HO CHI MINH UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION AND TECHNOLOGY

Sorting Algorithms

Performance Analysis

Course: Data Structures and Algorithms

Authors:
Hoang, Truong Le Vu (24120056)
Khoa, Nguyen Minh (24120073)
Minh, Ngo Gia (24120095)
Nghi, To Lang (24120101)

Instructors:
M.s. Uyen, Phan Thi Phuong

April 2025

Contents

1 Information

1.1 Authors

This project was carried out by the following four authors.

Student's name (Vietnamese)	Student's ID
Trương Lê Vũ Hoàng	24120056
Nguyễn Minh Khoa	24120073
Ngô Gia Minh	24120095
Tô Lãng Nghị	24120101

Table 1: List of authors

1.2 Declaration

- Our group takes full responsibility for any issues related to this report.
- The content in this report is original work.
- Any unauthorized copying is strictly prohibited.
- All references are fully cited.

1.3 System specification

We measure the running time of sorting algorithm on the following computer specifications:

- OS: Window 11 home
- CPU: Intel core i7-12700H (up to 4.70 GHz, 24MB)
- RAM: 512GB PCIe 3.0 NVMe M.2 SSD
- VGA: NVIDIA GeForce RTX 3050 4GB GDDR6

Note: The running time may differ depending on the computer specifications.

2 Introduction

2.1 Content

Sorting is the process of ordering or arranging a given collection of elements in some particular order. It is a crucial problem not only in daily life, but especially in Computer Science.
From the beginning of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. Until now, there have been many sorting algorithms, each of which has its own **advantages** and **disadvantages**.
In this report, our group will present **12 sorting algorithms** (which include Selection Sort, Insertion Sort, Shell Sort, Bubble Sort, Heap Sort, Merge Sort, Quick Sort, Radix Sort, Counting Sort, Binary Insertion Sort, Shaker Sort, and Flash Sort) based on two main metrics: **comparison count** and **running time**. It is our hope that this report will be a useful tool for learners who are trying to study and apply sorting algorithms in practice.
In addition, We would also like to thank M.S. Phan Thi Phuong Uyen for providing helpful exercises that supported our learning of sorting algorithms.

2.2 Goals

- By completing this report, the authors of this project will have a comprehensive understanding of sorting algorithms.
- This report meets the requirements of the instructor.
- Offer useful resources on various sorting algorithms.

3 Algorithm presentation

3.1 Insertion Sort

3.1.1 Idea

The Insertion Sort algorithm resembles sorting playing cards. Think of picking one card at a time and putting it in the right-ordered spot. Similarly, the Insertion Sort partitions the array into two parts: sorted and unsorted. Repeatedly, the Insertion Sort takes each item from the unsorted region and inserts it into the correct order within the sorted one.[3]

3.1.2 Pseudo code

Algorithm 1 Insertion Sort algorithm

```
function InsertSort($A, n$)                    ▷ Sort array A containing n elements
    for $i \leftarrow 1$ to $n - 1$ do
        $j \leftarrow i - 1$
        key = A[i]
        while $j > 0$ and $A[j] > key$ do
            $A[j + 1] \leftarrow A[j]$
            $j \leftarrow j - 1$
        end while
        A[j + 1] = key
    end for
end function
```

3.1.3 Step-by-step description

1. **Key selection:**
   - Start from the second element A[1] and store it in the variable **key**.
2. **Comparison:**
   - Compare key with elements in the sorted part.
   - Shift elements greater than key forward to create space for the insertion.
3. **Insertion:**
   - Insert key into its correct position in the sorted part.
4. **Iteration:**
   - Repeat steps 1-3 until the array is sorted.

3.1.4 Visualization

Consider the input array is A={3, 2, 4, 1 ,5}. In the left is the sorted array and in the right are the elements that we need to insert into it. Iterate from the second element of the array till the end of the array.

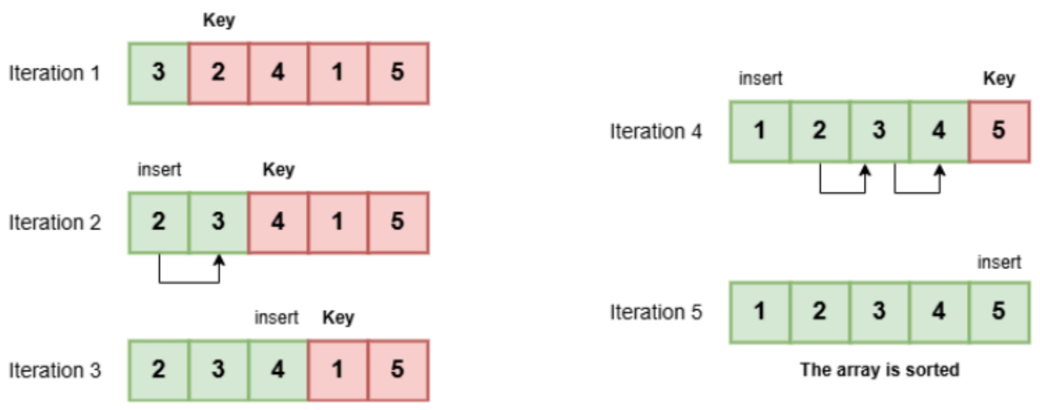


Figure 3.1.1: Insertion Sort visualization

3.1.5 Complexity Analysis

- Time Complexity
  - **Best Case:** $O(n)$
  Suppose the array is already sorted, each iteration causes no element sliding over since the key is greater than or equal to every element on its left.
  Hence, the number of basic operations is:

  $$C(n) = n - 1$$

  Order of Growth: $O(n)$
  - **Worst Case:** $O(n^2)$
  When the array is sorted in reverse order, every iteration at element i requires i − 1 comparisons because the key is always smaller than all the elements to its left.
  The number of basic operations:

  $$C(n) = 1 + 2 + 3 + \ldots + (n-1) = [n(n-1)]/2$$

  Order of Growth: $O(n^2)$
  - **Average Case:** $O(n^2)$
  Assume that in the average case, each element needs to be compared with roughly half of the previously sorted elements.
  The approximate number of basic operations would be:

  $$C(n) = [n(n-1)]/4$$

  Order of Growth: $O(n^2)$
  Almost Sorted Case: $O(n)$
  What if we knew that the array is "almost sorted"? Suppose every element in the array is at most a constant distance C away. In this case, over all $n - 1$ insertions, the total running time would be: $C \times (n-1)$, which is $O(n)$.
  [1] This means that insertion sort is an appropriate choice for small arrays.

- **Space Complexity:** $O(1)$ Insertion Sort is in-place, meaning it does not use any extra proportional space, so the space complexity is $O(1)$.

3.1.6 Variations

Binary Insertion Sort: 3.10

3.2 Selection Sort

3.2.1 Idea

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. [7]

3.2.2 Pseudo Code

Algorithm 2 Selection Sort algorithm

```
function SelectionSort($A, n$)                ▷ Sort array A containing n elements
    for $i \leftarrow 1$ to $n - 2$ do
        $minIndex \leftarrow i$
        for $j \leftarrow i + 1$ to $n - 1$ do
            if $A[j] < A[minIndex]$ then
                $minIndex \leftarrow j$
            end if
            Swap(A[i], A[minIndex])
        end for
    end for
end function
```

3.2.3 Step-by-step Description

1. Start with the first element in the array (let's call its position i).
2. Look through the rest of the array (from position i to the end) to find the smallest number.
3. Swap that smallest number with the number at position i.
4. Move to the next position (i = i + 1) and repeat the process.
5. Keep doing this until you've gone through the whole array.

3.2.4 Visualization

Consider:

- The blue color represents the current index.
- The yellow color indicates the index of the smallest element from the current index to n − 1.
- The gray color marks the sorted portion of the array, which remains unchangeable.

Input array:	5	7	3	9	1
Step 1:	5	7	3	9	1
Step 2:	1	7	3	9	5
Step 3:	1	7	3	9	5
Step 4:	1	3	7	9	5
Step 5:	1	3	7	9	5
Step 6:	1	3	5	9	7
Step 7:	1	3	5	9	7
Step 8:	1	3	5	7	9

Table 2: Visualization of Selection Sort.

3.2.5 Variations

- Heap sort is the enhancement of the selection sort, we will discuss about it in 3.5.

3.2.6 Complexity Analysis

- **Time complexity:** $O(n^2)$
  - The total number of operations for the Selection Sort algorithm:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-i-1) = \frac{n(n-1)}{2}$$

  - The algorithm is $O(n^2)$ time complexity in all case. This is because the algorithm doesn't have a break condition to stop the loop.
  - The number of swap operations is n-1.

- **Space complexity:**$O(1)$
  - Selection Sort is in-place algorithm, meaning it does not require additional memory or storage to perform the sorting.
  - The algorith only uses a variable to store the index of the smallest element and a temp variable which uses in the swap function.

3.3 Shell Sort

3.3.1 Idea

Shell Sort is an optimization of Insertion Sort. It allows the exchange of items that are far apart. The algorithm starts with a large gap and applies insertion sort on sub-arrays whose elements are a certain index distance (gap) apart. The gap is then gradually reduced, and when it reaches 1, the array is fully sorted.

