compare 5 with 1 → pick 1.
2. Compare 5 with 9 → pick 5.
3. Compare 6 with 9 → pick 6.
4. Compare 12 with 9 → pick 9.
5. Compare 12 with 10 → pick 10.
6. Add 12 → result: 1, 5, 6, 9, 10, 12

Final Sorted Array

1, 5, 6, 9, 10, 12


6 5 12 10 9 1
6 5 12
10 9 1
6
5 12
10
9 1
6
5
12
10
9
1
6
5 12
10
1 9
5 6 12
1 9 10
1 5 6 9 10 12


Merge Sort - Case Comparison
Best Case
Worst Case
Average Case
Steps
Input Size
0
50000
100000
150000
200000
250000
0
1000
2000
3000
4000
5000

3- Heap Sort

Steps for Heap Sort

Given Array:

12, 6, 10, 5, 1, 9

Step 1: Build a Max Heap

In a max heap, the largest element is at the root. We construct the heap from the given array.

1. Start from the last non-leaf node (index n//2−1\text{n} // 2 - 1n//2−1) and move upwards.
    - **Initial Array**: **12, 6, 10, 5, 1, 9**
2. Adjust nodes to satisfy the max heap property.

Heapify Steps:

- **Start with Node 6 (Index 1)**:
    - Children: **5 (Index 3)** and **1 (Index 4)**.
    - No adjustment needed as 6 > 5 and 6 > 1 .
- **Move to Node 10 (Index 2)**:
    - Children: **9 (Index 5)**.
    - No adjustment needed as 10 > 9
- **Move to Node 12 (Index 0)**:

  - Children: **6 (Index 1)** and **10 (Index 2)**.
  - No adjustment needed as 12 > 6 and 12 > 10.

  Max Heap: **12, 6, 10, 5, 1, 9**

Step 2: Extract Elements

1. Swap the root (largest element) with the last element in the array.
2. Reduce the heap size by 1.
3. Heapify the reduced heap to maintain the max heap property.

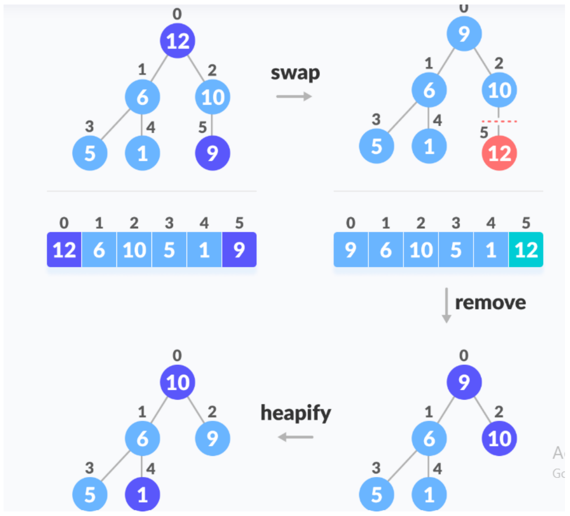
Extraction Steps:

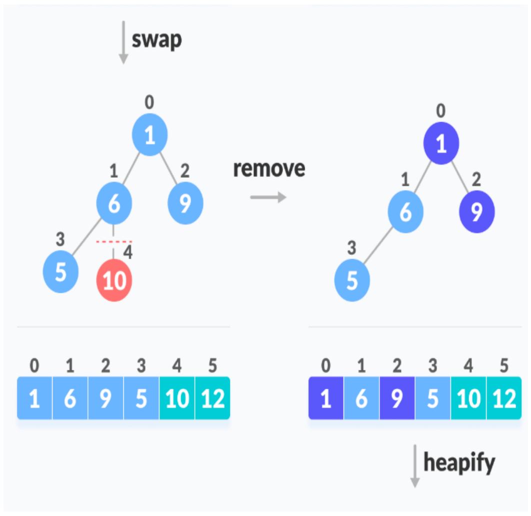
- **Extract 12**:
  - Swap **12** and **9** → **9, 6, 10, 5, 1, 12**
  - Heapify the reduced heap **9, 6, 10, 5, 1**:
    - Root: **9**, Children: **6, 10**.
    - Swap **9** with **10** → **10, 6, 9, 5, 1, 12**.
- **Extract 10**:
  - Swap **10** and **1** → **1, 6, 9, 5, 10, 12**
  - Heapify the reduced heap **1, 6, 9, 5**:
    - Root: **1**, Children: **6, 9**.

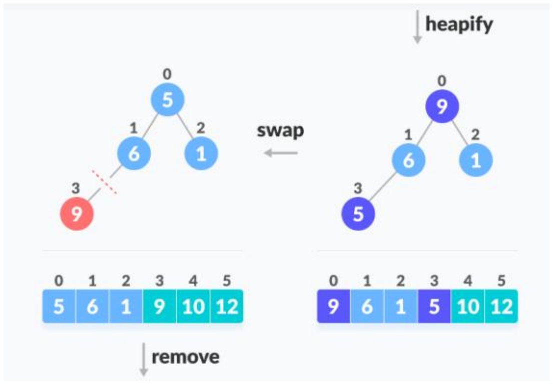
        - Swap **1** with **9** → **9, 6, 1, 5, 10, 12**
        - Heapify subtree: Swap **1** with **5** → **9, 6, 5, 1, 10, 12**.
- **Extract 9**:
    - Swap **9** and **1** → **1, 6, 5, 9, 10, 12**
    - Heapify the reduced heap **1, 6, 5**
        - Root: **1**, Children: **6, 5**.
        - Swap **1** with **6** → **6, 1, 5, 9, 10, 12**.
- **Extract 6**:
    - Swap **6** and **5** → **5, 1, 6, 9, 10, 12**.
    - Heapify the reduced heap **5, 1**
        - Root: **5**, Child: **1**.
        - No adjustment needed.
- **Extract 5**:
    - Swap **5** and **1** → **1, 5, 6, 9, 10, 12**

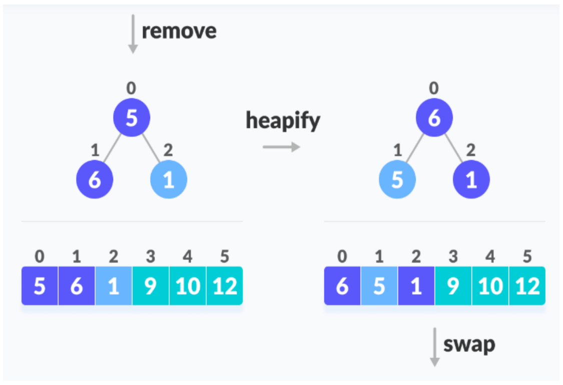
Final Sorted Array:

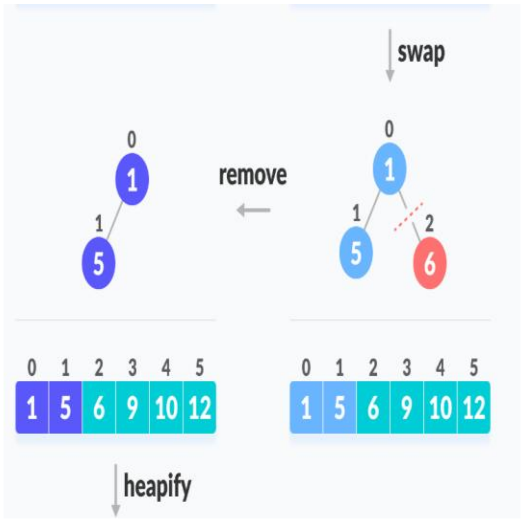
1, 5, 6, 9, 10, 12

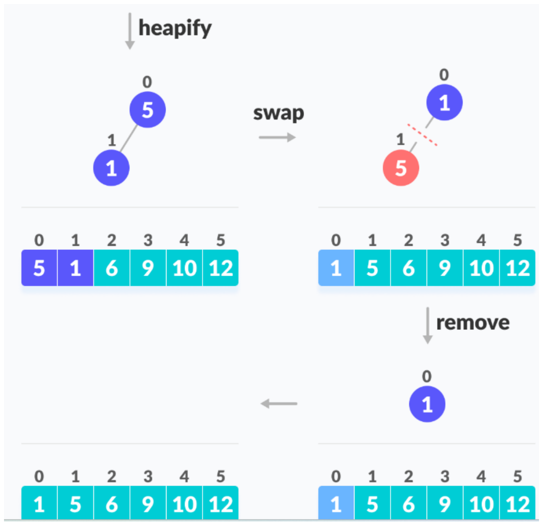

swap
remove
heapify

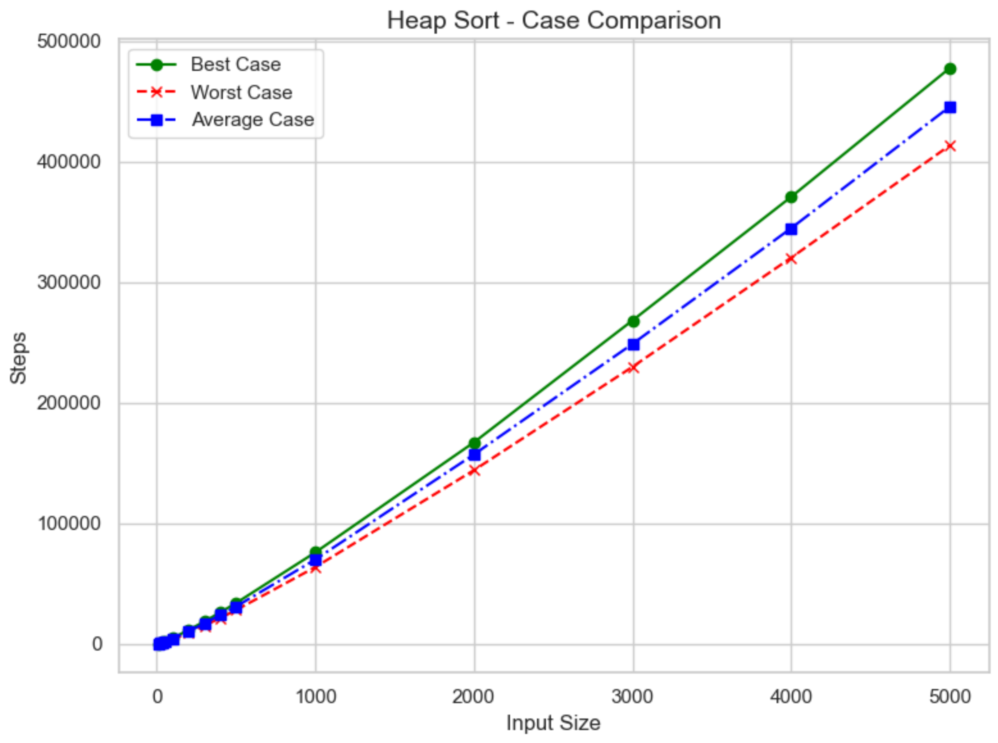

swap
remove
heapify


heapify
swap
remove


remove
heapify
swap


swap
remove
heapify


heapify
swap
remove



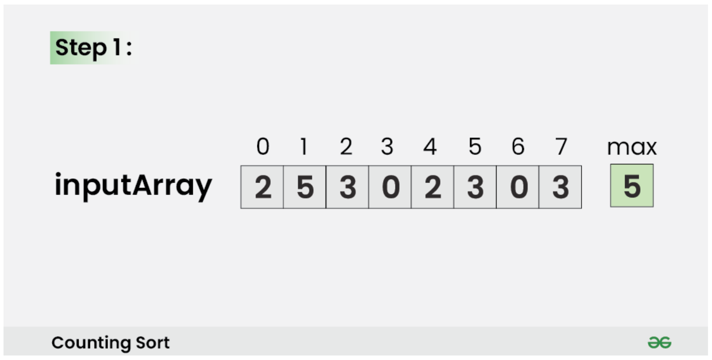
4- Counting sort

Counting Sort is a non-comparison-based sorting algorithm. It is efficient for sorting integers or categorical data within a limited range. The algorithm counts the frequency of each element in the array and uses this information to place the elements in the correct position in the sorted array.

<u>How does Counting Sort Algorithm work?</u>

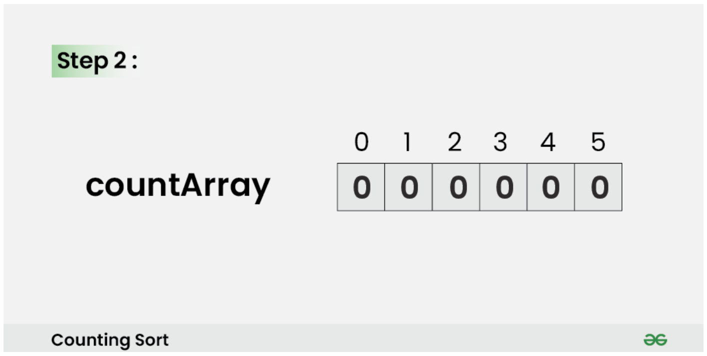
Step1 :

- Find out the **maximum** element from the given array.



Step 2:

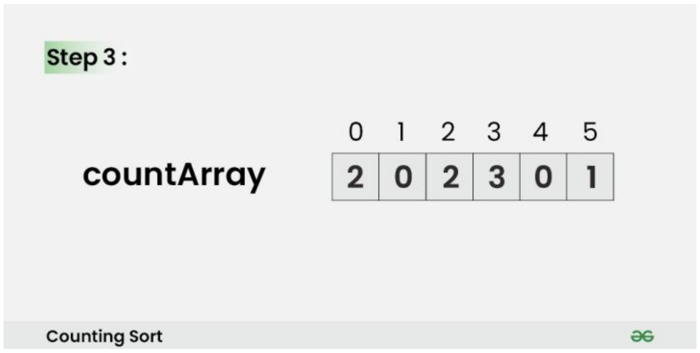
- Initialize a **countArray[]** of length **max+1** with all elements as **0**. This array will be used for storing the occurrences of the elements of the input array.



Step 3:

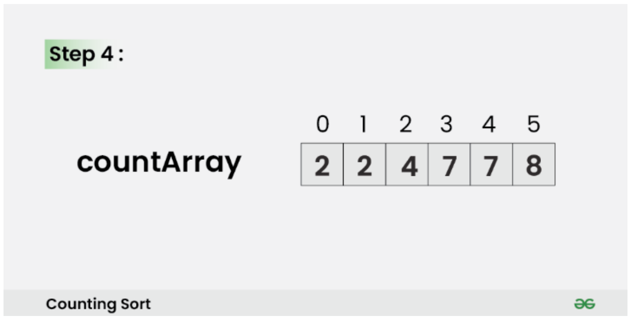
- In the **countArray[]**, store the count of each unique element of the input array at their respective indices.

- **For Example:** The count of element **2** in the input array is **2.** So, store **2** at index **2** in the **countArray[]**. Similarly, the count of element **5** in the input array is **1**, hence store **1** at index **5** in the **countArray[]**.



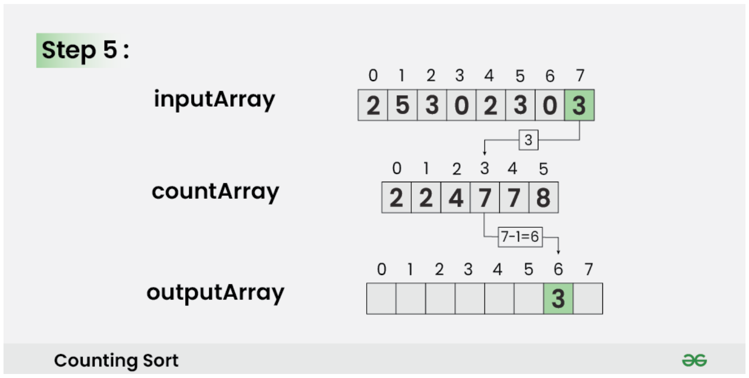
Step 4:

- Store the **cumulative sum** or **prefix sum** of the elements of the **countArray[]** by doing **countArray[i] = countArray[i – 1] + countArray[i].** This will help in placing the elements of the input array at the correct index in the output array.



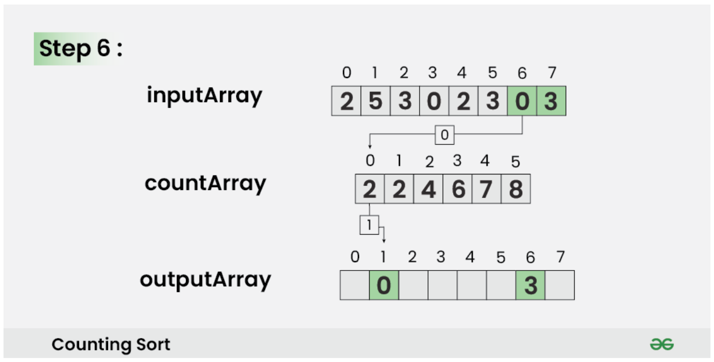
Step 5:

- Iterate from end of the input array and because traversing input array from end preserves the order of equal elements, which eventually makes this sorting algorithm **stable**.
- *Update **outputArray[countArray[inputArray[i]] – 1] = inputArray[i]**.*
- *Also, update **countArray[inputArray[i]] = countArray[inputArray[i]]**– -.*



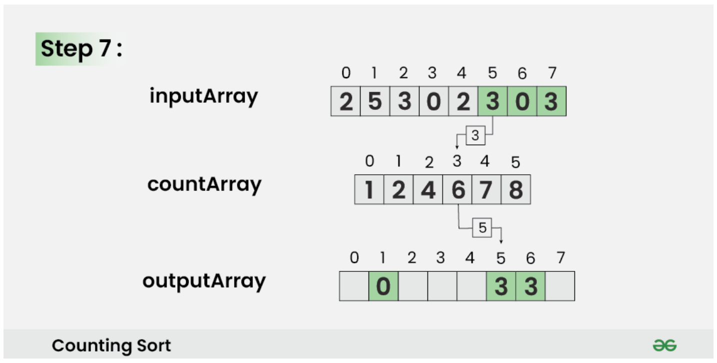
Step 6: For i = 6,

Update ***outputArray[countArray[inputArray[6]] – 1] = inputArray[6]***
Also, update ***countArray[inputArray[6]] = countArray[inputArray[6]]- –***



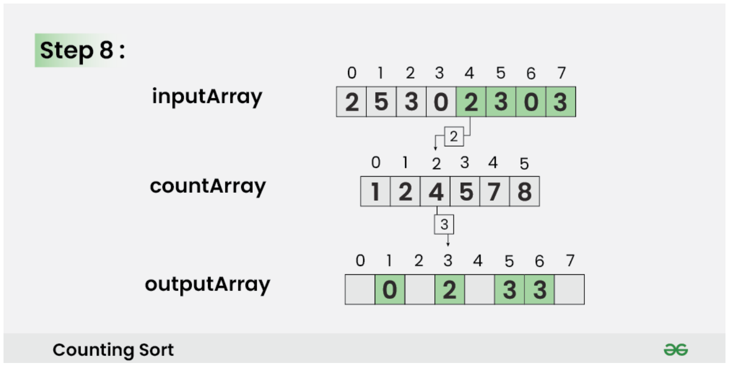
Step 7: For i = 5,

Update ***outputArray[countArray[inputArray[5]] – 1] = inputArray[5]***
Also, update ***countArray[inputArray[5]] = countArray[inputArray[5]]- –***



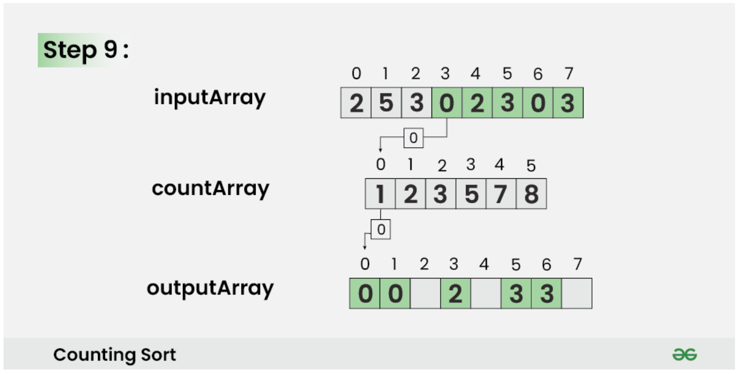
Step 8: For i = 4,

Update ***outputArray[countArray[inputArray[4]] – 1] = inputArray[4]***
Also, update ***countArray[inputArray[4]] = countArray[inputArray[4]]- –***



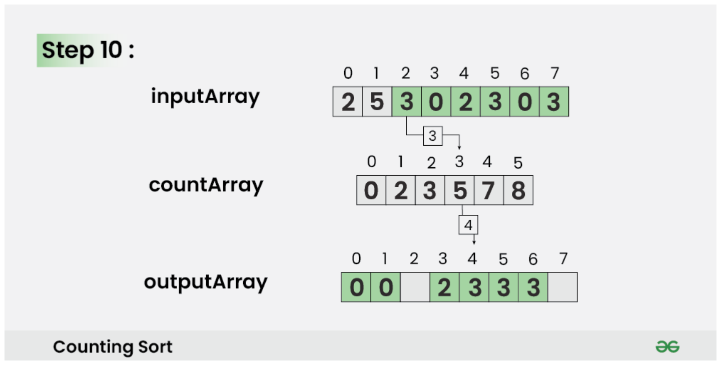
Step 9: For i = 3,

Update ***outputArray[countArray[inputArray[3]] – 1] = inputArray[3]***
Also, update ***countArray[inputArray[3]] = countArray[inputArray[3]]- –***



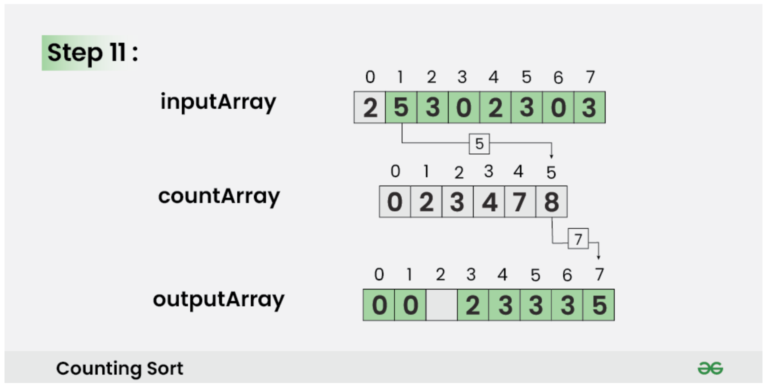
Step 10: For i = 2,

Update ***outputArray[countArray[inputArray[2]] – 1] = inputArray[2]***
Also, update ***countArray[inputArray[2]] = countArray[inputArray[2]]- –***



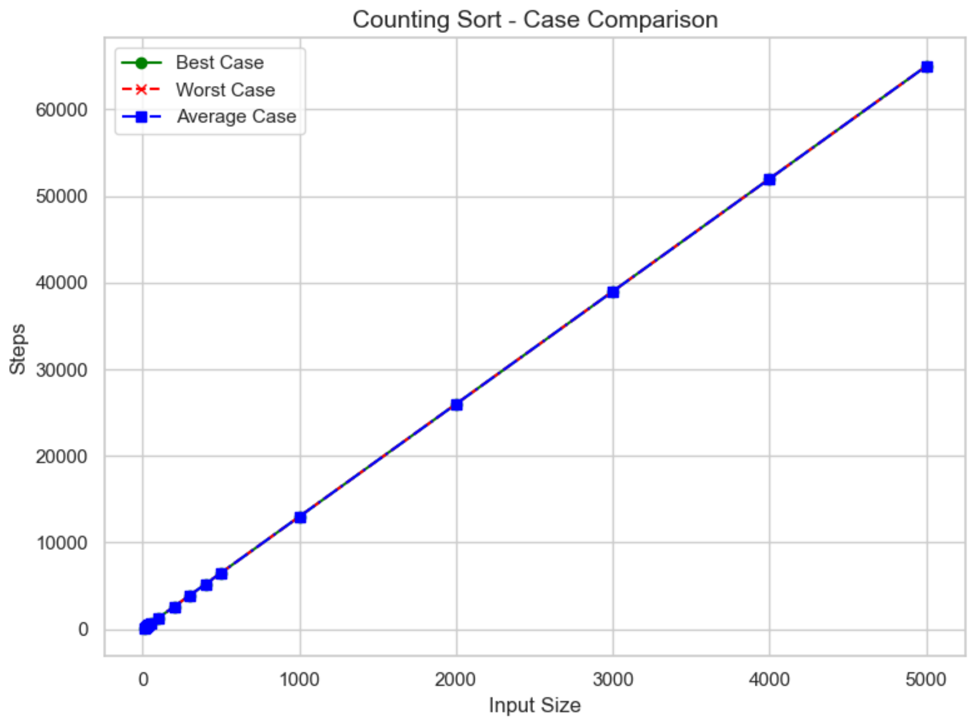
Step 11: For i = 1,

Update ***outputArray[countArray[inputArray[1]] – 1] = inputArray[1]*** *Also, update* ***countArray[inputArray[1]] = countArray[inputArray[1]]- –***



Step 12: For i = 0,

Update ***outputArray[countArray[inputArray[0]] – 1] = inputArray[0]*** *Also, update* ***countArray[inputArray[0]] = countArray[inputArray[0]]- –***



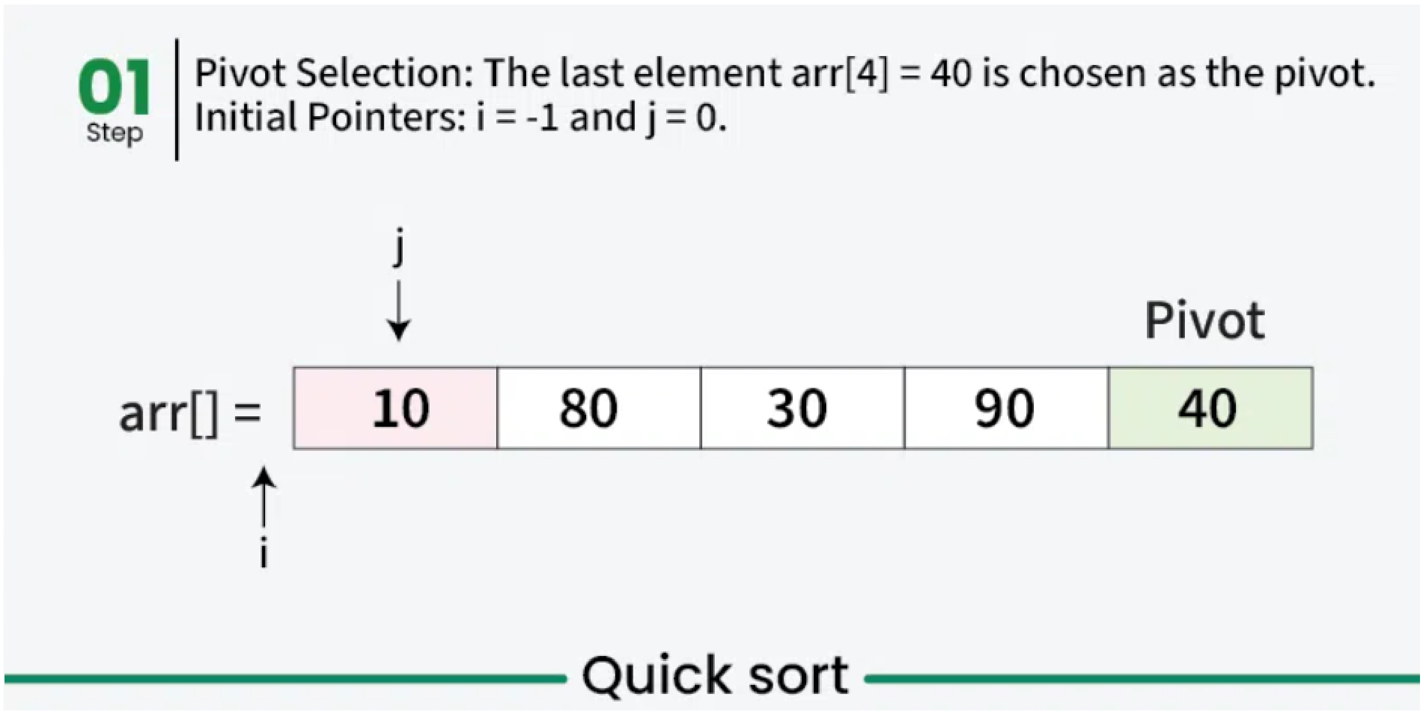
5- Quick sort

Quick Sort is a highly efficient sorting algorithm that follows the **divide-and-conquer** approach. It works by selecting a "pivot" element from the array and partitioning the other elements into two subarrays:

- Elements smaller than the pivot.
- Elements larger than the pivot.

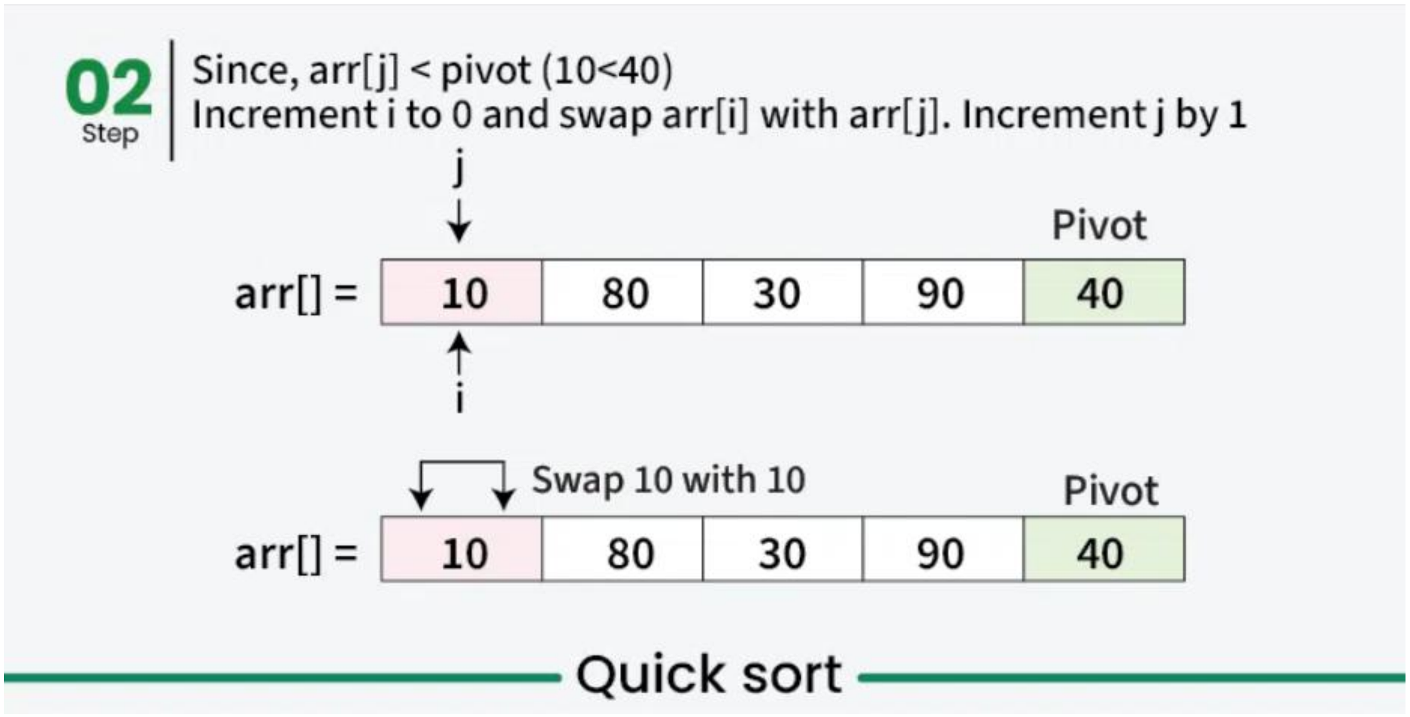
These subarrays are then recursively sorted.

Steps to execute quick sort



01 Step

Pivot Selection: The last element arr[4] = 40 is chosen as the pivot.
Initial Pointers: i = -1 and j = 0.

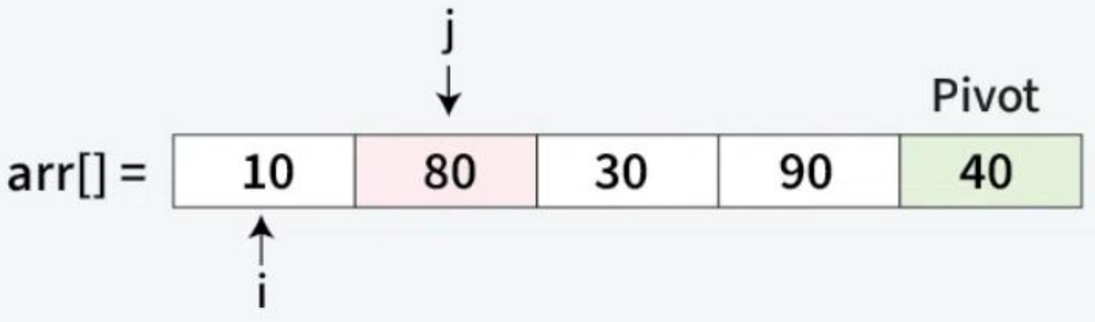


02 Step

Since, arr[j] < pivot (10<40)
Increment i to 0 and swap arr[i] with arr[j]. Increment j by 1

03 Step

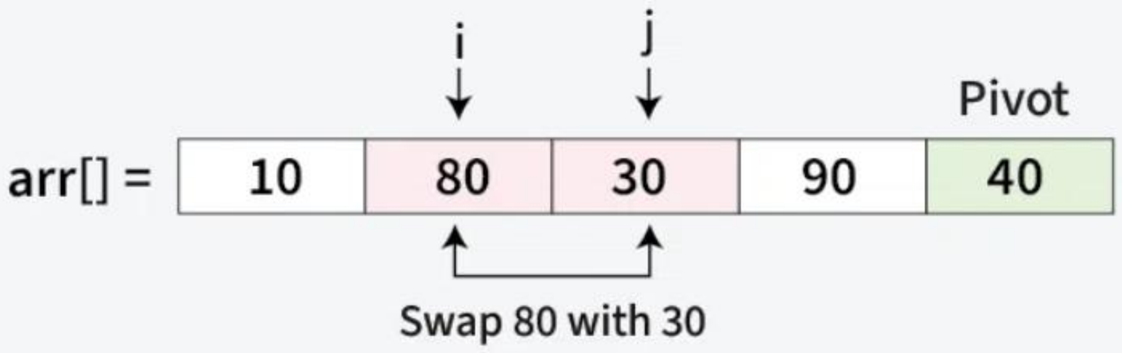
Since, arr[j] > pivot (80>40)
No swap needed. Increment j by 1