3.3.2 Pseudo Code

Algorithm 3 Shell Sort algorithm

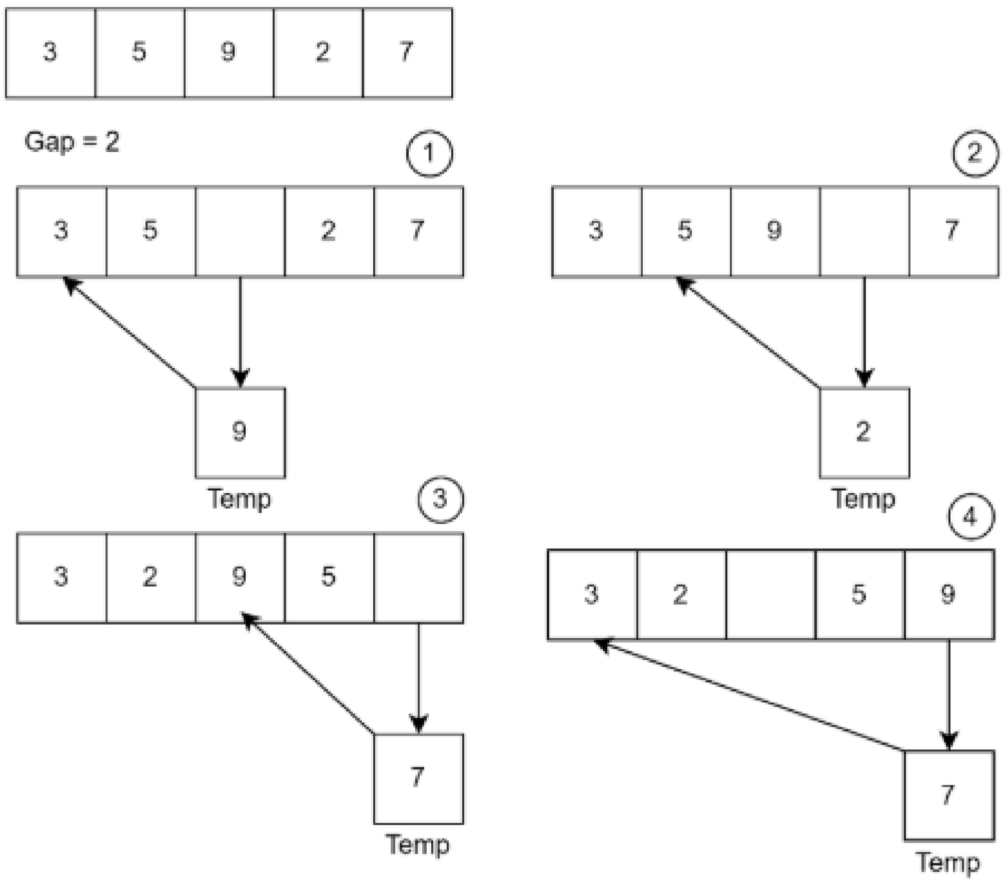
```
function shellSort($A, n$)                              ▷ Sort array A containing n elements
    $gap \leftarrow n/2$
    while $gap > 0$ do
        for $i \leftarrow gap$ to $n - 1$ do
            $temp \leftarrow list[i]$
            $j \leftarrow i$
            while $j \geq gap$ and $list[j - gap] > temp$ do
                $list[j] \leftarrow list[j - gap]$
                $j \leftarrow j - gap$
            end while
            $list[j] \leftarrow temp$
        end for
    end while
    $gap \leftarrow gap/2$
end function
```

3.3.3 Step-by-step Description

- Step 1: Initiate value of gap, the most basic and popular one is size / 2.
- Step 2: Initiate i = gap.
- Step 3: Initiate temp = list[i], j = gap.
- Step 4: Compare list[j – gap] with list[i], if list[i] < list[j – gap], move list[j – gap] to list[i].
- Step 5: Decrease j by gap; if j is non-negative, do step 4.
- Step 6: Increase i by 1; if i < size, do step 3.
- Step 7: divide the gap by 2, if the gap is non-negative do step 2.

3.3.4 Visualization

Consider that the input array is A={3 5 9 2 7} following the step-by-step description, the array will be sorted. Below is an illustration of the process.



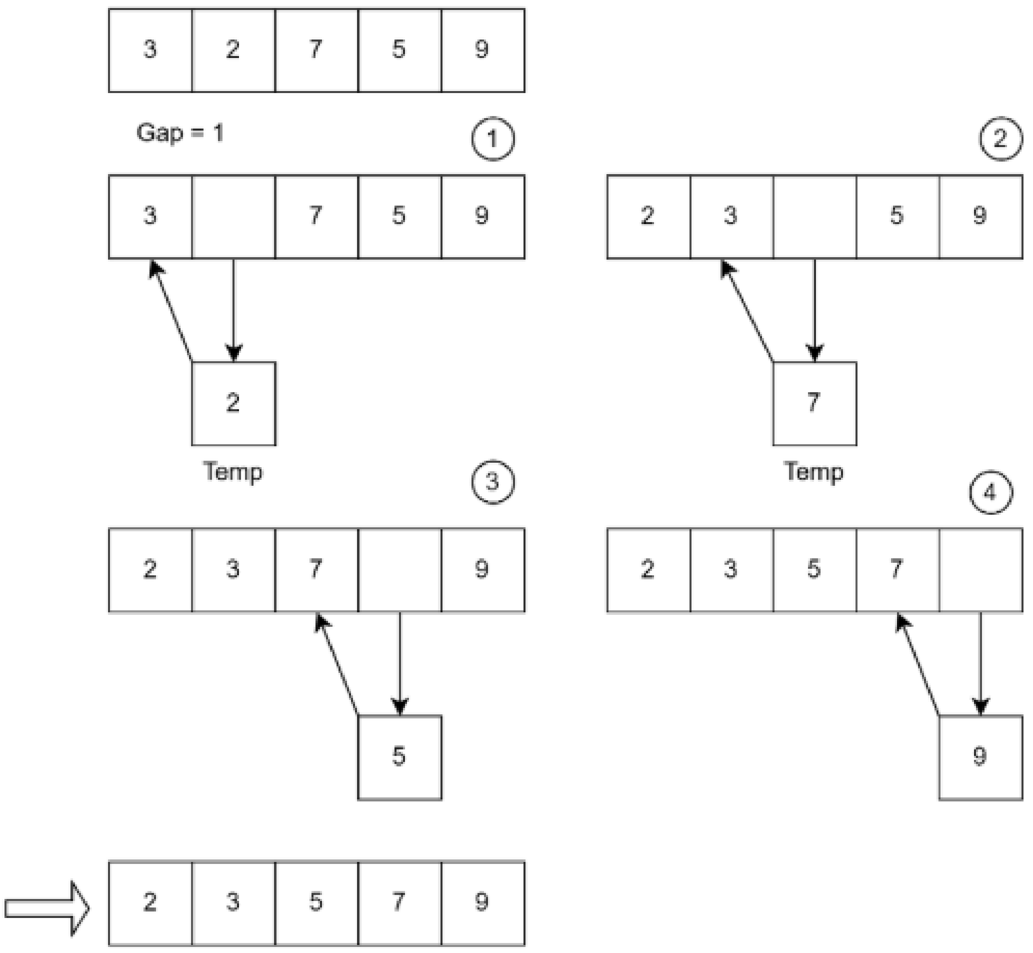


Figure 3.3.1: Shell Sort visualization

3.3.5 Variations

There are several ways that we can initiate the value of the gap. Some of them are more efficient than the basic one that we have introduced before. For example:

- (Basic and most popular)Shell: n/2, n/4, n/8, ... (n: size of array)
- Hibbard: $k \in N^*|(2^k - 1) < n(n : sizeofarray)$[8]
- Knuth: $k \in N^*|(3^k - 1)/2 < n$ (n: size of array)[8]
- Sedgewick: $k \in N$, Time Complexity: $O(N^{\frac{4}{3}})$ [8]
  If k even: element[k] $= 9 \times 2^k - 9 \times 2^{\frac{k}{2}} + 1$.
  If k odd: element[k] $= 8 \times 2^(k) - 6 \times 2^{\frac{k+1}{2}} + 1$

3.3.6 Complexity Analysis(Basic Version {n/2, n/4, n/8, ...})

- **Time Complexity**
  Worst case: This case occurs when the input array is in reverse sorted order or in some special cases that lead to poor partitioning. We have to use Insertion Sort with gap = 1 because the previous steps do not work efficiently.
  => The time complexity is: $O(n^2)$
  Example special case: [1,9,2,8,3,7,4,6,5], gaps{4, 2, 1};
  => Shell sort will not work efficiently until gap = 1 because min and max values are alternating.
- **Average case**: The average case time complexity of Shell Sort is approximately $O(nlogn)$

- **Best case**: This occurs when the array is already sorted.
  => Time complexity: $O(nlogn)$

3.4 Bubble Sort

3.4.1 Idea

Bubble Sort is a simple comparison-based sorting algorithm. It is based on the idea of repeatedly **comparing pairs of adjacent elements** and then swapping their positions if they exist in the wrong order.