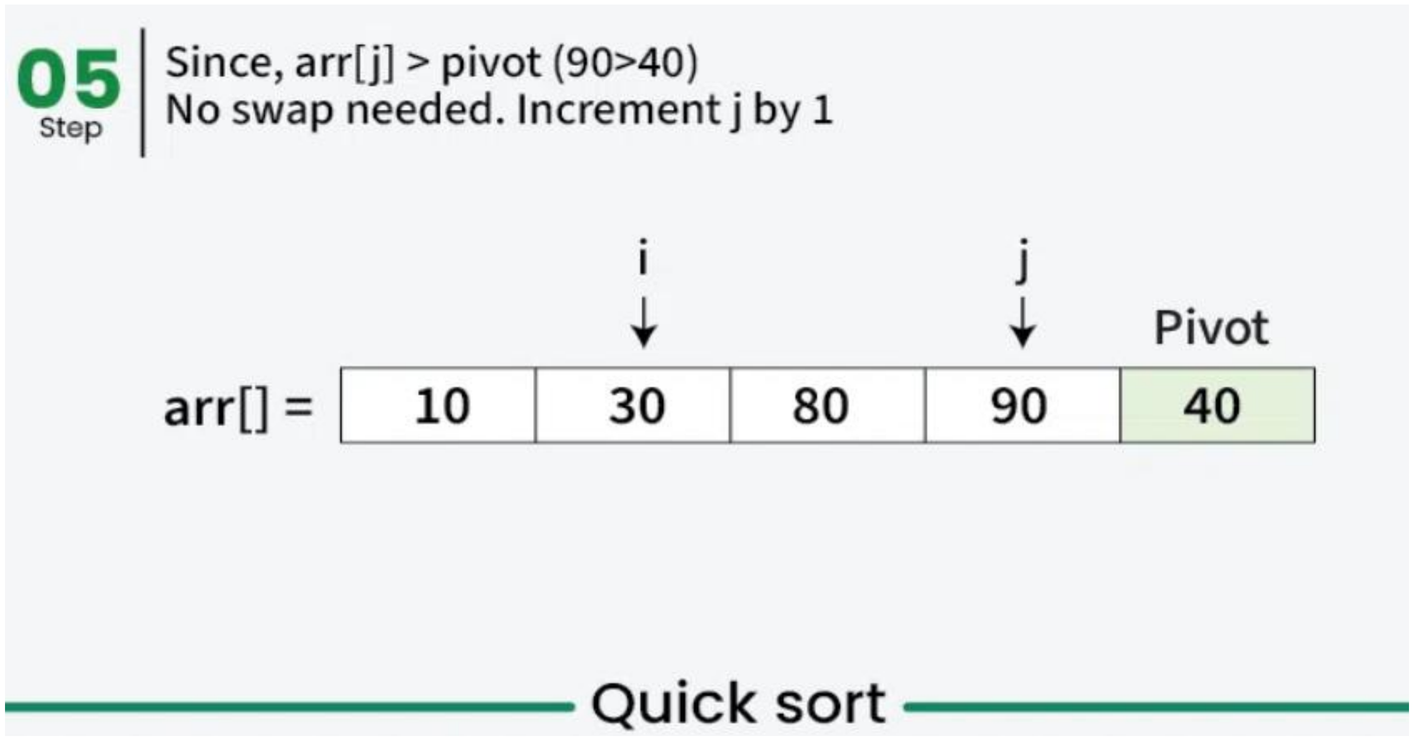
Quick sort

04 Step

Since, arr[j] < pivot (30<40)
Increment i by 1 and swap arr[i] with arr[j]. Increment j by 1

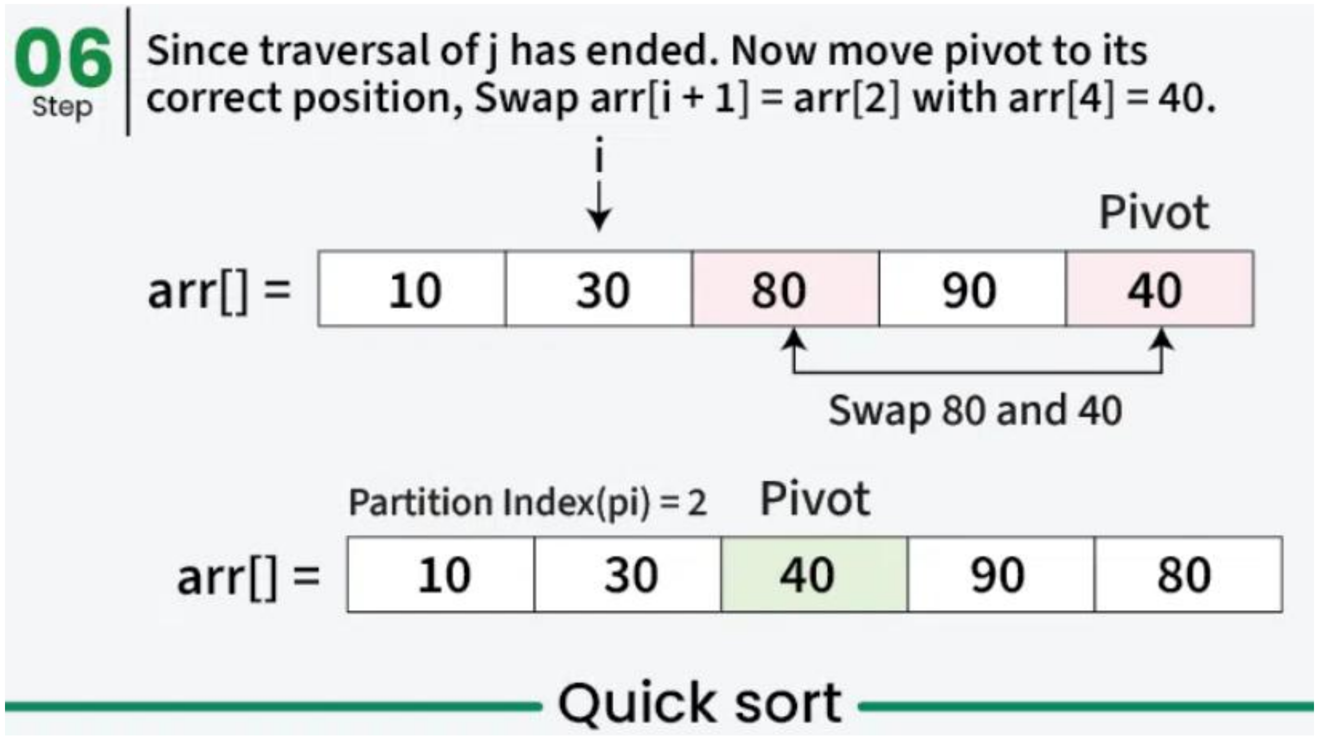


Quick sort



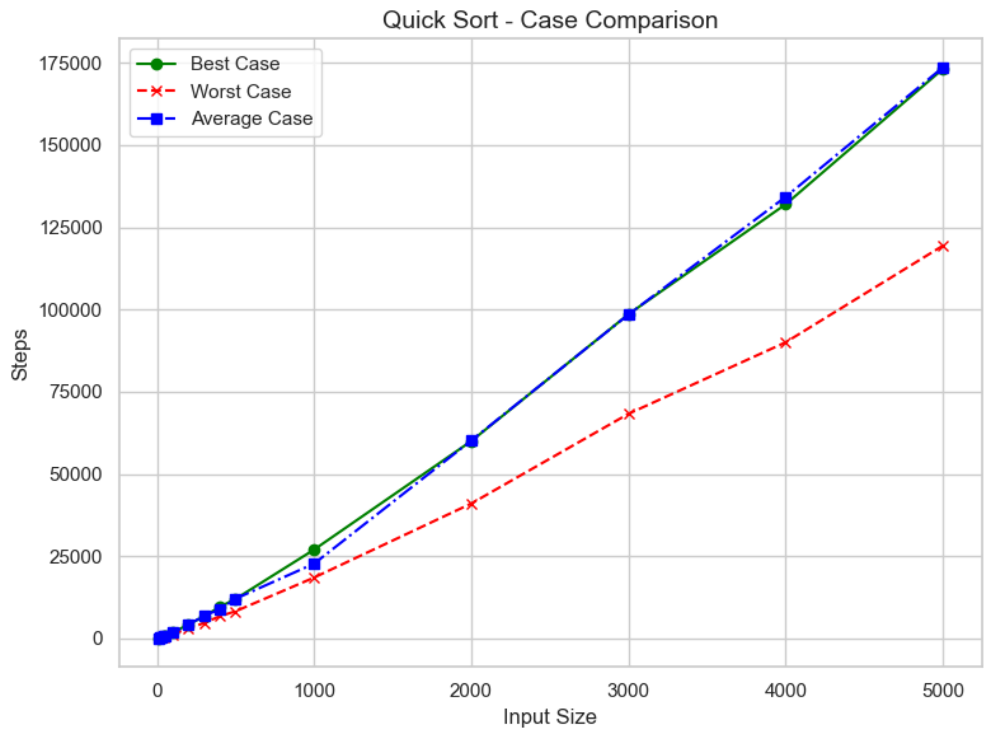
05 Step

Since, arr[j] > pivot (90>40)
No swap needed. Increment j by 1



06 Step

Since traversal of j has ended. Now move pivot to its correct position, Swap arr[i + 1] = arr[2] with arr[4] = 40.



6- Bubble sort

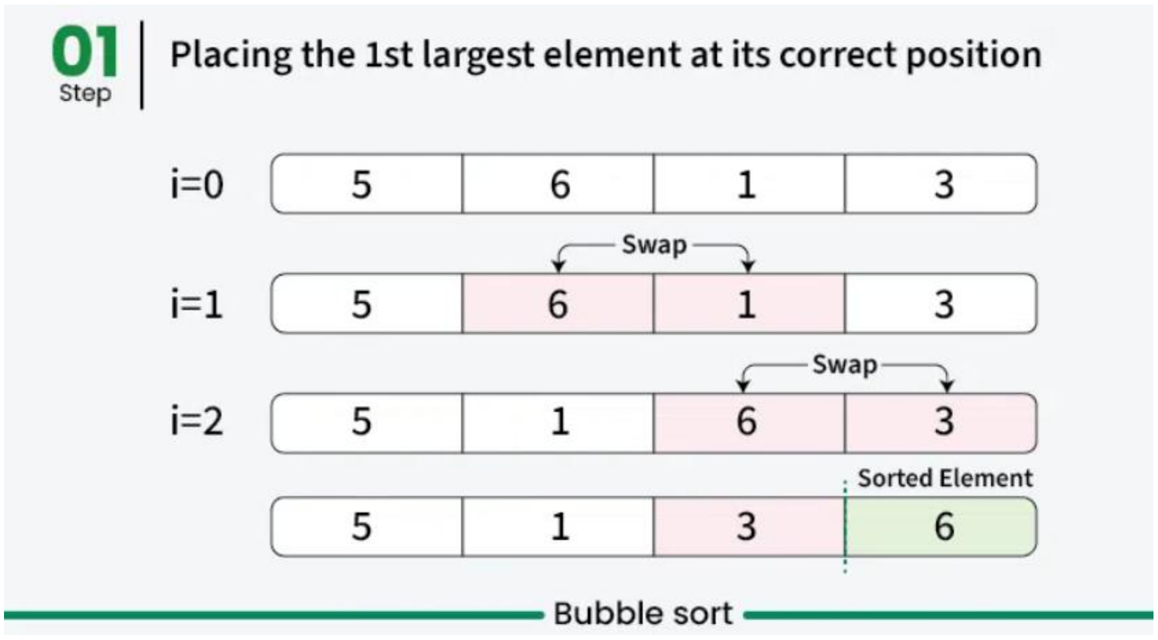
Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity are quite high.

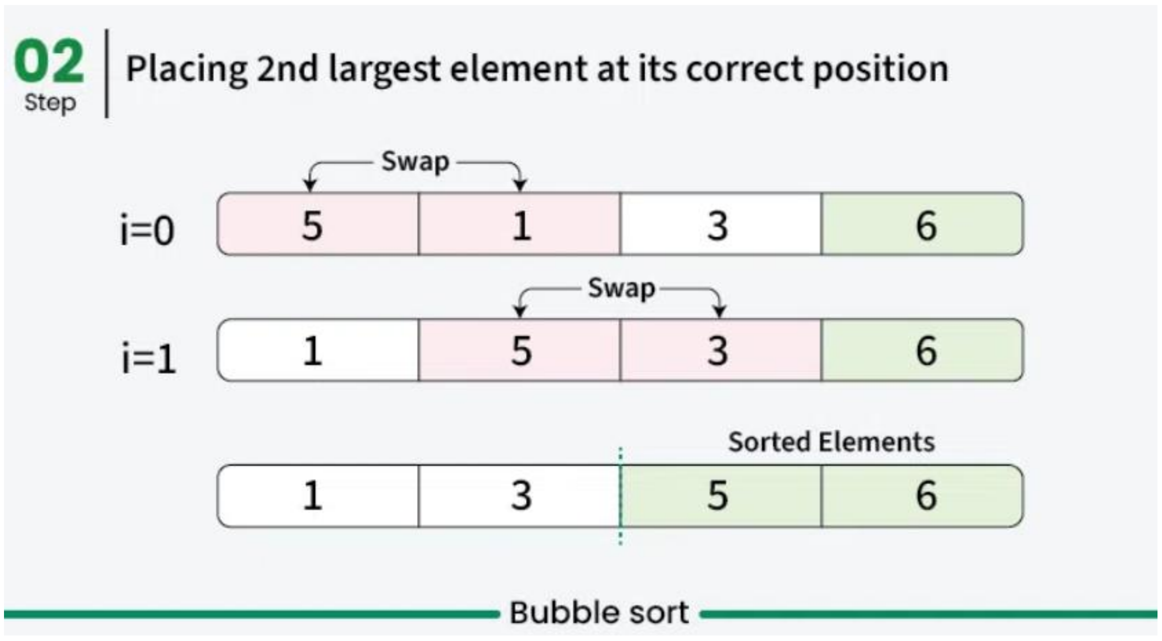
- **We sort the array using multiple passes. After the first pass, the maximum element goes to end (its correct position). Same way, after second pass, the second largest element goes to second last position and so on.**

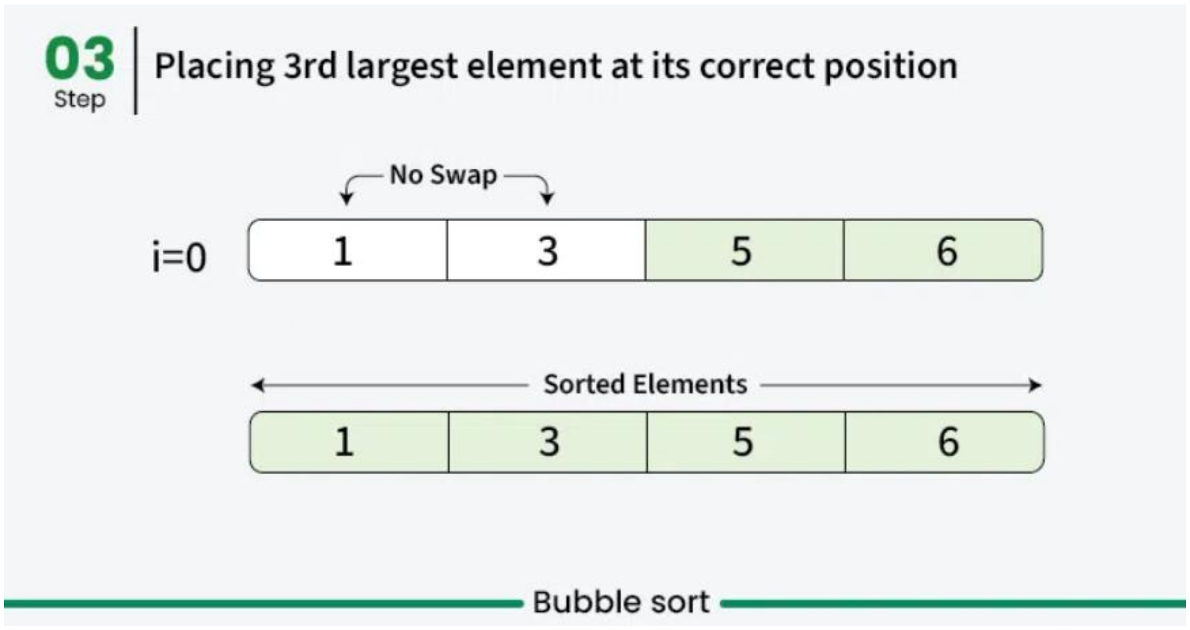
- **In every pass, we process only those elements that have already not moved to correct position. After k passes, the largest k elements must have been moved to the last k positions.**

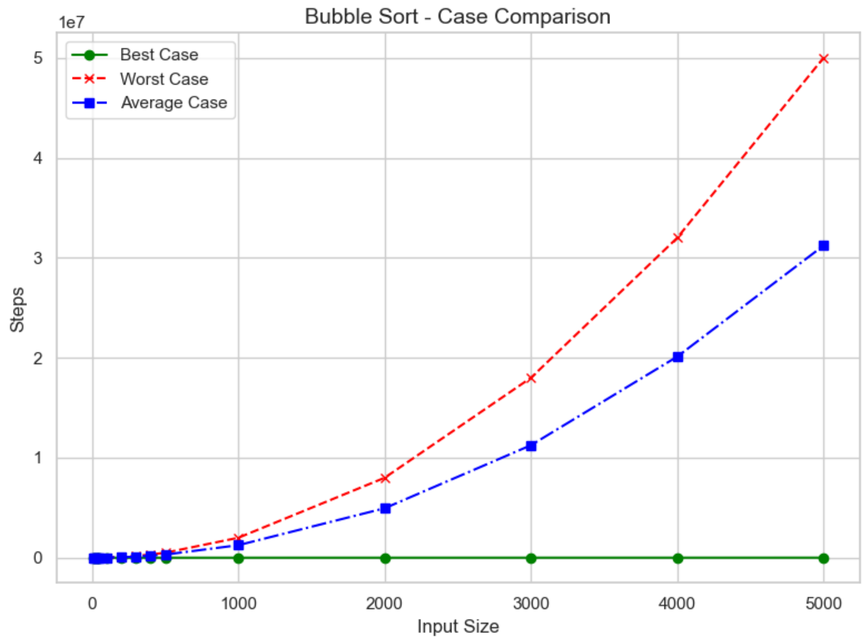
- **In a pass, we consider remaining elements and compare all adjacent and swap if larger element is before a smaller element. If we keep doing this, we get the largest (among the remaining elements) at its correct position.**

Steps to execute bubble sort






03
Step
Placing 3rd largest element at its correct position
No Swap
i=0
1
3
5
6
Sorted Elements
1
3
5
6
Bubble sort


Bubble Sort - Case Comparison
1e7
Best Case
Worst Case
Average Case
Steps
Input Size
0
1
2
3
4
5
0
1000
2000
3000
4000
5000

7- Radix sort

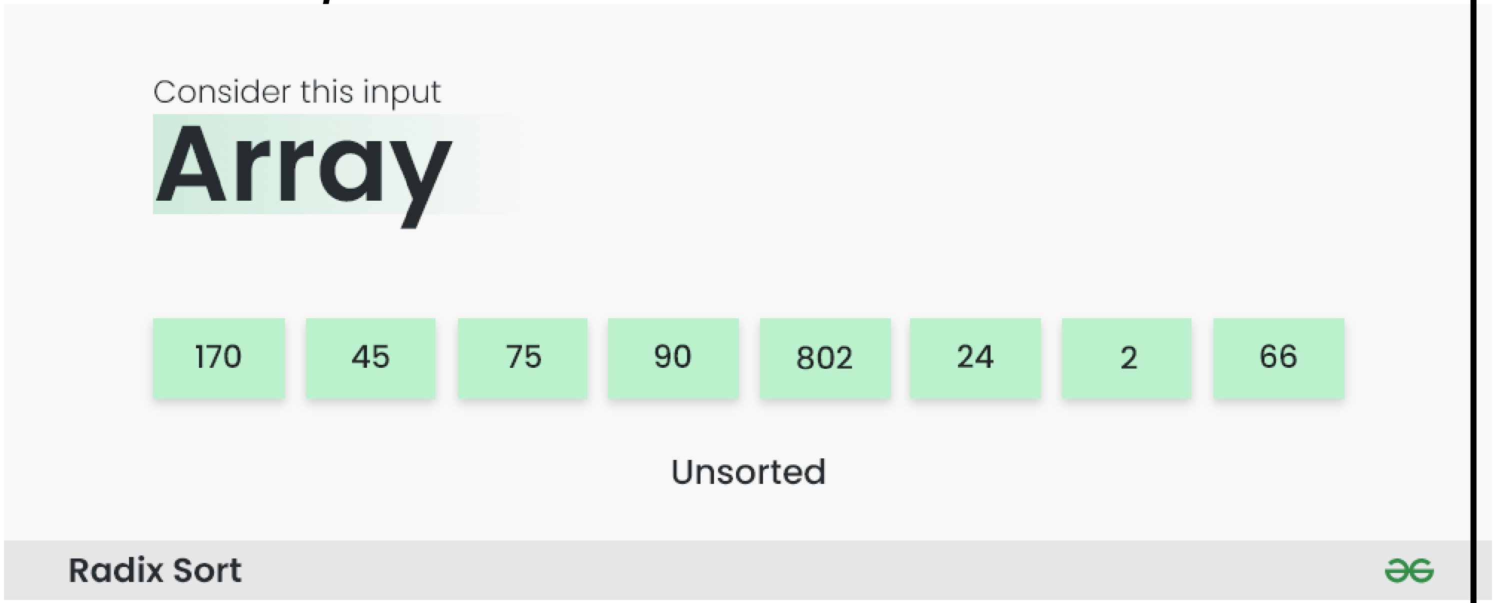
Radix Sort is a linear sorting algorithm that sorts elements by processing them digit by digit. It is an efficient sorting algorithm for integers or strings with fixed-size keys.
Rather than comparing elements directly, Radix Sort distributes the elements into buckets based on each digit's value. By repeatedly sorting the elements by their significant digits, from the least significant to the most significant, Radix Sort achieves the final sorted order.

Radix Sort Algorithm

The key idea behind Radix Sort is to exploit the concept of place value. It assumes that sorting numbers digit by digit will eventually result in a fully sorted list. Radix Sort can be performed using different variations, such as Least Significant Digit (LSD) Radix Sort or Most Significant Digit (MSD) Radix Sort.

Steps to execute Radix sort

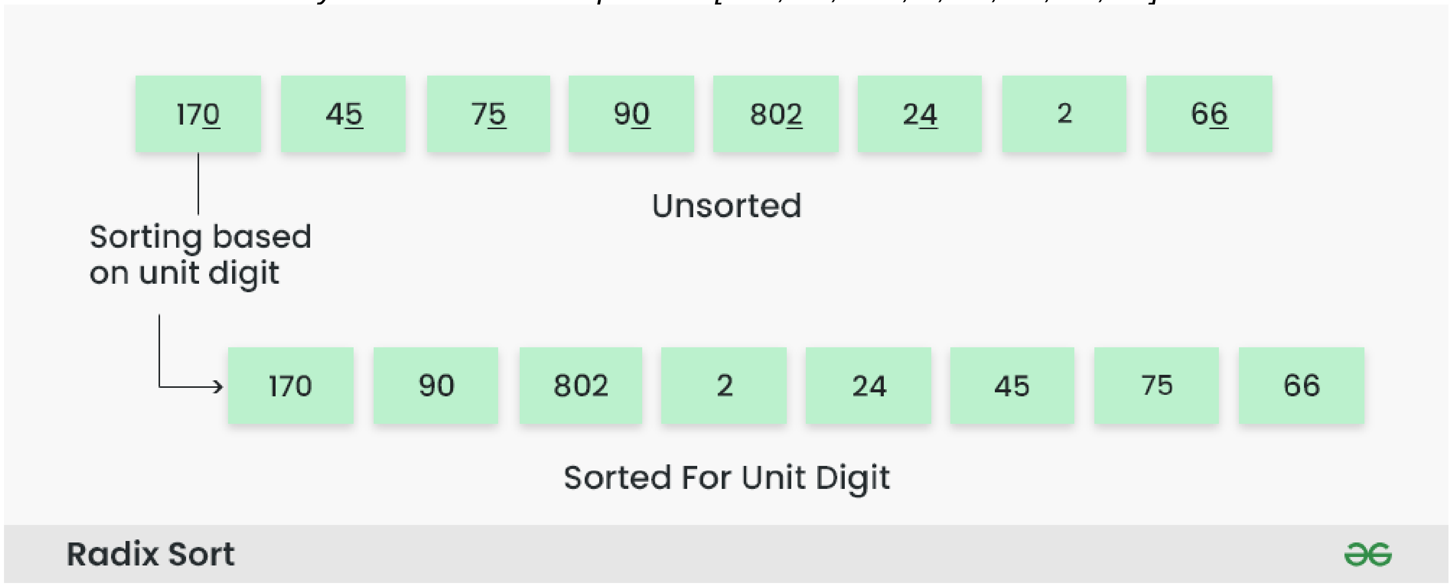
To perform radix sort on the array [170, 45, 75, 90, 802, 24, 2, 66], we follow these steps:



***Step 1**: Find the largest element in the array, which is 802. It has three digits, so we will iterate three times, once for each significant place.*

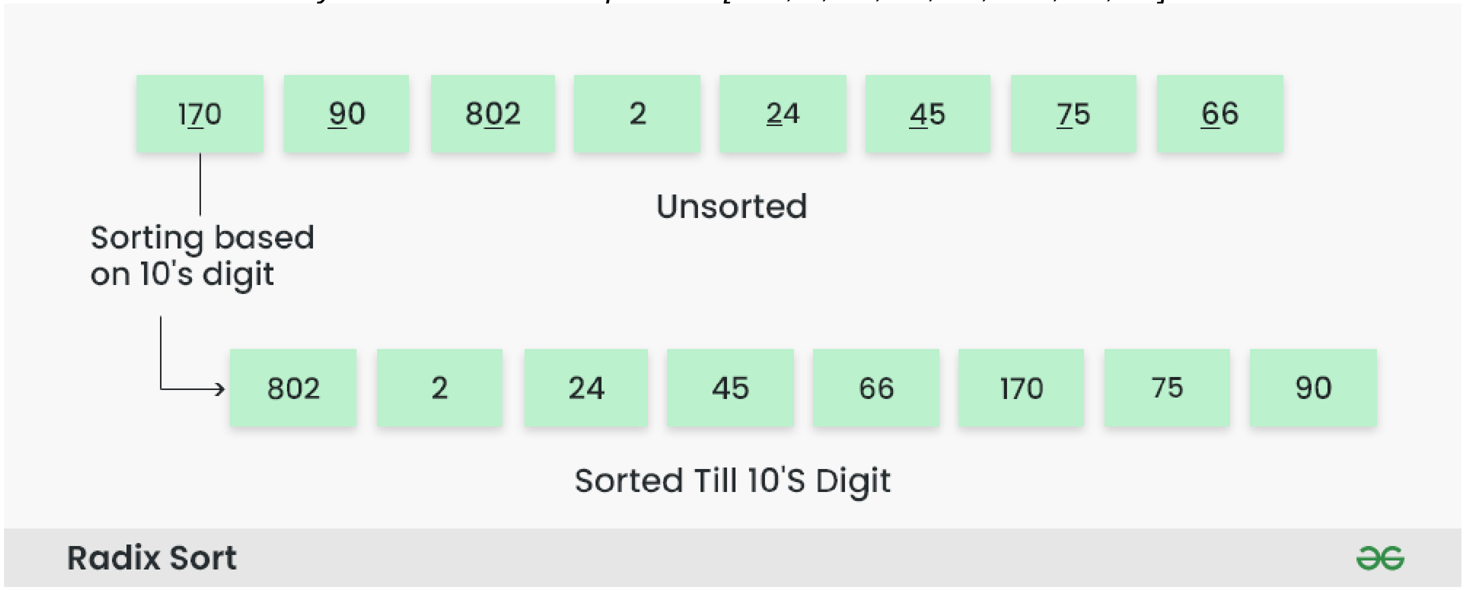
***Step 2**: Sort the elements based on the unit place digits (X=0). We use a stable sorting technique, such as counting sort, to sort the digits at each significant place. It's important to understand that the default implementation of counting sort is unstable i.e. same keys can be in a different order than the input array. To solve this problem, We can iterate the input array in reverse order to build the output array. This strategy helps us to keep the same keys in the same order as they appear in the input array.*
Sorting based on the unit place:

- *Perform counting sort on the array based on the unit place digits.*
- *The sorted array based on the unit place is [170, 90, 802, 2, 24, 45, 75, 66].*



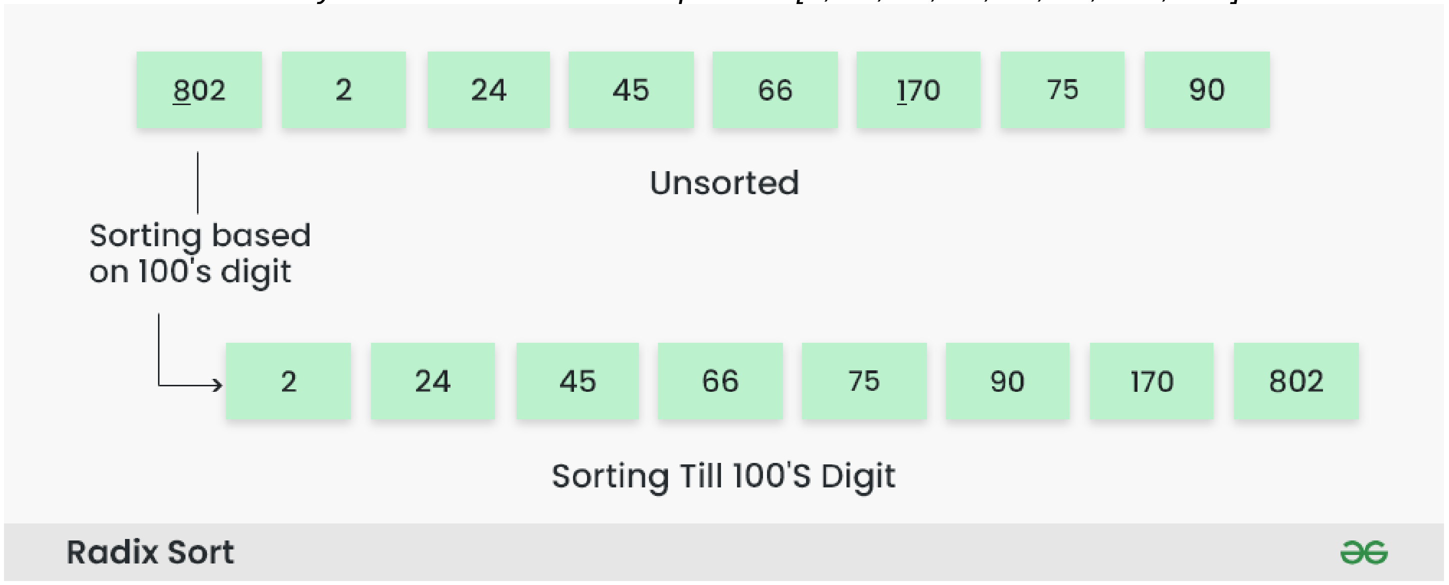
***Step 3:** Sort the elements based on the tens place digits.*
Sorting based on the tens place:

- *Perform counting sort on the array based on the tens place digits.*
- *The sorted array based on the tens place is [802, 2, 24, 45, 66, 170, 75, 90].*