3.4.2 Pseudo code

Algorithm 4 Bubble Sort algorithm

```
function bubbleSort($A, n$)                    ▷ Sort array A containing n elements
    for $i \leftarrow 0$ to $n-1$ do
        for $j \leftarrow 0$ to $n-i-1$ do
            if $A[j] > A[j+1]$ then
                swap($A[j], A[j+1]$)           ▷ Swap elements that are in wrong order
            end if
        end for
    end for
end function
```

3.4.3 Step-by-step description

1. Begin from the start of the array.
2. Compare each pair of adjacent elements.
3. Swap the elements if they are in the wrong orders.
4. Move to next pair and repeat step 2, step 3 until the end of the array.
5. Reduce the range of comparison by one and repeat step 1-4 until the range reach 0.

3.4.4 Visualization

Consider we have an array A = {5, 3, 4, 1, 2}. Following the step-by-step description we will have a sorted array. The table below illustrates the process of sorting array A using Bubble Sort.

Step	Array	Explanation
1	5 3 4 1 2	Initial array.
2	**3 5** 4 1 2	Compare then swap 5 and 3
3	3 **4 5** 1 2	Compare then swap 5 and 4
4	3 4 **1 5** 2	Compare then swap 5 and 1
5	3 4 1 **2 5**	Compare then swap 5 and 2
6	**3 4** 1 2 5	Compare 3 and 4, no swap
7	3 **4 1** 2 5	Compare then swap 4 and 1
8	3 1 **2 4** 5	Compare then swap 4 and 2
9	**1 3** 2 4 5	Compare then swap 3 and 1
10	1 **2 3** 4 5	Compare then swap 5 and 1
11	**1 2** 3 4 5	Compare 1 and 2, no swap
12	1 2 3 4 5	Finish iteration, the array is fully sorted

Table 3: Visualization of Bubble Sort.

3.4.5 Complexity Analysis

- **Time Complexity**
  The outer loop runs from $i = 0$ to $n-2$, resulting in $n-1$ iterations. As the outer loop is executed, the inner loop moves from $j = 0$ to $n-2-i$. As a result, the total number of comparisons can be

expressed as:

$$C(n) = \sum_{i=0}^{n-2} n - 2 - i$$

Simplifying the equation

$$C(n) = (n-1) + (n-2) + ... + 1 = \frac{(n-1)n}{2}$$

This simplifies to:

$$C(n) = \frac{n^2 - n}{2}$$

Therefore, the time complexity of Bubble Sort is: $O(n^2)$

- **Space Complexity**
  Bubble Sort is an in-place sorting so its time complexity is: $O(1)$

3.5 Heap Sort

3.5.1 Idea

Before learning Heap Sorting, we should know about the heap. A heap can be defined as a binary tree with keys assigned to its nodes, one key per node, provided the following two conditions are met:

- The shape property—the binary tree is essentially complete (or simply complete), i.e., all its levels are full except possibly the last level, where only some rightmost leaves may be missing.
- The parental dominance or heap property—the key in each node is greater than or equal to the keys in its children. (This condition is considered automatically satisfied for all leaves). This is called the max-heap property.
- There are other variations as well, such as min-heap

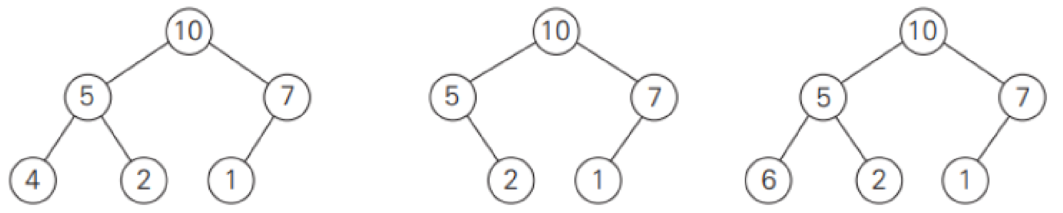


Figure 3.5.1: Example of heap

Here is a list of important properties of heaps:

- The root of a heap always contains its largest element.
- A node of a heap considered with all its descendants is also a heap.
- There exists exactly one essentially complete binary tree with n nodes. Its height is equal to log2(n)
- Heap' s array-based representation: Heap is a collection of n elements (a0, a1, . . . , an-1) in which every element at (at position i) in the first half is greater than or equal to the elements at position 2i + 1 and 2i+2.

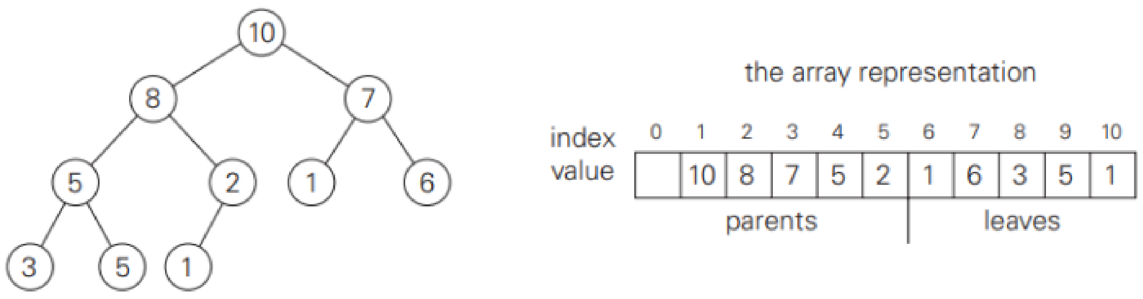


Figure 3.5.2: Heap and its array representation

Now we can discuss about Heap Sort - an interesting sorting algorithm discovered by J.W.J. Williams (in 1964). Idea is the same as Selection Sort. It has two stages:

- Stage 1: Construct a heap for a given array.
- Stage 2: Apply the maximum key deletion n-1 times to the remaining heap => Exchange the first and the last element of the heap. => Decrease the heap size by 1. => Rebuild the heap at the first position.

3.5.2 Pseudo Code

Algorithm 5 Heap Sort algorithm

```
function heapConstruct($A, n$)
    for $i \leftarrow n/2 - 1$ to 0 do heapRebuild(i, A, n)
    end for
end function
function heapRebuild($i, A, n$)
    $largest \leftarrow i$
    $left \leftarrow 2 * i + 1$
    $right \leftarrow 2 * i + 2$
    if $left < n$ and $A[left] > A[largest]$ then
        $largest \leftarrow left$
    end if
    if $right < n$ and $A[right] > A[largest]$ then
        $largest \leftarrow right$
    end if
    if $largest \neq i$ then
        swap(a[i], a[largest]) heapRebuild(largest, A, n)
    end if
end function
function heapSort($A, n$)
    heapConstruct(A, n)                                    ▷ Stage 1
    $r \leftarrow n - 1$
    while $r > 0$ do                                       ▷ Stage 2
        swap(A[0], A[r])
        heapRebuild(0, A, r)
        $r \leftarrow r - 1$
    end while
end function
```

3.5.3 Step-by-step description

To fully understand the concept behind the problem, we should begin by examining the heapRebuild function. This function ensures that the subtree rooted at a given index i maintains the properties of a max-heap. Specifically, it checks whether the left and right children at positions 2i+1 and 2i+2 are smaller than or equal to the parent at index i. If one of the children is larger than the parent, the function swaps the parent with the larger child and recursively calls itself on the new position of the parent. This recursive process continues until the max-heap property is restored throughout the subtree. Once the heapRebuild function is well understood, we can move on to how heapSort is constructed. The heap sort algorithm works in three main steps:

- Step 1: We call the heapConstruct function to transform the input array into a max-heap of n elements. Because of the array-based representation of a heap, the function begins by calling heapRebuild starting from the middle of the array (index n/2 - 1) and works its way back to the beginning. This ensures that all subtrees are properly arranged as max-heaps.
- Step 2: We repeatedly swap the first element (the largest value in the heap) with the last element in the unsorted portion of the array. This places the largest value in its correct sorted position. After the swap, we reduce the size of the heap by one (ignoring the last element, which is now sorted), and call heapRebuild on the root to rebuild the max-heap.
- Step 3: We repeat the swap and heapRebuild process until all elements are sorted, and the heap is empty.

3.5.4 Visualization

Example: We input array: 4, 10, 3, 5, 7 , 11. There are two ways to present visualization of heapsort (Tree and Array):

- **Visualization of heap (Tree):**
  => Input array:

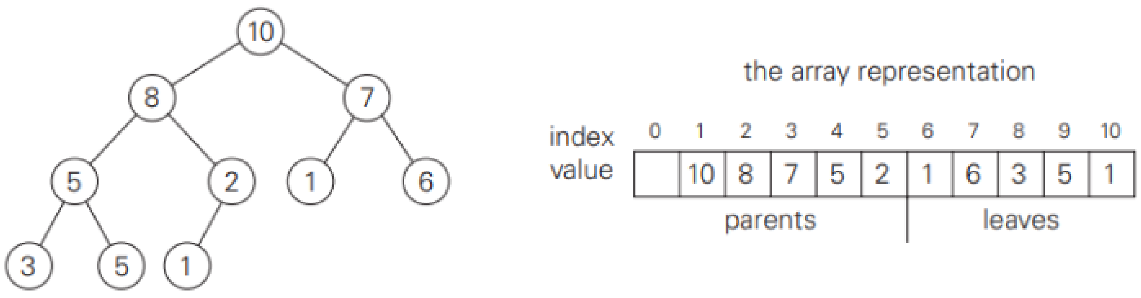


Figure 3.5.3: Heap and its array representation

=> Stage 1: Construct a heap for a given array.