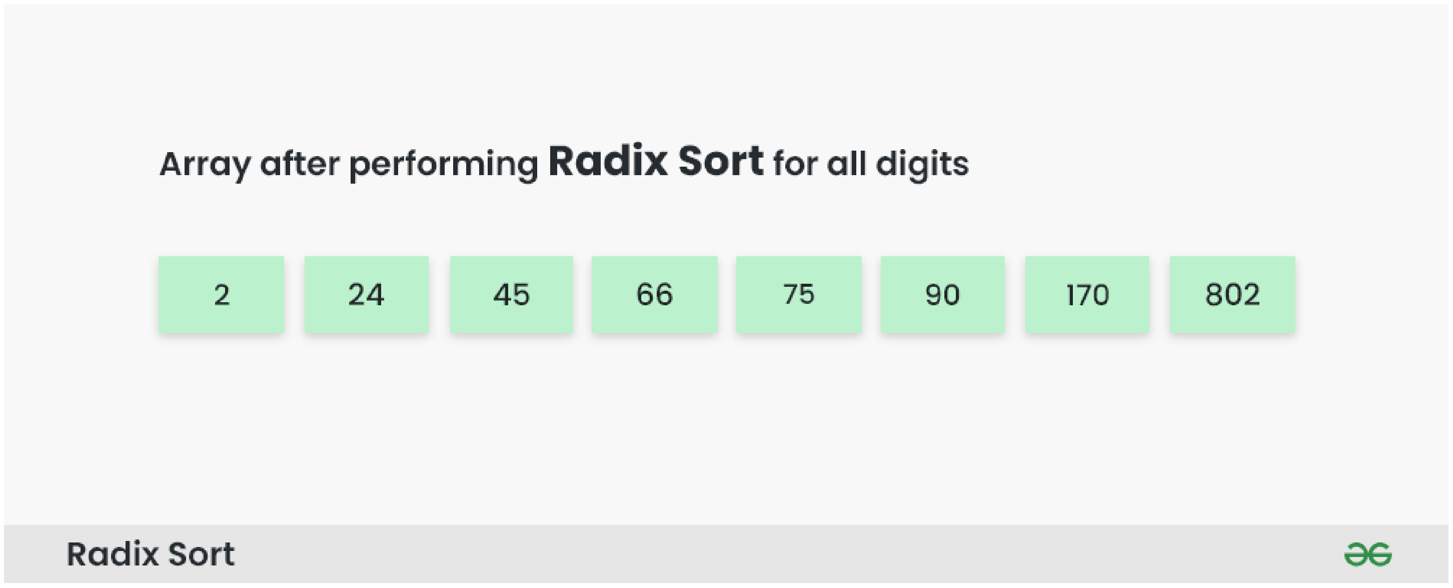


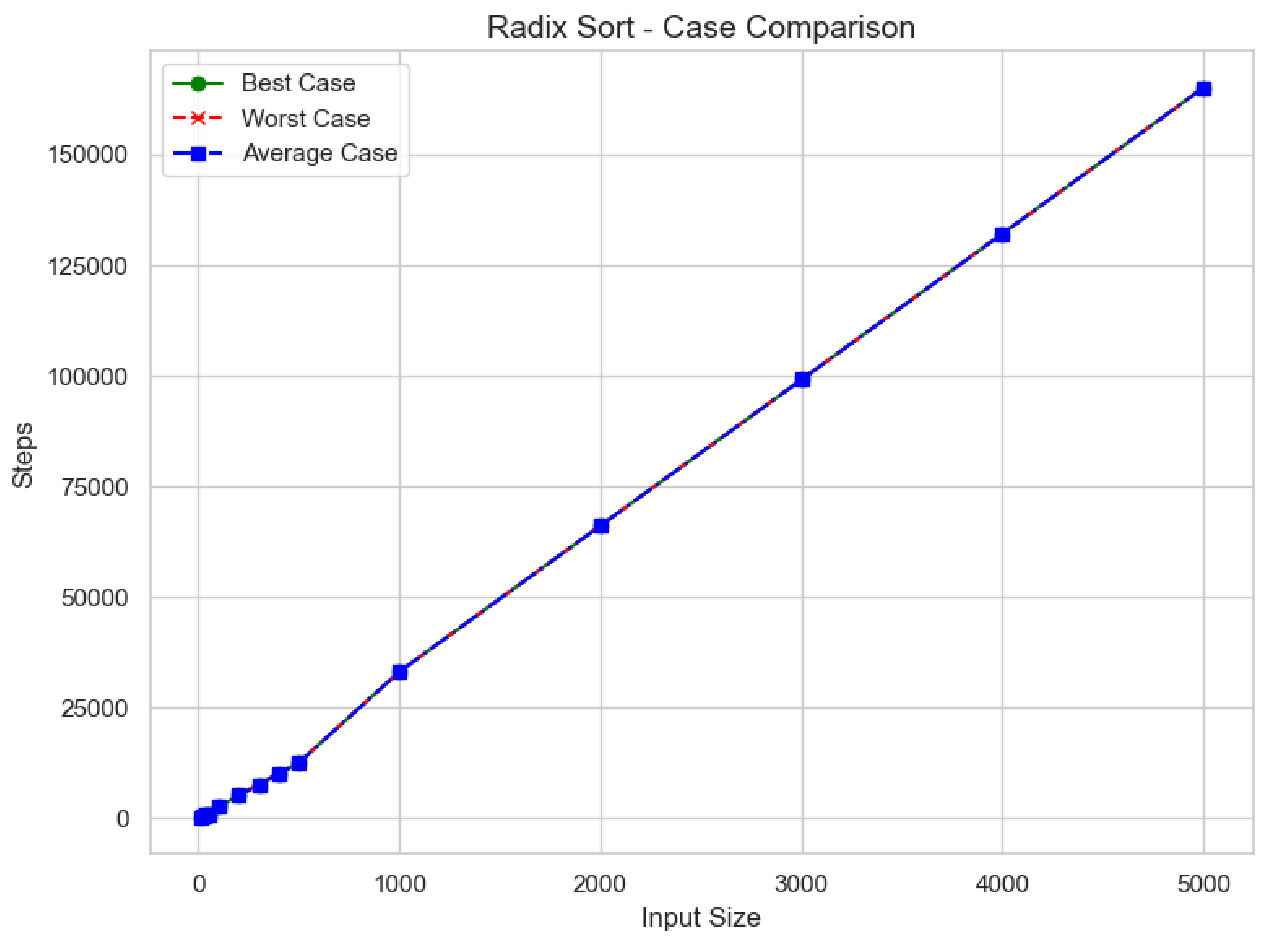
***Step 4**: Sort the elements based on the hundreds place digits.*
Sorting based on the hundreds place:

- *Perform counting sort on the array based on the hundreds place digits.*
- *The sorted array based on the hundreds place is [2, 24, 45, 66, 75, 90, 170, 802].*



***Step 5**: The array is now sorted in ascending order.*
The final sorted array using radix sort is [2, 24, 45, 66, 75, 90, 170, 802].





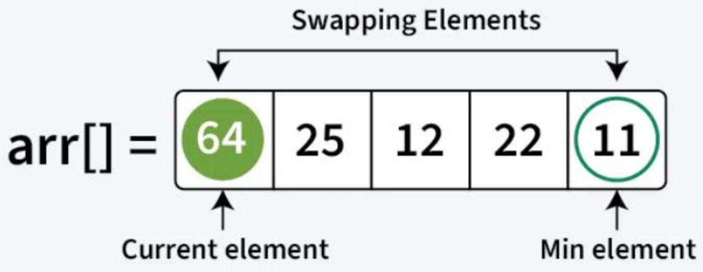
8- Selection sort

Selection Sort is a comparison-based sorting algorithm. It sorts an array by repeatedly selecting the smallest (or largest) element from the unsorted portion and swapping it with the first unsorted element. This process continues until the entire array is sorted.

1. **First we find the smallest element and swap it with the first element. This way we get the smallest element at its correct position.**
2. **Then we find the smallest among remaining elements (or second smallest) and swap it with the second element.**
3. **We keep doing this until we get all elements moved to correct position.**

01 Step

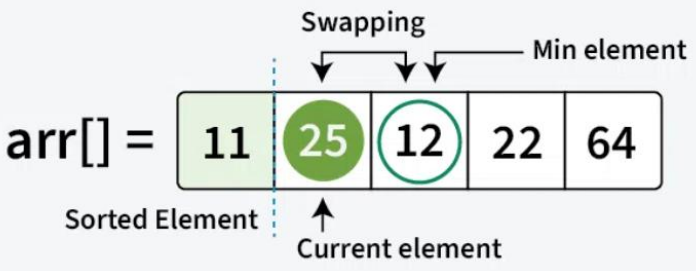
Start from the first element at index 0, find the smallest element in the rest of the array which is unsorted, and swap (11) with current element(64).



Selection Sort Algorithm

02 Step

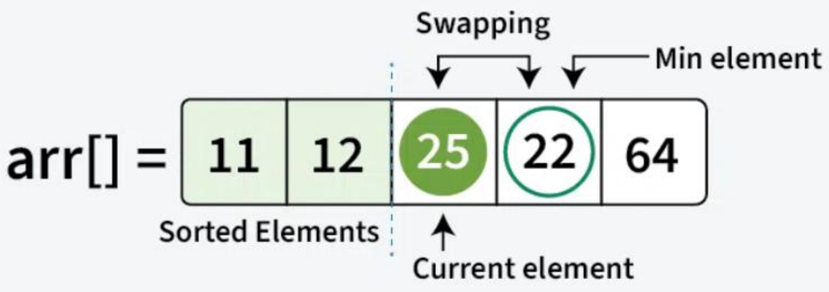
Move to the next element at index 1 (25). Find the smallest in unsorted subarray, and swap (12) with current element (25).



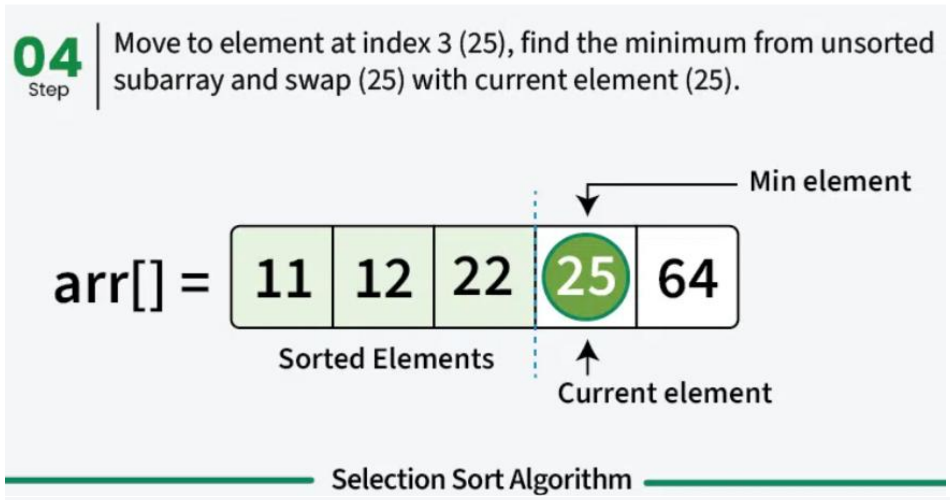
Selection Sort Algorithm

03 Step

Move to element at index 2 (25). Find the minimum element from unsorted subarray, Swap (22) with current element (25).

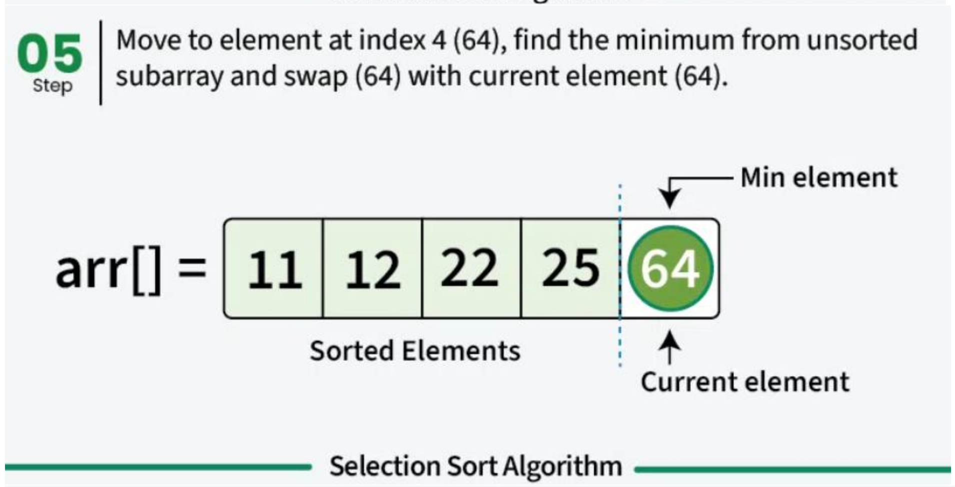


Selection Sort Algorithm



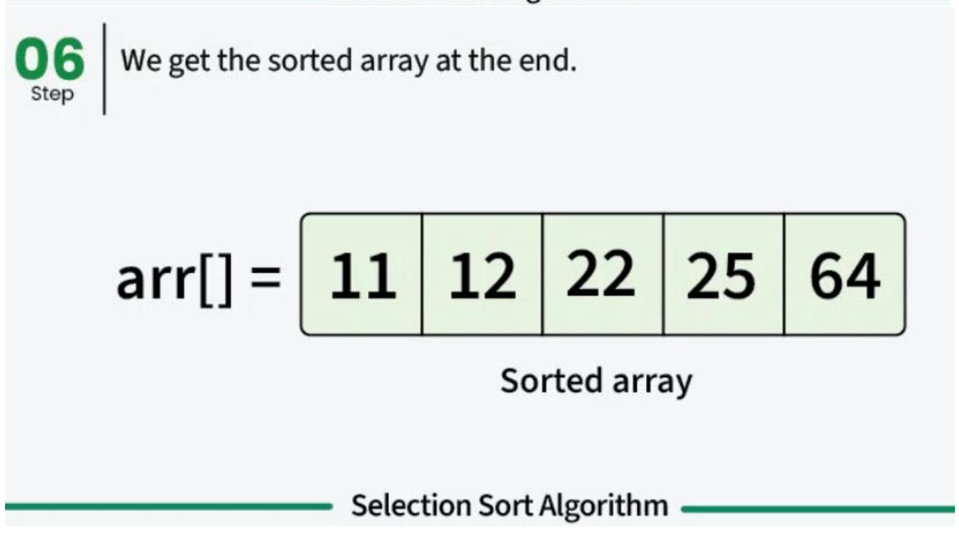
04 Step

Move to element at index 3 (25), find the minimum from unsorted subarray and swap (25) with current element (25).



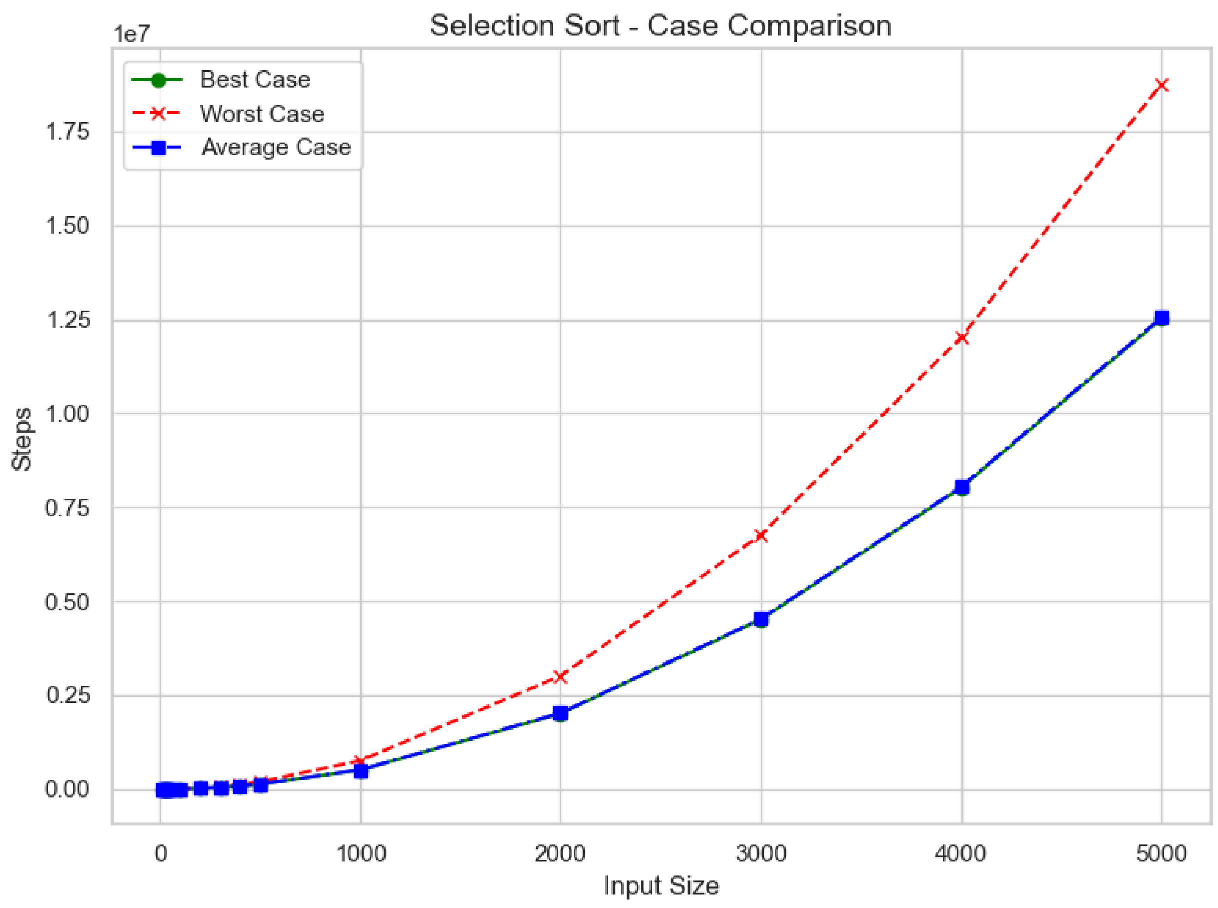
05 Step

Move to element at index 4 (64), find the minimum from unsorted subarray and swap (64) with current element (64).



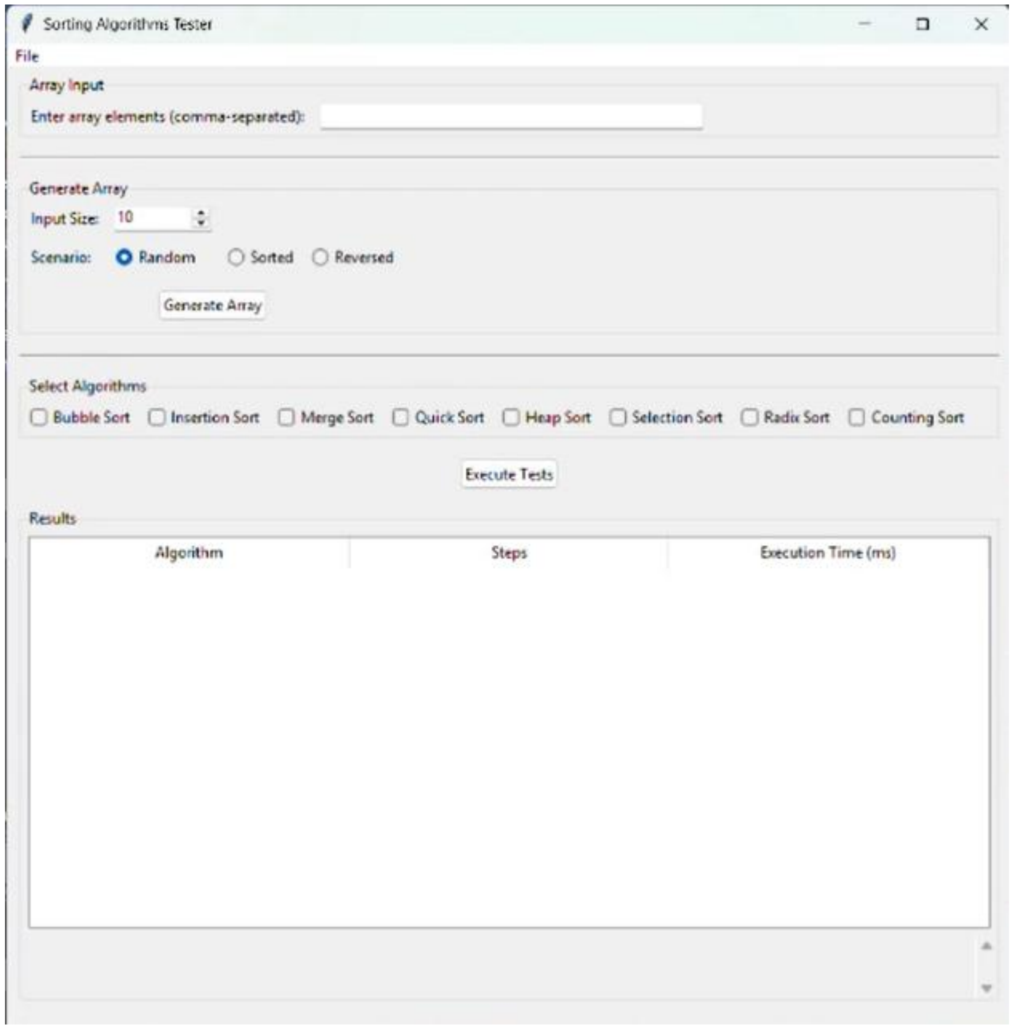
06 Step

We get the sorted array at the end.