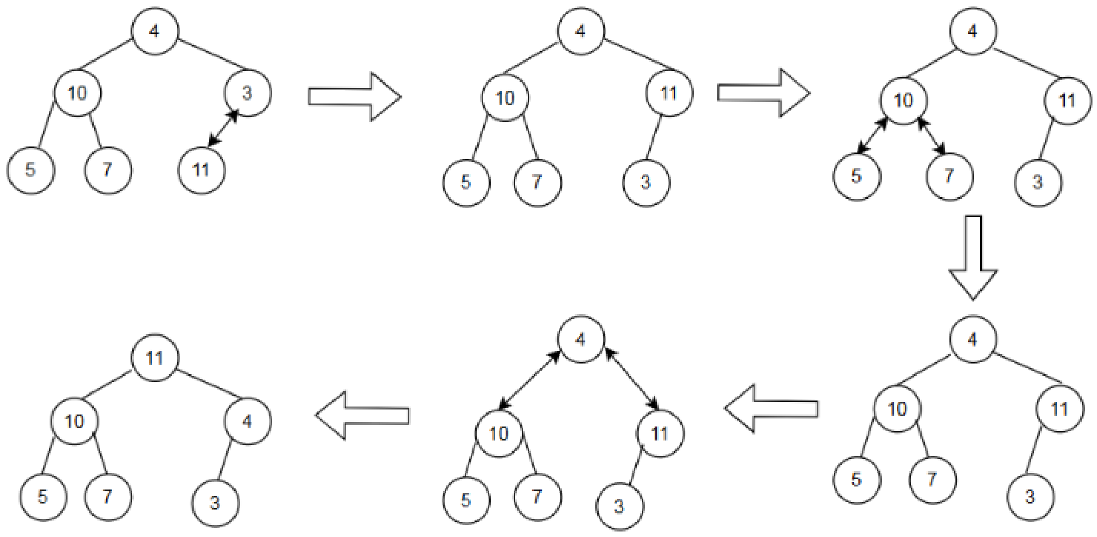


Figure 3.5.4: Construct heap

=> Stage 2: Apply the maximum key deletion n-1 times to the remaining heap

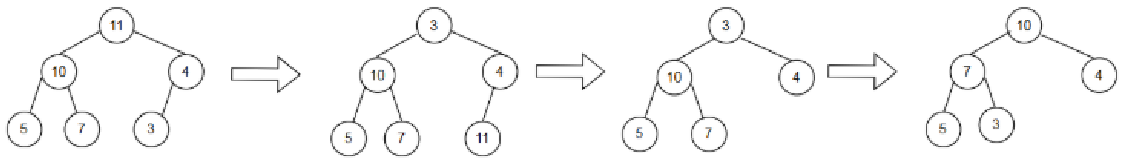


Figure 3.5.5: Construct heap

In this figure, - Step 1: Swap root node 11 with the last node 3 - Step 2: Consider the last node (11) as part of the sorted portion. - Step 3: Rebuild the max-heap with the remaining unsorted nodes. - Repeat the process untill there is no remaining unsorted node.

- **Visualization of heap (Array):** Input array: 4 10 3 5 7 11
  Stage 1: Construct a heap for a given array.

4	10	**3**	5	7	11
4	**10**	11	5	7	3
4	10	11	5	7	3
11	10	4	5	7	3

Table 4: Construction of heap

Stage 2: Apply the maximum key deletion n-1 times to the remaining heap.

11	10	4	5	7	3	
3	10	4	5	7	11	(i)
3	10	4	5	7	11	(ii)
10	7	4	5	3	11	(iii)
3	7	4	5	10	11	(i)
3	7	4	5	10	11	(ii)
7	5	4	3	10	11	(iii)
3	5	4	7	10	11	(i)
3	5	4	7	10	11	(ii)
5	3	4	7	10	11	(iii)
4	3	5	7	10	11	(i)
4	3	5	7	10	11	(ii)
4	3	5	7	10	11	(iii)
3	4	5	7	10	11	(i)

i. Exchange the first and the last element of the heap.

ii. Decrease the heap size by 1.

iii. Rebuild the heap at the first position.

3.5.5 Variations

The heap sort algorithm follows a similar idea to selection sort, both repeatedly select the largest element and place it in its correct position. However, heap sort is more efficient because it gives us a better overall time complexity.

3.5.6 Complexity Analysis

- **Time complexity:**
  => **heapConstruct** function: $O(n)$
  => **heapRebuild** function: $O(nlog(n))$
  => The heap sort algorithm has $O(nlog(n))$ time complexity in all case.

- **Space complexity:** $O(1)$, heap sort is an in-place algorithm, no extra require memory.

3.6 Merge Sort

3.6.1 Idea

Merge sort is a divide-and-conquer sorting algorithm that works by recursively dividing an array into smaller subarrays until each subarray only contains a single element. Since a single element is inherently sorted, the algorithm then combines these subarrays in sorted order. This continues until all subarrays are merged back into a single sorted array. [3]

3.6.2 Pseudo Code

Algorithm 6 Merge Sort Algorithm Part 1

```
function MERGESORT($A, left, right$)
    if $left < right$ then
        $mid \leftarrow (leffft + right)/2$
        mergeSort(A, left, mid)
        meregeSort(A, mid, right)
        merge(A, left, mid, right
    end if
end function
function MERGE(A, left, mid, right )
    $n1 \leftarrow mid - left + 1$
    $n2 \leftarrow right - mid$
    Create arrays L[0...n1-1] and R[0...n2-1]
    for $i \leftarrow 0$ to $n1 - 1$ do
        $L[i] \leftarrow A[left + i]$
    end for
    for $j \leftarrow 0$ to $n2 - 1$ do
        $R[j] \leftarrow A[mid + 1 + j]$
    end for
    $i \leftarrow 0$
    $j \leftarrow 0$
    $k \leftarrow 0$
    while $i < n1$ and $j < n2$ do
        if $L[i] \leq R[j]$ then
            $A[k] \leftarrow L[i]$
            $i \leftarrow i + 1$
        else
            $A[k] \leftarrow R[j]$
            $j \leftarrow j + 1$
        end if
        $k \leftarrow k + 1$
    end while
```

Algorithm 7 Merge Sort Algorithm Part 2

```
    while $i < n1$ do
        $A[k] \leftarrow L[i]$
        $i \leftarrow i + 1$
        $k \leftarrow k + 1$
    end while
    while $j < n2$ do
        $A[k] \leftarrow R[j]$
        $j \leftarrow j + 1$
        $k \leftarrow k + 1$
    end while
end function
```

3.6.3 Step-by-step Description

1. **Divide**: Divide the array into two halves until each subarray contains one element, which is the base case for merging.
2. **Merge** Create a temporary array to hold the merged result. Use three pointers: left array's pointer, right array's pointer, merged array's pointer.
   - Compare the elements of two merging arrays, place the smaller one into the temporary array, and move the corresponding pointer forward. This process continues until one of the subarrays is fully traversed.
   - When one subarray has been fully traversed, copy the remaining elements of the other subarray to the merged array.
3. **Recursion:** Repeat steps 1 and 2 until the original array is sorted.

3.6.4 Visualization

Consider the input array A={2, 4, 1, 6, 3 ,5}. Follow the process of dividing and merging, the array will be sorted. Below is the visualization of the process

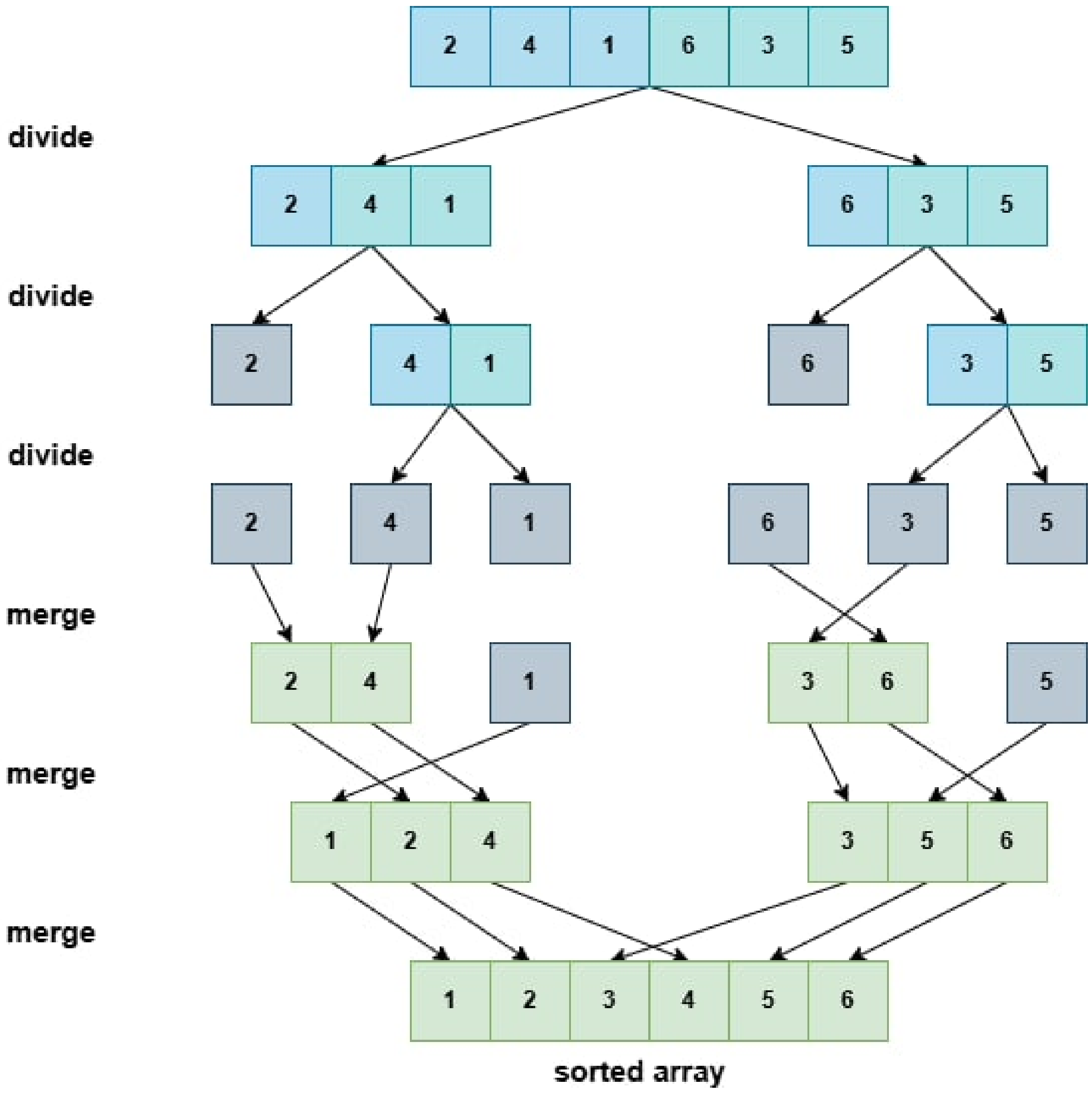


Figure 3.6.1: Merge Sort Visualization

3.6.5 Variations

- **Merge-Insertion Sort / Ford–Johnson algorithm:**
  Merge-Insertion Sort takes advantage of the efficiency of Insertion Sort for small arrays. This algorithm works by first grouping the input elements into pairs and sorting each pair, identifying the larger and smaller elements. The larger elements are then recursively sorted using the same method. After that, the smaller elements are inserted into the sorted list of larger elements in a particularly chosen order, typically using a binary insertion strategy, to minimize additional comparisons[2]
- **Tim Sort:**
  Timsort is a hybrid sorting algorithm created by Tim Peters in 2002. It is designed to perform efficiently on real-world data by utilizing the already ordered subsequences–called runs in the input array. Like the Ford-Johnson algorithm, it combines the ideas of Insertion Sort and Merge Sort for optimal performance.[5]

3.6.6 Complexity Analysis[4]

- **Time Complexity:**
  The time complexity of merge sort is $O(nlogn)$ in all scenarios—best, average, and worst cases. This is because the array is repeatedly divided in half and then merged back together, ensuring that the performance remains stable. This consistency makes merge sort predictable and reliable for handling various input conditions.
  Breakdown of the time complexity:
  Divide: The array is divided into two halves. This step takes constant time, O(1), for each level of recursion, but since the array is divided log n times, the total time for all divisions is $O(logn)$.
  Conquer: Each half is sorted recursively. Since the array is divided in half at each level, there are **log n** levels of recursion. At each level, the total number of elements to be sorted is **n**, so the total time for all levels is $n \times logn$, which is $O(nlogn)$.
  Combine: The sorted halves are merged. Merging two sorted arrays of size n/2 takes $O(n)$ time, and since there are **log n** levels of merging, the total time for merging is also $O(nlogn)$.
  Therefore, the overall time complexity of merge sort is $O(nlogn)$.

- **Space Complexity:**
  The space complexity of merge sort is $O(n)$ due to extra space allocated to hold the temporary arrays during the merging process. While the algorithm is recursive and entails some overhead from the recursive calls, this does not influence the overall space complexity, which remains linear to the size of the input array.

3.7 Quick Sort[9]

3.7.1 Idea

Quick Sort is a divide-and-conquer algorithm (like merge sort), it works by choosing a **pivot** element from the array and **partioning** the array into two sub-arrays, according to whether they are less than or greater than the pivot. Then, do the same operation recursively on the sub-arrays to the left and right side of the pivot element. This continues until the array is sorted. Quick Sort has different versions that use different approaches to selecting the pivot element.

3.7.2 Pseudo Code

3.7.3 Step-by-step Description

1. Choose a pivot element from the array (the pseudo code above initializes the pivot index p at the first element of the current partition).
2. Partition the array into two sub-arrays, with elements less than the pivot on the left side and elements greater than the pivot on the right side. Swap the pivot element to the middle of two sub-arrays.
3. Recursively repeating the partitioning step on the left and right sub-arrays until the array is sorted.
4. Combine the two sorted sub-arrays to obtain the final sorted array.

Algorithm 8 Quick Sort Algorithm

```
function PARTITION($A, first, last, pivot$)
    $firstS1 \leftarrow first$
    $firstUnknown \leftarrow first + 1$
    while $firstUnknown \leq last$ do
        if $A[firstUnknown] < A[pivot]$ then
            $swap(A[firstUnknown], A[lastS1 + 1])$
            $lastS1 \leftarrow lastS1 + 1$
        end if
        $firstUnknown \leftarrow firstUnknown + 1;$
    end while
    $swap(A[lastS1], A[pivot])$
    return lastS1
end function
function QUICKSORT($A, first, last$)
    if $first < last$ then
        $pivotIndex \leftarrow partition(A, first, last, 0)$
        quickSort(a, first, pivotIndex - 1)
        quickSort(a, pivotIndex + 1, last)
    end if
end function
```

3.7.4 Visualization

Consider we have input array A = {10, 15, 0, 5, 20}. Following the step-by-step description, choose the first elements as the pivot if element at firstUnknown is smaller than pivot. Swap element at firstUnknown with element next to lastS1. Continue until firstUnkown reaches the end of the array, then swap element at lastS1 with pivot. Repeat the same process with two sub-arrays before and after pivot.
The table below illustrate the sorting process.
Yellow cell represent elements that are in correct positions.

pivot, lastS1	firstUnknown			
10	15	0	5	20
pivot, lastS1		firstUnknown		
10	15	0	5	20
pivot	lastS1		firstUnknown	
10	15	0	5	20
pivot			lastS1	firstUnknown
10	0	5	15	20
pivot			lastS1	firstUnknown
5	0	10	15	20
pivot, lastS1	firstUnknown		pivot, lastS1	firstUnknown
5	0	10	15	20
pivot	lastS1		pivot	lastS1
0	5	10	15	20

Table 5: Quick Sort Visualization

3.7.5 Variations

- **Median-of-three pivot selection Quick Sort**: Compare the first, last and middle elements of the list and then choose the middle value as the pivot.
- **Dual-pivot Quick Sort**: Pick two elements from the array to be sorted (the pivots), partition the remaining elements into those less than the lesser pivot, those between the pivots, and those greater than the greater pivot, and recursively sort these partitions.

3.7.6 Complexity Analysis

- **Time Complexity:**
  Best case:
  This happens when the split occurs in the middle of every sub-arrays.

  $$C(n) = 2C(n/2) + n - 1, (n > 1)$$

  Order of growth: $O(nlogn)$
  Worst case:
  This happens when the pivot is continuously chosen poorly (the largest or smallest element in the array).

  $$C(n) = 1 + 2 + 3 + ... + (n - 1) = \frac{n \times (n-1)}{2}$$

  Order of growth: $O(n^2)$
  Average case:
  This happens when the pivot is uniform random from 0 to n - 1, averaging over all possible splits and noting that the number of comparisons for the partition is n - 1, the average number of comparisons over all permutations of the input sequence can be estimated accurately by solving the recurrence relation:

  $$C(n) = n - 1 + \sum_{i=0}^{n-1}(C(i) + C(n-i-1)) \approx 1.39nlogn$$

  Order of growth: $O(nlogn)$

3.8 Radix Sort

3.8.1 Idea

Radix Sort is a non-comparison sorting algorithm. The core is processing numbers digit by digit(or group of digit) by specific sequences(LSD or MSD). Radix Sort distributes the elements into buckets based on its value of digit. When all digits of max element are processed, the input list is fully sorted. It is one of the most efficient sorting algorithms for non-negative integers and strings(the original one).