Purpose of the code

The code provides comparison of the above algorithms in number of steps and execution time with different array sizes using bar graphs and asymptotic notation graphs and the user can choose between algorithms to compare, can choose array size, enter their values and also choose if the array is random, sorted, or reversed.


Sorting Algorithms Tester
File
Array Input
Enter array elements (comma-separated):
Generate Array
Input Size: 10
Scenario: [✓ Random] Sorted Reversed
Generate Array
Select Algorithms
Bubble Sort
Insertion Sort
Merge Sort
Quick Sort
Heap Sort
Selection Sort
Radix Sort
Counting Sort
Execute Tests
Results
Algorithm
Steps
Execution Time (ms)

Code Implementation

We divided the code into 5 main parts, we will discuss each part individually.

First Part

Constants and Initializations:

Purpose:
This part defines constants and initial values that will be used throughout the program. Here's a breakdown of its components:

1. **NUM_TEST_CASES:**
    - **Specifies the number of test cases for sorting.**
    - **Here, NUM_TEST_CASES = 15, meaning the program will run 15 test cases with varying input sizes.**

2. **INPUT_SIZES:**
    - **Lists the sizes of arrays that will be tested in each test case.**
    - **For example, the first test case will use an array of size 10, the second size 20, and so on.**

3. **Scenarios (RANDOM, SORTED, REVERSED):**
    - Defines the three types of input scenarios:
        - RANDOM: Randomly shuffled arrays.
        - SORTED: Arrays sorted in ascending order.
        - REVERSED: Arrays sorted in descending order.

4. **algorithm_names:**
    - A list of sorting algorithms that will be tested, including:
        - Bubble Sort
        - Insertion Sort
        - Merge Sort
        - Quick Sort
        - Heap Sort
        - Selection Sort
        - Radix Sort
        - Counting Sort
5. **algorithm_complexities:**
    - A dictionary mapping each sorting algorithm to its time complexity.
    - For example:
        - Bubble Sort: O(n^2)
        - Merge Sort: O(n logn)
        - Quick Sort: O(n logn) (average case), O(n^2) (worst case)

Main Function:
This part acts as the configuration section for the sorting performance tests. It defines:

1. The input sizes for the tests.
2. The scenarios (random, sorted, reversed) under which the algorithms will be evaluated.
3. The algorithms to be tested and their theoretical complexities for analysis.

Second Part:
Sorting Algorithms Implementation:

This part is well-discussed in the part of Sorting Algorithms but in the implementation we added to calculate number of steps of each algorithm

Main Functionality:
This module provides the core logic for sorting algorithms, focusing on:

1. Efficient implementation of algorithms.
2. Tracking the number of steps required for performance analysis.
3. Allowing integration with other parts of the program via algorithm_functions.

Third Part:
Plot Graphs:

This module is responsible for visualizing the performance of sorting algorithms based on the data generated during testing. It creates graphs to compare the sorting algorithms under different scenarios, input sizes, and performance metrics.

---

1. plot_combined_steps_and_execution_line_graph
Purpose:

- Generates line graphs for steps and execution time comparison for each input size and scenario.

Steps:

- Reads data from sorting_efficiency_comparison.csv.
- Filters data by input size and scenario.
- Plots two lines:
    - Blue line for the number of steps.
    - Red line for execution time.
- Saves the graphs in the combined_line_graphs directory.

---

2. plot_combined_steps_and_execution
Purpose:

- Creates bar graphs comparing steps and execution time for each input size and scenario.

Steps:

- Reads data from the CSV file.
- Creates two bar plots:
    - Left Y-axis: Number of steps (blue bars).
    - Right Y-axis: Execution time (red bars).
- Saves the graphs in the combined_graphs directory.

---

3. plot_sorting_comparison
Purpose:

- Generates bar graphs comparing the number of steps for all algorithms under each scenario and input size.

Steps:

- Uses seaborn to create bar plots.
- Organizes the data by scenario (sorted, random, reversed).
- Saves the plots in the Steps_graph directory.

---

4. plot_execution_time_graph
Purpose:

- Generates bar graphs comparing the execution time of sorting algorithms.

Steps:

- Reads data from the CSV file.
- Creates bar graphs for execution time under each scenario and input size.
- Saves the plots in the execution_time_graph directory.

---

5. plot_asymptotic_notation
Purpose:

- Plots theoretical asymptotic complexities for each sorting algorithm.

Steps:

- Defines theoretical functions for each algorithm based on input size (O(n), O(n^2), etc.).
- Compares scenarios (sorted, random, reversed).
- Saves graphs in the asymptotic_notation_graphs directory.

---

6. compare_algorithm_steps_and_onotation
Purpose:

- Compares empirical data (steps) with theoretical complexities.

Steps:

- Reads the CSV file for empirical data.
- Uses interpolation for smooth curves.
- Plots empirical steps and theoretical steps side by side.
- Saves results in the sorting_comparison directory.

---

7. plot_case_comparison_individual
Purpose:

- Compares best, worst, and average cases for each algorithm under different input sizes.

Steps:

- Groups data by algorithm, scenario, and input size.

- Creates line plots for:
    - Best case (sorted scenario).
    - Worst case (reversed scenario).
    - Average case (random scenario).
- Saves graphs in the case_comparison_graphs directory.

---

8. main Function
Purpose:
- Executes all plotting functions to generate comprehensive visualizations.

Steps:
1. **Generates:**
    - Execution time graphs.
    - Steps comparison graphs.
    - Combined graphs for steps and execution time.
    - Asymptotic complexity graphs.
2. **Uses data from sorting_efficiency_comparison.csv.**

Output
1. **Directories for Graphs:**
    - combined_line_graphs: Line plots for steps and execution time.
    - combined_graphs: Bar plots for steps and execution time.
    - Steps_graph: Bar plots for steps comparison.
    - execution_time_graph: Bar plots for execution time.
    - asymptotic_notation_graphs: Asymptotic complexity plots.
    - sorting_comparison: Comparison of empirical vs theoretical steps.
    - case_comparison_graphs: Best, worst, and average case comparisons.
2. **Graph Types:**
    - Line graphs.
    - Bar graphs.
    - Combined (dual Y-axis) graphs.

Main Functionality
- This module is the visualization engine for sorting performance:
    - Compares algorithms under different conditions (input size, scenarios).
    - Relates empirical results to theoretical complexities.
- Provides clear, detailed visual insights into the performance of sorting algorithms.

Fourth Part:
Generator:

This file orchestrates the generation of input data, execution of sorting algorithms, and the recording of performance results for analysis.

---

1. Array Generation Functions

Purpose: Generate input arrays for testing different sorting scenarios.

- generate_random_array(size: int):
  - Generates a random array of unique integers between 1 and size.
  - Example: For size=5, it might generate 3,1,5,2,4
- generate_sorted_array(size: int):
  - Generates a sorted array in ascending order.
  - Example: For size=5, it generates 1,2,3,4,5
- generate_reversed_array(size: int):
  - Generates a sorted array in descending order.
  - Example: For size=5, it generates 5,4,3,2,1

---

2. generate_input_array_file

Purpose: Save the generated input arrays to files for documentation and reproducibility.

- Creates a directory named input_arrays if it doesn't already exist.
- Writes each generated array to a file named according to the test case number and scenario.
  - Example: input_case_1_random.txt

---

3. run_sorting_test

Purpose: Executes a single sorting algorithm on a given input array and records performance metrics.

- Steps:
  1. Makes a copy of the input array to preserve the original.
  2. Measures execution time and counts steps using the selected sorting algorithm.
  3. Writes the results to a CSV file, including:
     - Algorithm name
     - Scenario (random, sorted, reversed)
     - Input size
     - Steps performed
     - Execution time (in milliseconds)
     - Asymptotic complexity

---

4. main Function

Purpose: The main workflow for generating data, running tests, and saving results.

- Steps:
  1. Opens a CSV file named sorting_efficiency_comparison.csv to store results.
  2. Iterates through each test case (NUM_TEST_CASES).
     - For each input size (INPUT_SIZES), generates arrays for all scenarios (sorted, random, reversed).
  3. For each scenario, runs all sorting algorithms defined in algorithm_names.
  4. Writes the performance data (steps, execution time, etc.) into the CSV file.
  5. Calls the plot_graphs.main() function to generate visualizations of the results.

Main Functionality

- Input Generation: Creates arrays for different test cases and scenarios.
- Sorting Execution: Runs each sorting algorithm on generated arrays.
- Performance Analysis:
  - Measures and records sorting steps and execution time.

    - Documents the performance results in a CSV file.

Output

1. **CSV File:**
    - sorting_efficiency_comparison.csv contains the performance data for all test cases.
    - Columns: Algorithm, Scenario, Input Size, Steps, Execution Time (ms), Asymptotic Notation.
2. **Input Files:**
    - Stored in the input_arrays directory, one file per test case and scenario.
3. **Visualizations:**
    - Graphs generated by plot_graphs.main() for comparing sorting algorithm performance.

Fifth Part:

GUI-App:

This part provides a Graphical User Interface (GUI) to:

1. Input arrays.
2. Select and execute sorting algorithms.
3. Display results and generate visualizations.

Key Functions and Features

1. User Interface Setup (setup_ui)

Purpose:

- Sets up the layout and components of the GUI.

Components:

1. Array Input:
    - Input array manually (comma-separated values).
2. Generate Array:
    - Options to generate arrays (random, sorted, reversed) of specified size.
3. Algorithm Selection:
    - Checkboxes to select sorting algorithms.
4. Results Display:
    - A table to show sorting results (steps, execution time).
5. Buttons:
    - Run tests, save results, or generate graphs.

---

2. Array Generation (generate_array)

Purpose:

- Generates arrays based on user-selected size and scenario.

Scenarios:

- Random: Randomly shuffled integers.
- Sorted: Ascending order.
- Reversed: Descending order.

Output:

- Updates the array input field with the generated array.

3. Execute Sorting Tests (execute_tests)

Purpose:

- Runs selected sorting algorithms on the input array.

Steps:

1. Parse the input array or use the generated one.
2. Check selected algorithms from the checkboxes.
3. Execute each algorithm:
    - Calculate steps and execution time.
4. Display results in the results table.

---

4. Graph Generation (plot_graph_in_gui, print_comparison_graphs, save_graph_as_png)

Purpose:

- Visualizes sorting performance directly in the GUI or saves graphs.

Graph Types:

1. Bar graphs for steps and execution time.
2. Combined graphs with dual Y-axes.

Saving Graphs:

- Allows users to save the graph as a PNG file.

---

5. Save Results (save_user_test_cases, save_results)

Purpose:

- Saves sorting results in a CSV file for later analysis.

Options:

- Save test results directly.
- Append user-generated test cases to an existing file.

---

6. Multithreading (run_generator)

Purpose:

- Runs the generator.py module in a separate thread to avoid freezing the GUI.

---

7. Main Function

Purpose:

- Initializes the GUI and starts the application.

Steps:

1. Creates the root Tkinter window.
2. Calls setup_ui to design the interface.
3. Launches the generator thread.

Main Functionalities

1. Interactive Input:
    - Enter or generate arrays for sorting.
2. Custom Selection:
    - Choose algorithms to test.
3. Performance Analysis:
    - Display steps and execution time for each algorithm.

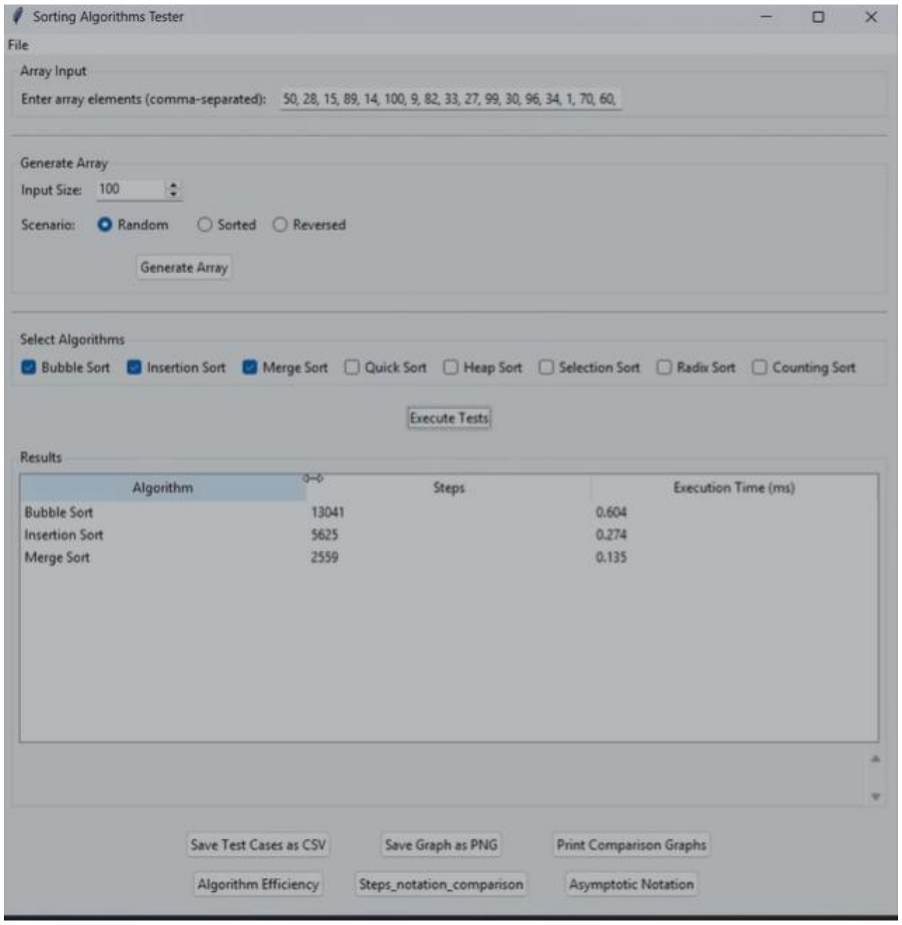
4. Data Visualization:
    - Generate and save graphs for results.
5. User-Friendly Interface:
    - Simple, intuitive layout with helpful messages.

Example Usage Flow

1. Enter or generate an array.
2. Select sorting algorithms to test.
3. Click "Execute Tests" to run and view results.
4. Save results or graphs for documentation.

Test Case

In this test we chose to make random array and the app made an array and then we chose sorting algorithms that we need to establish and the results are every algorithm, its number of steps and its execution time and then we have 3 options first one to save test case as CSV, second is to save a file as pgn, third is to print comparison graphs, fourth to print algorithm efficiently graphs, fifth to print step notation comparison and the last one to print asymptotic notation graphs.



Sorting Algorithms Tester

Array Input

Enter array elements (comma-separated): 50, 28, 15, 89, 14, 100, 9, 82, 33, 27, 99, 30, 96, 34, 1, 70, 60,

Generate Array

Input Size: 100

Scenario: [✓ Random] Sorted Reversed

Select Algorithms

[✓ Bubble Sort] [✓ Insertion Sort] [✓ Merge Sort] Quick Sort Heap Sort Selection Sort Radix Sort Counting Sort

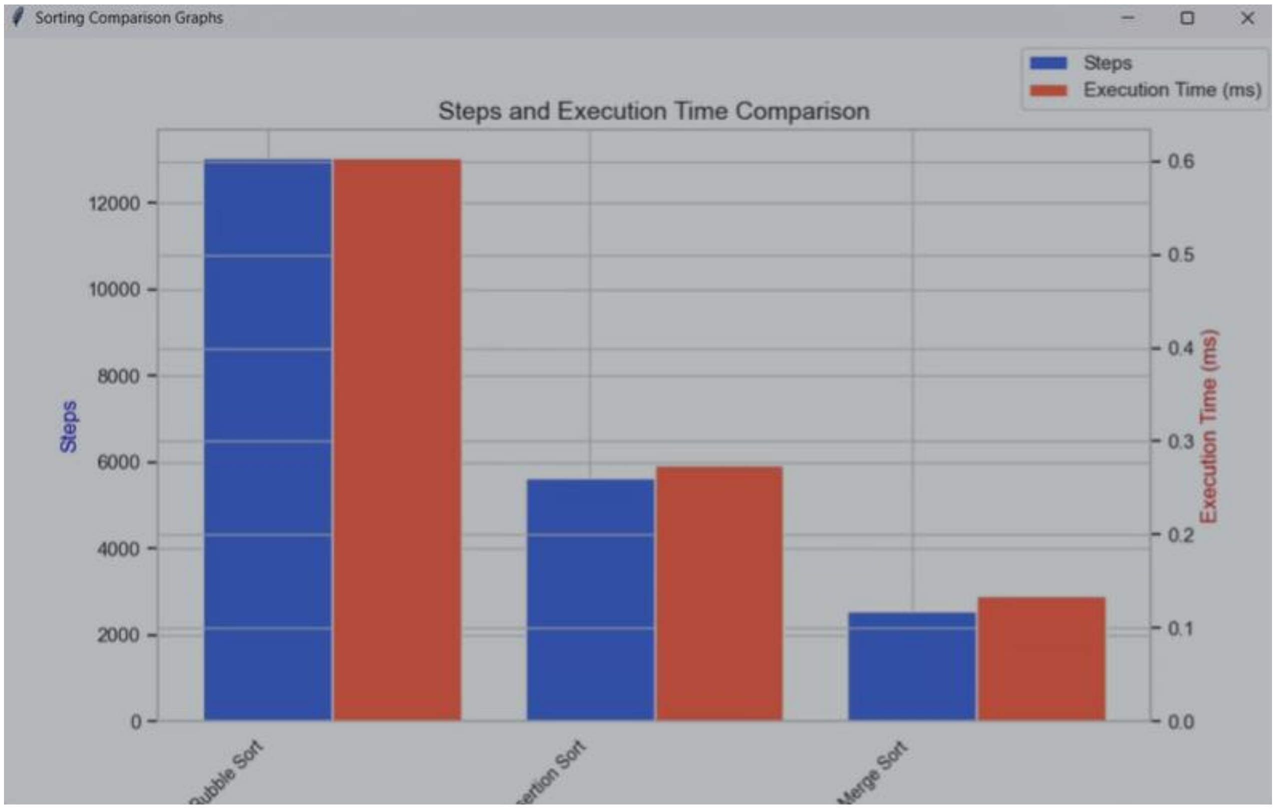
Results

Algorithm	Steps	Execution Time (ms)
Bubble Sort	13041	0.604
Insertion Sort	5625	0.274
Merge Sort	2559	0.135

1- CSV file

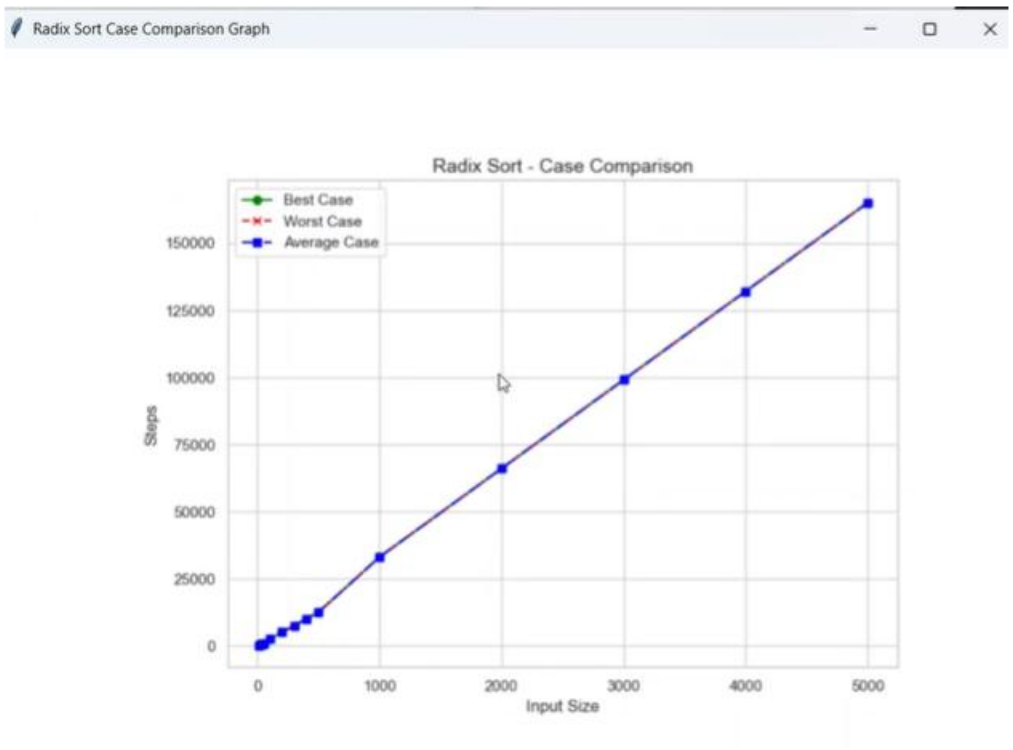
2- PGN

3- Comparison graphs



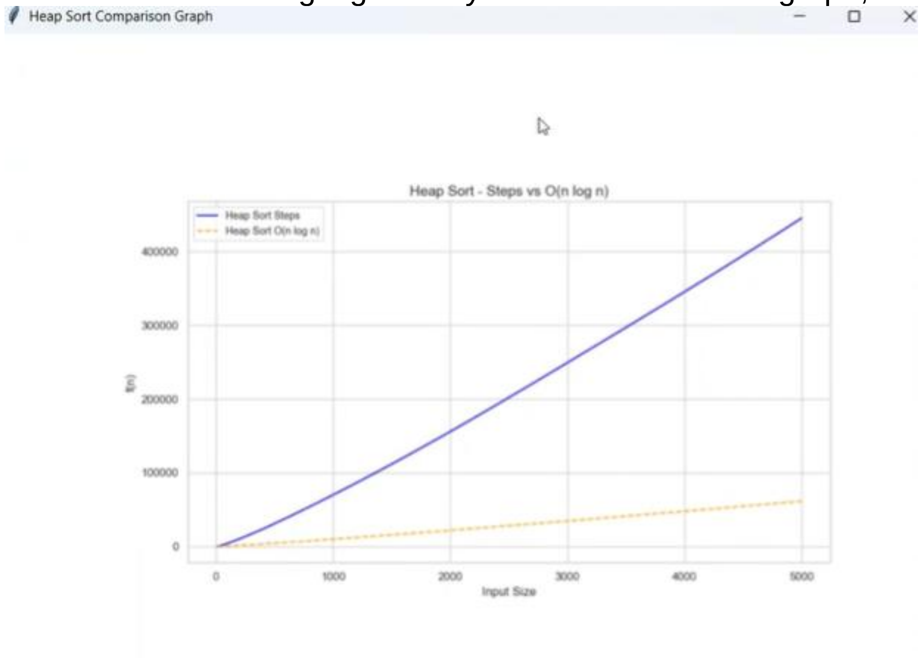
4- algorithm efficiently

- you choose which sorting algorithm you want to make the graph, I chose radix sort.



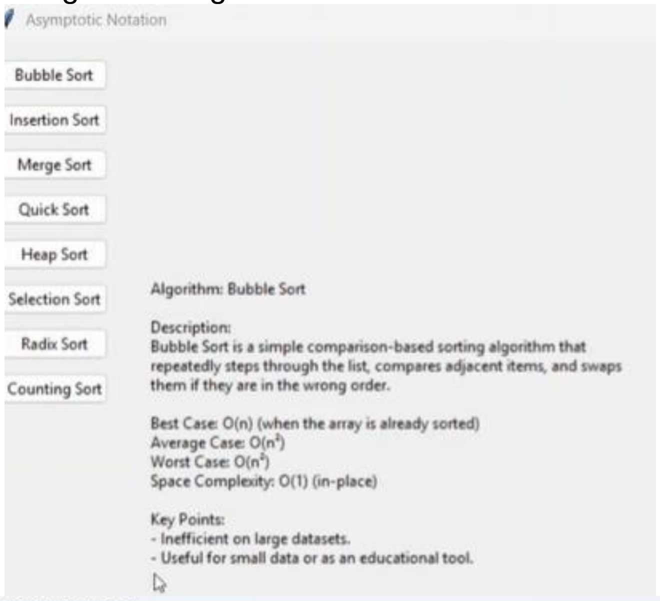
1- step notation comparison

you choose which sorting algorithm you want to make the graph, I chose heap sort.



2- asymptotic notation

You choose which one to make the graph, I chose Bubble sort and also description for the algorithm is generated.

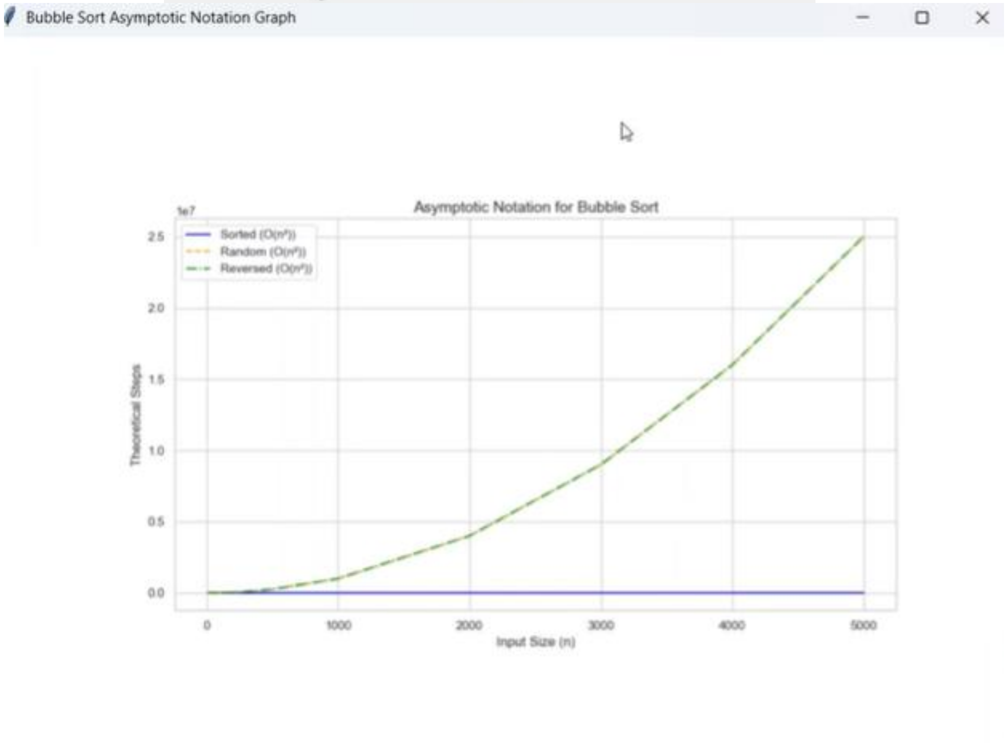


Algorithm: Bubble Sort

Description:
Bubble Sort is a simple comparison-based sorting algorithm that repeatedly steps through the list, compares adjacent items, and swaps them if they are in the wrong order.

Best Case: O(n) (when the array is already sorted)
Average Case: O(n^2)
Worst Case: O(n^2)
Space Complexity: O(1) (in-place)

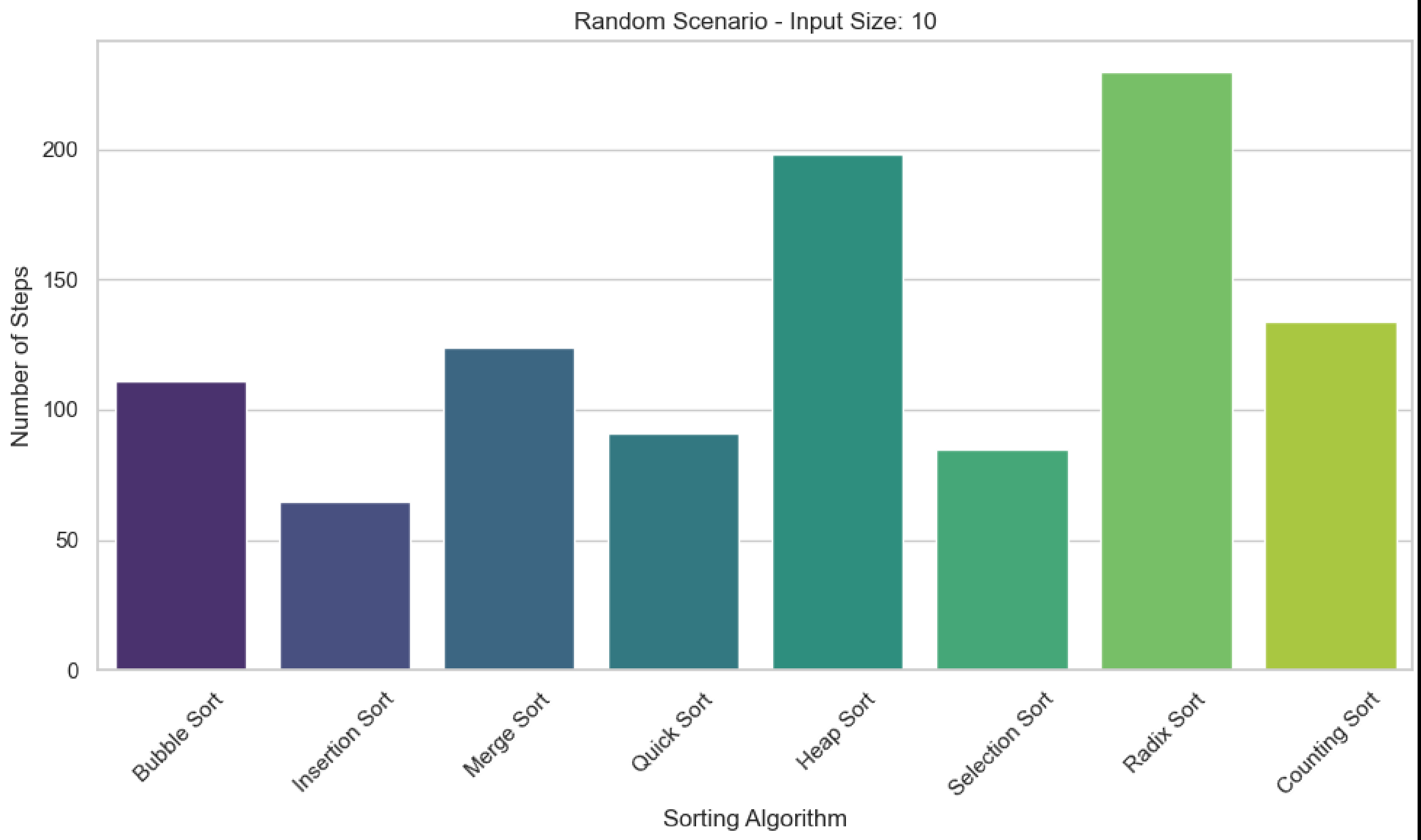
Key Points:
- Inefficient on large datasets.
- Useful for small data or as an educational tool.



Example for each graph type

Let’s see a comparison of all sorting algorithms of an array of size 10

1- Steps graph



Graph Insights

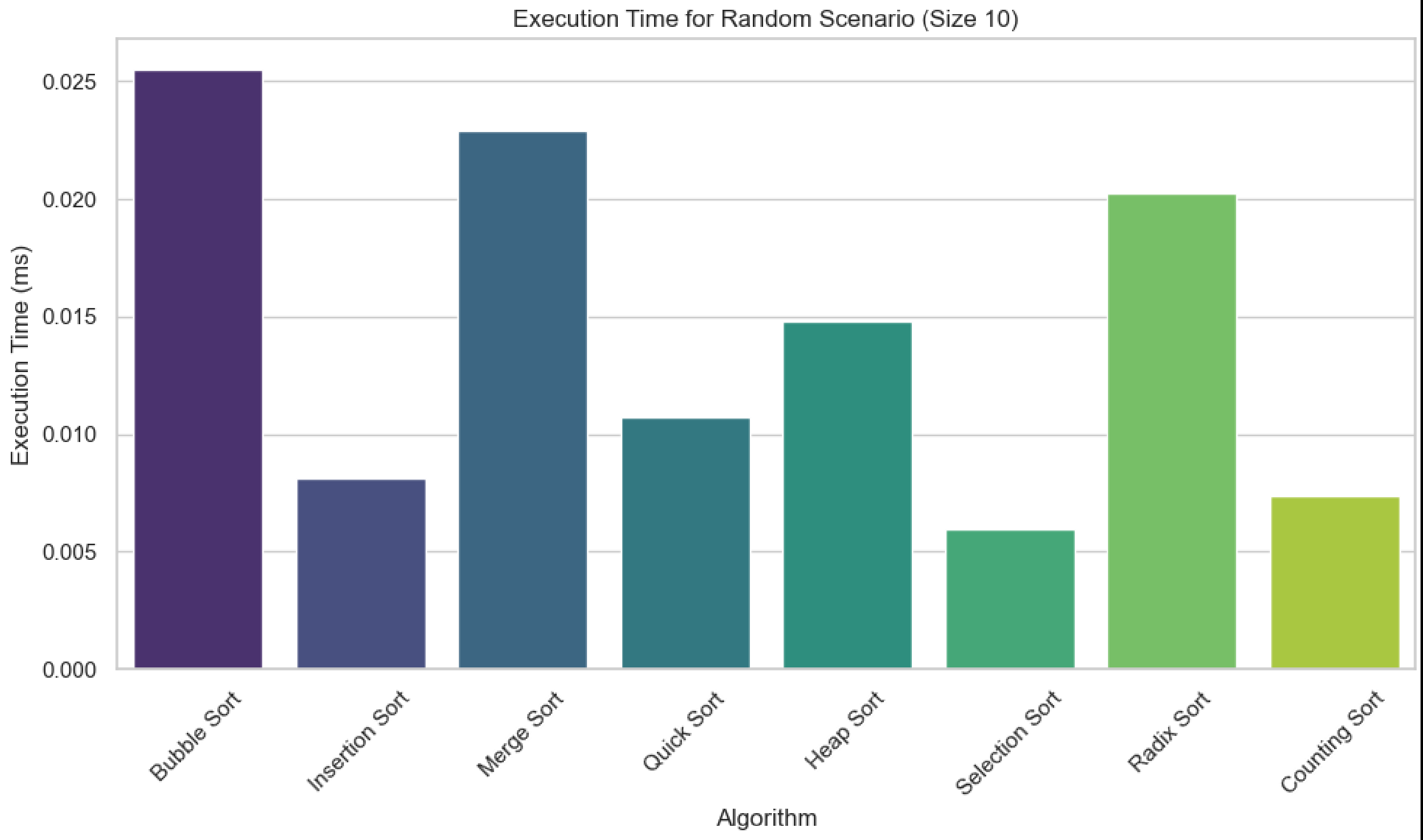
1. **Bubble Sort:**
   - Requires a significant number of steps (~100).
   - The inefficiency arises from its repeated comparison and swapping of adjacent elements, resulting in O(n^2) complexity.
2. **Insertion Sort:**
   - Performs better than Bubble Sort with fewer steps (~50).
   - Efficient for smaller datasets or nearly sorted arrays but still quadratic O(n^2)for random data.
3. **Merge Sort:**
   - Requires around 70 steps.
   - Uses a divide-and-conquer strategy, leading to efficient O(nlogn) performance even for random data.