3.8.2 Pseudo Code

(Version LSD, type integer, base 10)

Algorithm 9 Radix Sort Algorithm

```
function RADIXSORT($A, n$)                                  ▷ sort array A containing n elements
    $max \leftarrow A[0]$
    for $i \leftarrow 1$ to $n - 1$ do
        if $max < A[i]$ then $max \leftarrow list[i]$
        end if
    end for
    $exp \leftarrow 1$
    while $max/exp \neq 0$ do
        Initialize array $buckets$ of 10 empty lists
        for $i \leftarrow 0$ to $-1$ do
            $digit \leftarrow (A[i]/exp) \mod 10)$
            Add $A[i]$ to the end of $buckets[digit]$
        end for
        $index \leftarrow 0$
        for $i \leftarrow 0$ to 9 do
            for $j \leftarrow 0$ to $n - 1$ do
                $A[index] \leftarrow buckets[i][j]$
            end for
        end for
        $exp \leftarrow exp * 10$
    end while
end function
```

3.8.3 Step-by-step Description

- Step 1: Find max element in the input array.
- Step 2: Initiate 2d array Buckets[base][size].
- Step 3: Initiate int exp = 1; i = 0.
- Step 4: Initiate int digit = (list[i] / exp)
- Step 5: Push back list[i] to Buckets[digit].
- Step 6: If i < size, increase i by 1 and do step 4.
- Step 7: Copy the content of Buckets sequentially back into the input list.
- Step 8: If max / exp != 0, multiply exp by base and do step 2.

3.8.4 Visualization

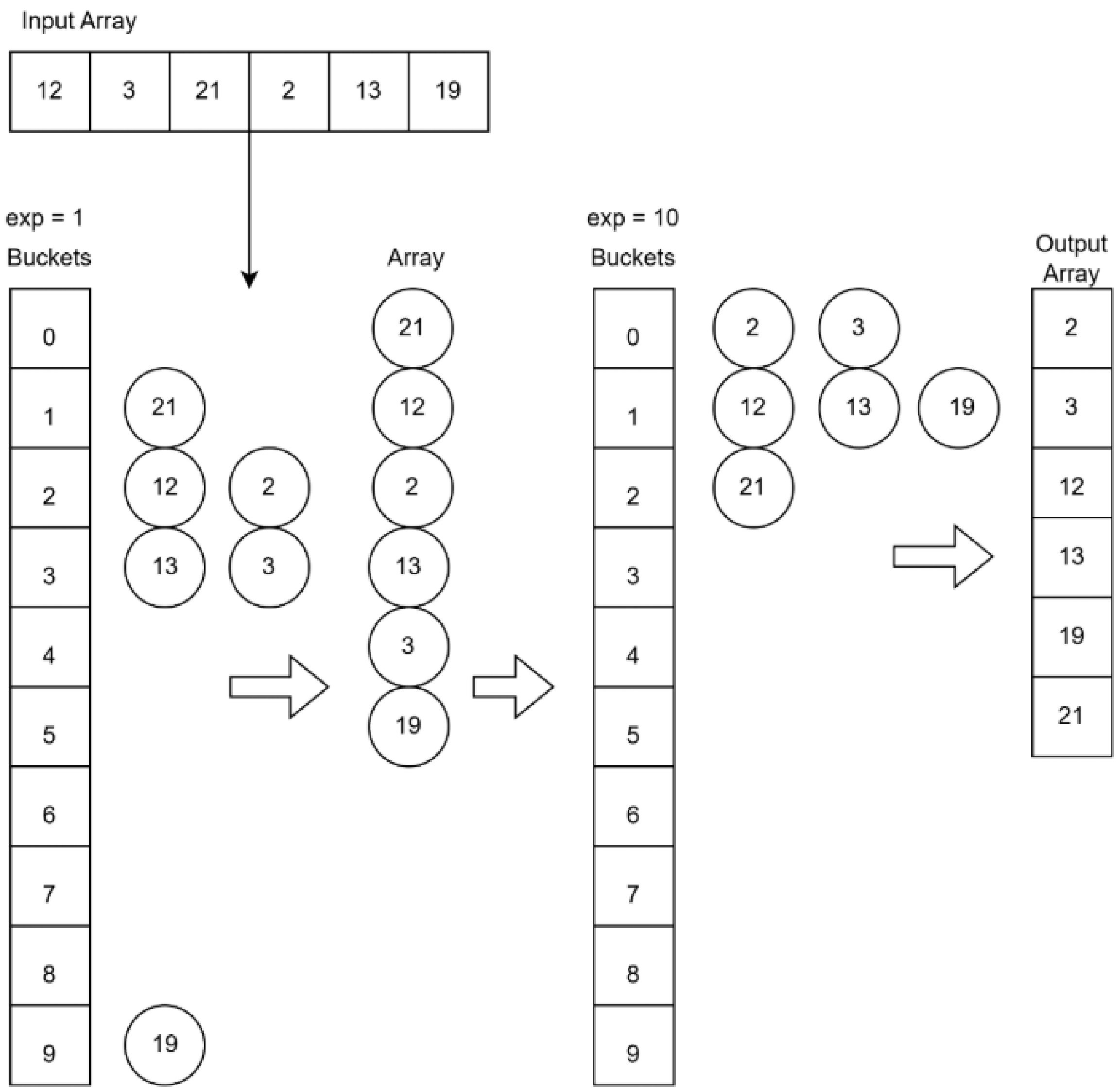


Figure 3.8.1: Radix Sort Visualization

3.8.5 Variations

There are many ways to implement Radix Sort. One of the most popular and modern ways is using Counting Sort as a sub sort with each exponent. By this idea, we will reduce the space that the algorithm requires.

3.8.6 Complexity Analysis

- **Time Complexity:**
  - Find maximum element in input array: $O(N)$
  - Pushback elements to buckets: $O(N)$
  - Copy the content of the buckets back into the input list: $O(b + N) = O(N)$ (there are a lot of types of base 2, 10, 16, ...)
    => Step 3 time complexity: $O(count \times (N + base))$
    => Radix Sort time complexity: $O(count \times (N + base))$, where count is digits that max value have, N is size of input list, base is base of the number system.
- **Space Complexity:**

  - 2D Array Buckets: $O(base * N)$, where N is the size of the input list, base is the base of the number system. Each bucket must have a capacity of up to N elements, as it is possible for all elements in the list to share the same digit at a given exponent.

3.9 Counting Sort

3.9.1 Idea

Counting sort is a non-comparison-based algorithm. The basic idea of this algorithm is to count the frequency of each distinct value that appears in the list and use that information to place elements in their correct positions. It works efficiently if the list contains non-negative integers and the range of input values are small compared to the size of the array.

3.9.2 Pseudo Code

Algorithm 10 Counting Sort Algorithm

```
function COUNTINGSORT(A, n)                                    ▷ Sort list a with n elements
    $max \leftarrow 0$
    Initialize array $ans$ of size n
    for $i \leftarrow 0$ to $n - 1$ do
        if $A[i] > max$ then
            $max \leftarrow A[i]$
        end if
    end for
    Initialize array $count$ of max + 1 zero
    for $i \leftarrow 0$ to $n - 1$ do
        $count[A[i]] \leftarrow count[A[i]] + 1$
    end for
    for $i \leftarrow 1$ to $size - 1$ do
        $count[i] \leftarrow count[i] + count[i - 1]$
    end for
    for $i \leftarrow n - 1$ to 0 do
        $ans[count[A[i]] - 1] = A[i]$
        $count[A[i]] \leftarrow count[A[i]] - 1$
    end for
end function
```

3.9.3 Step-by-step Description

1. Find the largest value in the list.
2. Initiate array Count with (max value + 1) element and set all elements to 0 to store the frequency of elements in the input array.
3. In array Count, increase 1 in count[value] when the value in the input list appears.
   Example: input arr[] = 1, 2, 4, 5;
   Arr[0] = 1 => count[arr[0]]++;
4. Calculate and store the sum of the current count and the previous count at the current index. In the end, the count array will store, at each index, the number of values less than or equal to that index.
5. Build an output array. For each element in the input array, place it at the index specified by the count array calculated in the previous step.