4. **Quick Sort:**
    - Takes fewer steps (~60), showcasing its average-case O(nlogn) efficiency.
    - Performance depends on pivot selection; random arrays often yield good pivots.
5. **Heap Sort:**
    - Requires the most steps (~170).
    - Despite its O(nlogn) complexity, heap construction and extraction phases contribute to a higher step count.
6. **Selection Sort:**
    - Steps (~80) are slightly lower than Bubble Sort but still quadratic O(n^2).
    - Repeatedly finds the minimum value and places it in its correct position**.**
7. **Radix Sort:**
    - Surprisingly requires the highest number of steps (~200).
    - Its performance depends on the number of digits processed O(n·d), which can become costly for small arrays with multiple-digit elements.
8. **Counting Sort:**
    - Performs efficiently (~120 steps), as it directly uses a frequency count.
    - Its O(n+k) complexity makes it suitable for small arrays within a limited range.

Key Observations

- **Efficient Algorithms:**
    - Quick Sort and Merge Sort perform well for random arrays due to their O(nlogn) efficiency.
- **Inefficient Algorithms:**
    - Bubble Sort and Selection Sort show poor performance, requiring many steps for small arrays.
    - Heap Sort has a surprisingly high step count, likely due to the overhead of maintaining the heap structure.
- **Algorithm Suitability:**
    - Quick Sort and Merge Sort are recommended for general-purpose sorting.
    - Counting Sort and Radix Sort can be effective for specific datasets with bounded values.

2- Execution Time graph



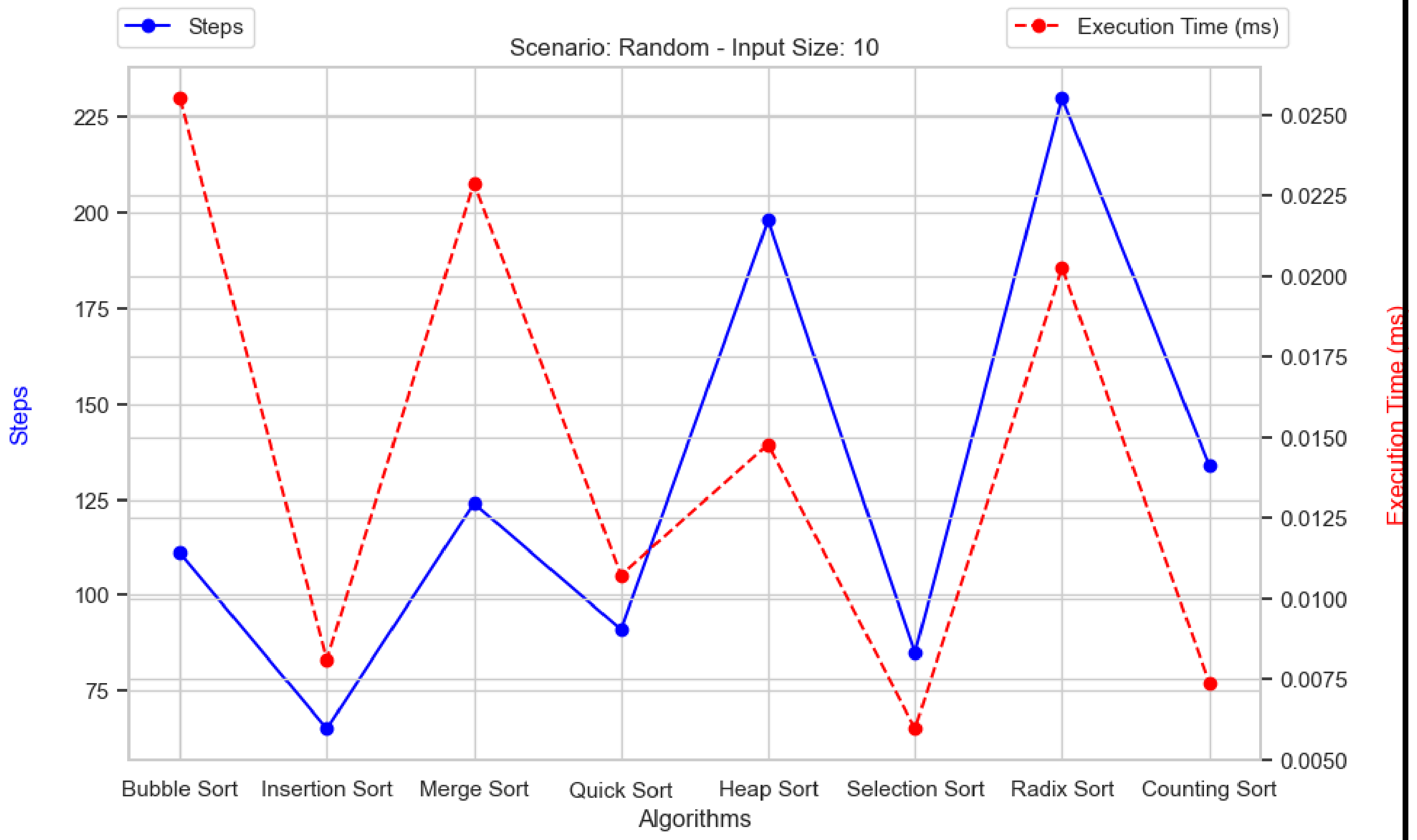
1. **Bubble Sort:**
   - Has the highest execution time (~0.025 ms).
   - This reflects its inefficiency for even small datasets due to its O(n^2) complexity.
2. **Insertion Sort:**
   - Performs better than Bubble Sort, with a significantly lower execution time (~0.008 ms).
   - Its execution time remains relatively efficient for small arrays, especially if partially sorted.
3. **Merge Sort:**
   - Requires around ~0.02 ms.
   - Despite its O(nlogn) complexity, the overhead of recursive calls and merging contributes t its time.
4. **Quick Sort:**
   - Shows better execution time (~0.012 ms) compared to Merge Sort, highlighting its efficient in-place sorting.
   - Its performance depends on the pivot selection, with random scenarios often yielding favorable results.
5. **Heap Sort:**
   - Has a slightly higher execution time (~0.015 ms).
   - The heap construction and repeated extraction phases add to the overhead.

6. **Selection Sort:**
   - Shows a lower execution time (~0.006 ms) compared to Bubble Sort, but it still struggles with O(n^2) complexity.
7. **Radix Sort:**
   - Surprisingly has a higher execution time (~0.02 ms), indicating that its digit-wise processing introduces overhead for small arrays.
   - It performs better with larger datasets or simpler inputs.
8. **Counting Sort:**
   - Has one of the lowest execution times (~0.01 ms), thanks to its O(n+k) complexity.
   - Efficient for arrays with small ranges of values, making it a good choice for this input size.

Key Observations

1. **Efficient Algorithms:**
   - Insertion Sort, Quick Sort, and Counting Sort demonstrate efficient execution times, making them suitable for small arrays.
   - Counting Sort performs exceptionally well for numeric data with small ranges.
2. **Inefficient Algorithms:**
   - Bubble Sort has the highest execution time due to its repeated comparison and swapping.
   - Radix Sort performs worse than expected, likely due to overhead in processing multiple digits for a small array.
3. **Best Choices:**
   - For small arrays:
     - Use Counting Sort or Quick Sort for efficiency.
     - Avoid Bubble Sort and Radix Sort unless specifically required.

3- Combined line graph



Blue Line (Steps)

- Represents the number of steps each algorithm takes to sort the array.
- Peaks and valleys illustrate the variation in efficiency:
  - Radix Sort and Heap Sort have the highest step counts.
  - Insertion Sort and Counting Sort have significantly lower step counts.

Red Line (Execution Time)

- Represents the actual time taken to execute the sorting in milliseconds.
- Peaks and dips reflect computational overhead:
  - Bubble Sort and Radix Sort show the highest execution times.
  - Counting Sort and Selection Sort are the fastest.

Detailed Analysis

1. Bubble Sort:
   - High number of steps (~225) corresponds to its O(n^2) complexity.
   - Also has the highest execution time (~0.025 ms), making it inefficient.
2. Insertion Sort:
   - Lowest step count (~75) and low execution time (~0.007 ms).

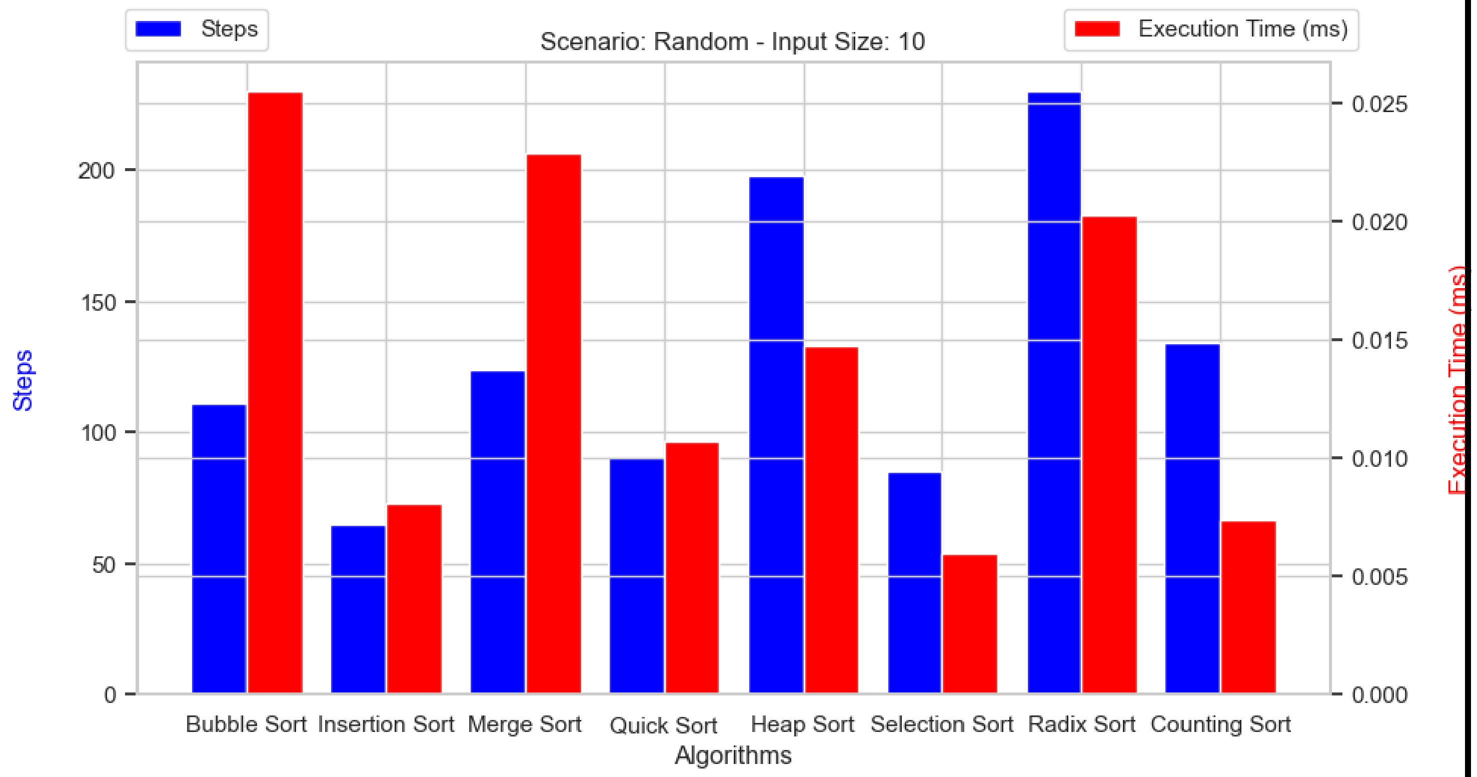
- Performs well for small datasets, reflecting its efficiency for partially sorted arrays.

3. Merge Sort:
   - Moderate step count (~125) with higher execution time (~0.02 ms).
   - The recursive merging contributes to both metrics.
4. Quick Sort:
   - Similar step count (~110) and execution time (~0.012 ms) as Merge Sort.
   - Efficient pivot selection keeps steps and time low for random arrays.
5. Heap Sort:
   - High step count (~200) due to heap construction and sorting phases.
   - Moderate execution time (~0.015 ms) despite its O(nlogn) complexity.
6. Selection Sort:
   - Moderate step count (~125) but surprisingly low execution time (~0.006 ms).
   - Likely benefits from straightforward swapping without recursion.
7. Radix Sort:
   - Highest step count (~225) due to digit-wise processing O(n·d)
   - Also has high execution time (~0.02 ms), indicating overhead for small arrays.
8. Counting Sort:
   - Among the lowest step counts (~100) and low execution time (~0.01 ms).
   - Efficient for small arrays with bounded value ranges.

Key Observations

- Efficient Algorithms:
  - Counting Sort and Insertion Sort demonstrate both low steps and low execution time, making them ideal for small datasets.
- Inefficient Algorithms:
  - Bubble Sort and Radix Sort exhibit high steps and execution time, reflecting inefficiency fo small arrays.
- General Insights:
  - Quick Sort and Merge Sort strike a balance between steps and execution time, performing well for general scenarios.

4- Combined bar graph



Blue Bars (Steps)

- Indicate the number of steps each algorithm performs to complete the sorting.
- Peaks and troughs highlight the variation in algorithm efficiency:
  - Radix Sort and Heap Sort show the highest number of steps.
  - Insertion Sort and Counting Sort demonstrate the lowest number of steps.

Red Bars (Execution Time)

- Represent the time (in milliseconds) taken by each algorithm to sort the array.
- Peaks and dips reflect differences in computational overhead:
  - Bubble Sort and Radix Sort exhibit the highest execution times.
  - Selection Sort and Counting Sort have the fastest execution times.

Detailed Analysis

1. Bubble Sort:
   - High step count (~225) reflects its O(n^2) inefficiency.
   - The highest execution time (~0.025 ms) indicates its unsuitability for even small arrays.
2. Insertion Sort:
   - Low step count (~75) and execution time (~0.007 ms) make it efficient for small datasets, especially when partially sorted.
3. Merge Sort:
   - Moderate step count (~125) but higher execution time (~0.02 ms).
   - Recursive operations contribute to increased time despite its O(nlogn) complexity.

4. Quick Sort:
   - Balanced performance with a step count (~110) and execution time (~0.012 ms).
   - Efficient pivot selection enables its competitive execution time.
5. Heap Sort:
   - High step count (~200) due to heap maintenance overhead.
   - Execution time (~0.015 ms) reflects its efficient O(nlogn) performance.
6. Selection Sort:
   - Moderate step count (~125) but surprisingly low execution time (~0.006 ms).
   - Its straightforward logic benefits small datasets.
7. Radix Sort:
   - High step count (~225) from digit-by-digit processing O(n·d).
   - High execution time (~0.02 ms) suggests inefficiency for small arrays with multi-digit numbers.
8. Counting Sort:
   - Among the lowest step counts (~100) and execution times (~0.01 ms).
   - Ideal for arrays with small ranges due to its O(n+k) complexity.

Key Observations

1. Efficient Algorithms:
   - Counting Sort, Insertion Sort, and Quick Sort stand out for both low steps and low execution time.
2. Inefficient Algorithms:
   - Bubble Sort and Radix Sort demonstrate poor performance in both metrics.
3. Balanced Algorithms:
   - Merge Sort and Heap Sort provide consistent performance, though they involve additional overhead compared to Quick Sort.
4. Recommendations:
   - For small arrays, prioritize Counting Sort or Insertion Sort for efficiency.
   - Avoid Bubble Sort and Radix Sort unless required for specific use cases.

Conclusion

In this project, we explored the efficiency of eight different sorting algorithms—Bubble Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort, Selection Sort, Radix Sort, and Counting Sort—by analyzing their performance on arrays of size 10 under random scenarios. The evaluation was based on two key metrics: **number of steps required** and **execution time**.

Key Findings:

1. **Efficient Algorithms:**
    - Counting Sort and Insertion Sort consistently demonstrated low step counts and fast execution times, making them ideal for small datasets.
    - Quick Sort balanced efficiency and performance with its O(nlogn)O(n \log n)O(nlogn) complexity, achieving competitive results for both metrics.
2. **Inefficient Algorithms:**
    - Bubble Sort showed the highest execution time and step count, confirming its inefficiency due to O(n2)O(n^2)O(n2) complexity.
    - Radix Sort, despite being efficient for larger datasets, showed high step counts and execution times for smaller arrays, indicating overhead for processing multi-digit elements.
3. **Balanced Algorithms:**
    - Merge Sort and Heap Sort provided consistent performance but incurred additional computational overhead due to recursive operations and heap maintenance.
4. **Algorithm Suitability:**
    - For small datasets like the ones tested, Counting Sort, Insertion Sort, and Quick Sort are the most practical choices.
    - Bubble Sort and Radix Sort should be avoided unless specific scenarios necessitate their use.

Overall Impact:
This project highlights the importance of selecting the right sorting algorithm based on the size and characteristics of the dataset. While theoretical complexities provide a general understanding, the practical analysis of steps and execution time reveals the real-world trade-offs between different algorithms.

Future Work:

- Expanding the study to larger datasets and varied input scenarios (e.g., sorted and reversed arrays).
- Analyzing additional metrics, such as memory usage, to evaluate the algorithms' overall resource efficiency.

This project demonstrates how algorithm selection can significantly impact computational performance, providing valuable insights for both academic and practical applications.

Appendices

The link for the source code, the output files and the documentation in [Data Structure and Algorithms]