3.9.4 Visualization

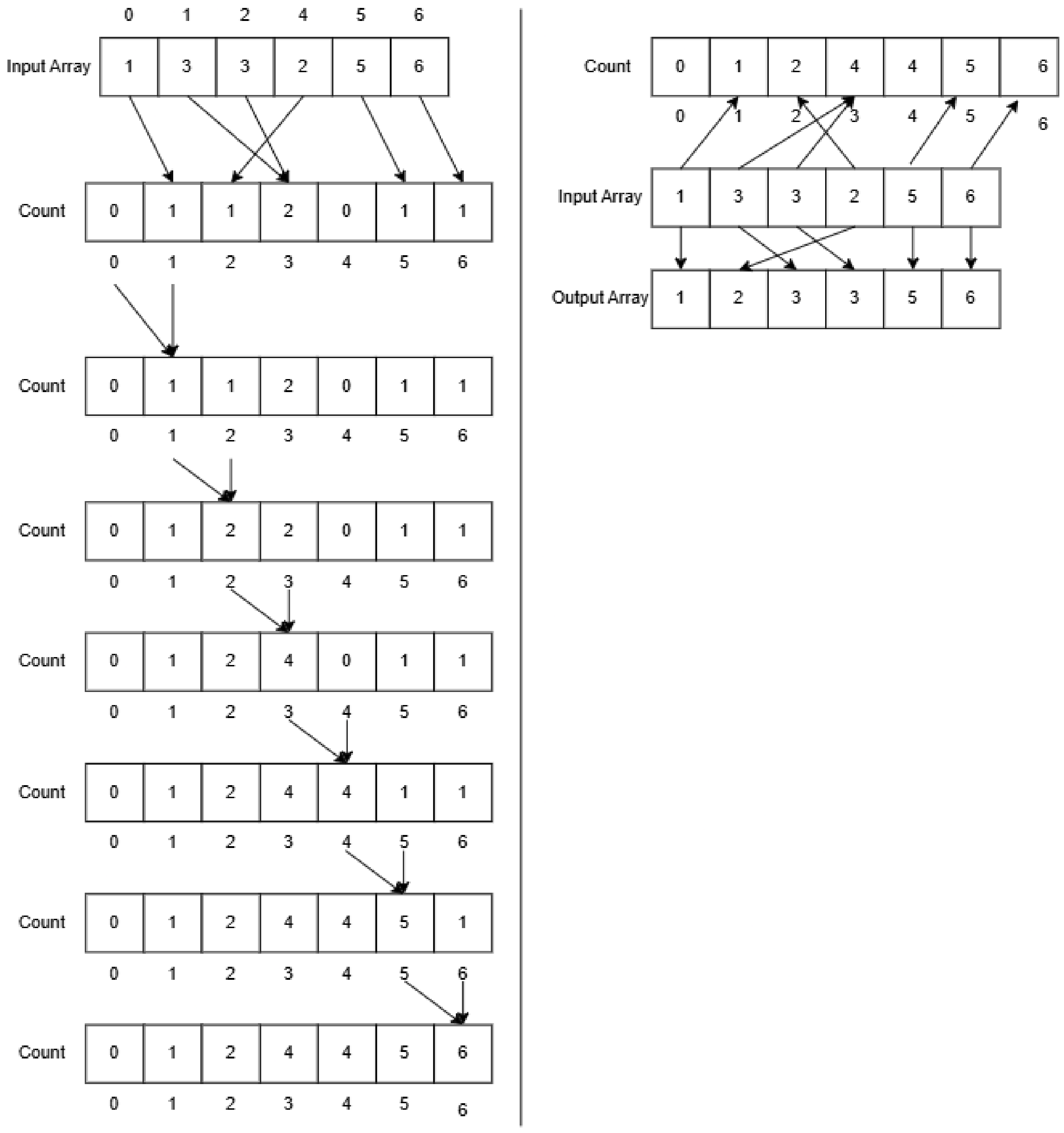


Figure 3.9.1: Counting Sort Visualization

3.9.5 Variations

There are some variations of counting sort that we can sort negative integer array.

3.9.6 Complexity Analysis

- **Time Complexity:**
  n: number of elements in the input array.
  k: largest value in the input array.
  1. Find max loop: $O(n)$.
  2. Count frequency loop: $O(n)$.
  3. Calculate the new position loop: $O(k)$.
  4. Build Answer array: $O(n)$.
  => Time Complexity is $O(n + k)$ in all case.
- **Space Complexity**
  1. Initiate Count Array: O(k + 1).

2. Output Array: O(n).
Space Complexity: O(n + k).

3.10 Binary Insertion Sort

3.10.1 Idea

Binary Insertion Sort algorithm is a variant of Insertion Sort. In standard Insertion Sort, linear search is used to find the correct position to insert each element, which can be improved. Since the left region of the element is already sorted during each iteration, we can apply **binary search** to efficiently find the insertion position in logarithmic time.
However, it is important to note that Binary Insertion Sort only reduces the number of comparisons made by utilizing binary search. The use of binary search efficiently improves the insertion process, but as we still need to **shift** the elements to make room for the inserted one, the number of swaps remains significant.[6]

3.10.2 Pseudo Code

Algorithm 11 Binary Insertion Sort

```
function BINARYSEARCH($A, key, low, high$)
    while $low \le high$ do
        $mid \leftarrow (low + high)/2$
        if $key < A[mid]$ then
            $high \leftarrow mid - 1$
        else
            $low \leftarrow mid + 1$
        end if
    end while
    return $low$
end function
function INSERTIONSORT($A, n$)
    for $i \leftarrow 1$ to $n - 1$ do
        $key \leftarrow A[i]$
        $insertPos \leftarrow binarySearch(A, key, 0, i - 1)$
        $j \leftarrow i - 1$
        while $j \ge insertPos$ do
            $A[j + 1] \leftarrow A[j]$
            $j \leftarrow j - 1$
        end while
        $A[insertPos] \leftarrow key$
    end for
end function
```

3.10.3 Step-by-step description

Initially, the entire array except the first element is the unsorted region.

1. **Key selection**
   Start from the second element A[1] and store it in the variable **key.**
2. **Find insertion position:**
   Use binary search on the sorted part of the array to find the correct position.
3. **Shifting Elements**
   Shift all elements from the insertion position to the right to make space for key
4. **Iteration:**
   Repeat steps 1-3 until the array is sorted.

3.10.4 Complexity Analysis

- **Time Complexity:**
  - **Best case:** $O(nlogn)$
  In Binary Insert Sort, when the array is initially sorted, we use binary search to find the right position for each element. Unlike regular Insertion Sort, which stops after comparing with the previous element, Binary Insert Sort continues to perform a binary search, which costs a logarithmic number of comparisons:

$$C(n) = log1 + log2 + log3 + \ldots + log(n-1) = log((n-1)!)$$

  which is $O(nlogn)$ (based on Stirling's approximation)[10] - **Average and Worst Case:** $O(n^2)$
  When the array is random or reversely sorted, Binary Insert Sort must consider both the cost of searching for the insertion point using binary search and the cost of shifting elements: - Binary Search for each iteration: $O(logn)$ - Shifting Elements: $O(n)$ After n − 1 iterations, the overall complexity will be $O(n^2 \times logn)$ Since $O(n^2)$ grows much faster than $O(nlogn)$ when n is large, the overall complexity simplifies to $O(n^2)$ Order of Growth: $O(n^2)$ **Additional Insight:** Binary Insertion Sort shares an inefficient quadratic time complexity with Insertion Sort, and can be worse in the best case. However, Binary Insertion Sort usually performs faster than Insertion Sort in practice, especially when the cost for comparison is high[6]

- **Space Complexity**
  Similar to Insertion Sort, Binary insertion sort requires only a constant amount of additional space. Hence, its space complexity depends on the method used for binary search.
  - Iterative Binary Search: $O(1)$
  - Recursive Binary Search: $O(logn)$ because of recursive calls

3.11 Shaker Sort

3.11.1 Idea

Shaker Sort is an extension of bubble sort. Shaker Sort sorts an array of elements by repeatedly swapping adjacent elements if they are in the wrong order. However, Shaker Sort also moves in the opposite direction after each pass through the array.

3.11.2 Pseudo Code

Algorithm 12 Shaker Sort Algorithm

```
function shakerSort(A, n)                              ▷ sort array A containing n elements
    $left \leftarrow 0$
    $right \leftarrow n - 1$
    $k \leftarrow n - 1$
    while $left < right$ do
        for $i \leftarrow left$ to $right - 1$ do
            if $A[i] > A[i-1]$ then
                $swap(a[i], a[i-1])$
                $k \leftarrow i$
            end if
        end for
        $right \leftarrow k$
        for $i \leftarrow right$ to $left + 1$ do
            if $A[i] < A[i-1]$ then
                $swap(a[i], a[i-1])$
                $k \leftarrow i$
            end if
        end for
        $left \leftarrow k$
    end while
end function
```

3.11.3 Step-by-step Description

- Step 1: Start from the beginning, swap each pair of adjacent elements if they are in the wrong order. At the end of the iteration, the largest number will reside at the end of the array.
- Step 2: Start from the item just before the most recently sorted item, and moving back to the start of the array. Here also, adjacent items are compared and are swapped if required. At the end of the iteration, the smallest number will reside at the beginning of the array
- Step 3: Iterate from the item after the most recently sorted item on the left of the array to the item before the most recently sorted item on the right of the array. Compare and swap if necessary.
- Step 4: Repeat step 3 but in opposite direction.
- Step 5: Repeat step 3 and step 4 until the array is sorted.

3.11.4 Visualization

Consider the input array is A={3, 1, 4, 5, 2}.

- Yellow cell indicate pair of elements that are being compared.
- Red cell indicate elements that are in the right positions.

3	1	4	5	2
1	3	4	5	2
1	3	4	5	2
1	3	4	5	2
1	3	4	2	5
1	3	4	2	5
1	3	2	4	5
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5

Table 6: Shaker Sort Visualization

3.11.5 Variations

Dual Cocktail Shaker Sort is a variant of Shaker Sort that performs a forward and backward pass per iteration simultaneously, improving performance compared to the original.

3.11.6 Complexity Analysis

- **Time Complexity:**
  Best case: $O(n)$
  This occurs when the original list is mostly ordered before applying the sorting algorithm.
  Average case: $O(n^2)$
  Worst case: $O(n^2)$

- **Space Complexity:** $O(1)$

3.12 Flash Sort

3.12.1 Idea

The Flash Sort algorithm was published by Karl-Dietrich Neubert in 1998. It's an in-place sorting algorithm that promises linear time complexity for evenly distributed data. It creates buckets (classes) and rearranges all elements according to buckets. Lastly, it sorts each bucket.
The algorithm works through the following steps:

1. Find minimum and maximum values.
2. Linearly divide the array into mm buckets.
3. Count the number of elements Lb for each bucket b.

4. Convert the counts of each bucket into the accumulative sum.
5. Rearrange all the elements of each bucket b in a position Ai where Lb-1 < i < Lb.
6. Sort each bucket using the **insertion sort.**

3.12.2 Pseudo Code

Algorithm 13 Flash Sort Algorithm Part 1

```
function GET_ID_BUCKET($value, min\_val, max\_val, m$)
    return $(m \times (value - min\_val))/(max\_val - min\_val + 1)$
end function
function INSERTIONSORT($array, start, end$)
    for $i \leftarrow start + 1$ to $end - 1$ do
        $temp \leftarrow array[i]$
        $j \leftarrow i - 1$
        while $j \geq start$ and $temp < array[i]$ do
            $array[j + 1] \leftarrow array[j]$
            $j \leftarrow j - 1$
        end while
        $array[j + 1] \leftarrow temp$
    end for
end function
function FLASHSORT($array, n$)
    if $n \leq 1$ then return
    end if
    $min\_val \leftarrow array[0]$
    $max\_val \leftarrow array[0]$
    for $i \leftarrow 0$ to $n - 1$ do
        if $array[i] < min\_val$ then
            $min\_val \leftarrow array[i]$
        end if
        if $array[i] > max\_val$ then
            $max\_val \leftarrow array[i]$
        end if
    end for
    if $min\_val = max\_val$ then
        return
    end if
    $m \leftarrow max(0.45 \times n, 1)$
```

Algorithm 14 Flash Sort Algorithm Part 2

```
    for $i \leftarrow 0$ to $n-1$ do
        $value \leftarrow array[i]$
        $b\_id \leftarrow get\_id\_bucket(value, minVal, maxVal, m)$
        $Lb[bId] \leftarrow Lb[bId] + 1$
    end for
    for $i \leftarrow 1$ to $m-1$ do
        $Lb[i] \leftarrow Lb[i] + Lb[i-1]$
    end for
    for $b \leftarrow 0$ to $m-2$ do
        if $b = 0$ then
            $start \leftarrow 0$
        else
            $start \leftarrow Lb[b-1]$
        end if
        $end \leftarrow Lb[b]$
        for $i \leftarrow start$ to $end - 1$ do
            $currentB \leftarrow get\_id\_bucket(array[i], min\_val, max\_val, m)$
        end for
        while $current\_b \neq b$ do
            if $currentB = 0$ then
                $swap\_start \leftarrow 0$
                $swap\_start \leftarrow Lb[currentB - 1]$
            end if
            $swap\_end \leftarrow Lb[currentB]$
            $swap\_index \leftarrow -1$
            for $j- > swapStart$ to $swapEnd - 1$ do
                if $get\_id\_bucket(array[j], min\_val, max\_val, m) \neq currentB$ then
                    $swapIndex \leftarrow j$
                    break
                end if
            end for
            if $swapIndex = -1$ then
                $swapIndex \leftarrow swapEnd - 1$
            end if
            swap(array[i], array[swap_index]
            $currentB \leftarrow getBucketId(array[i], min\_val, max\_val, m)$
        end while
    end for
    for $i \leftarrow 0$ to $m-1$ do
        if $i = 0$ then
            $start \leftarrow 0$
        else
            $start \leftarrow Lb[i-1]$
        end if
        $end \leftarrow Lb[i]$
        $insertionSort(array, start, end)$
    end for
end function
```

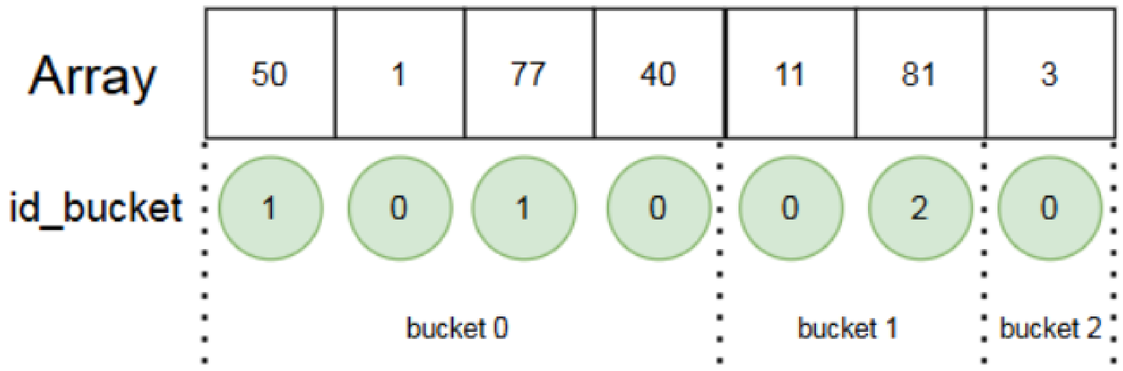
3.12.3 Step-by-step Description

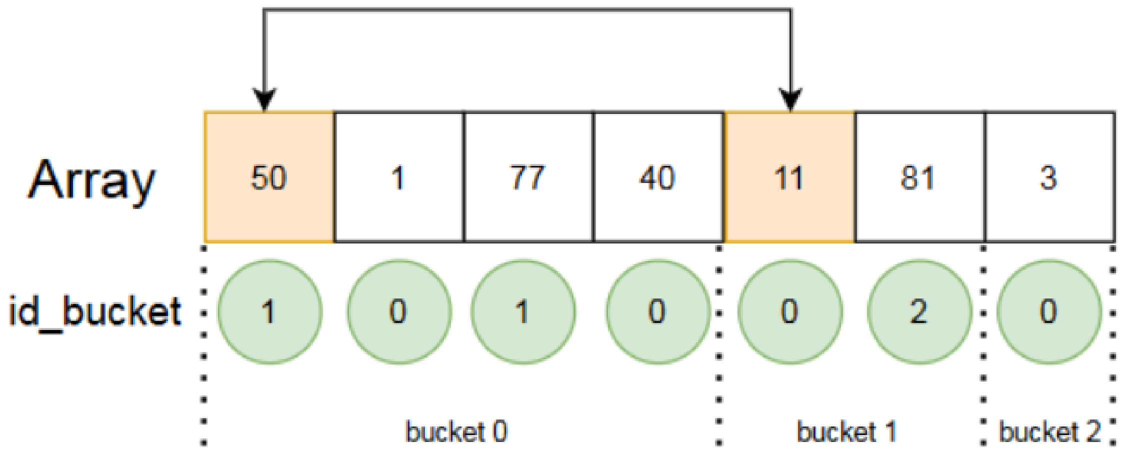
- Step 1: if the array has 1 or no element, we return it immediately because it is sorted already.
- Step 2: Find the min and max values in the array, to determine the range of the values for bucket distribution.
- Step 3: Calculate m – number of buckets, split the array into m buckets with$m \simeq 0.45 \times n$.
- Step 4: Count elements per bucket, using function **get_id_bucket** to distribute elements into buckets based on their values.

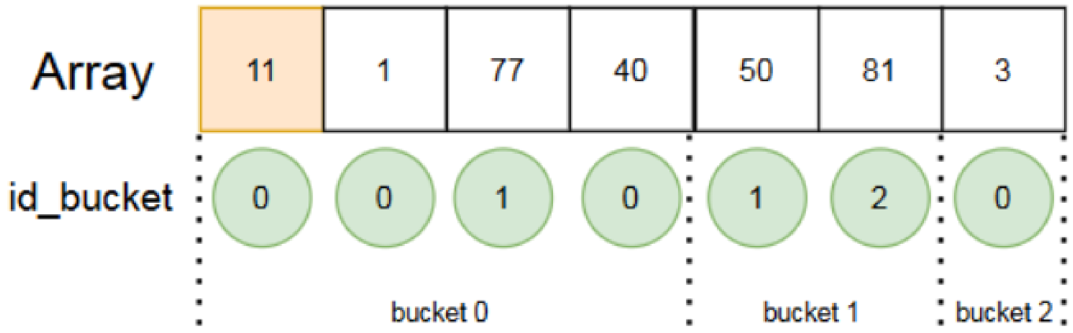
- Step 5: Convert Counts to Prefix Sums to define the end index of each bucket in the array. Step 6: Rearrange Elements into Buckets to ensure all elements are in their correct buckets through swaps.
- Step 7: Using function **insertionSort** for each bucket to finalize the sorted order within each bucket.

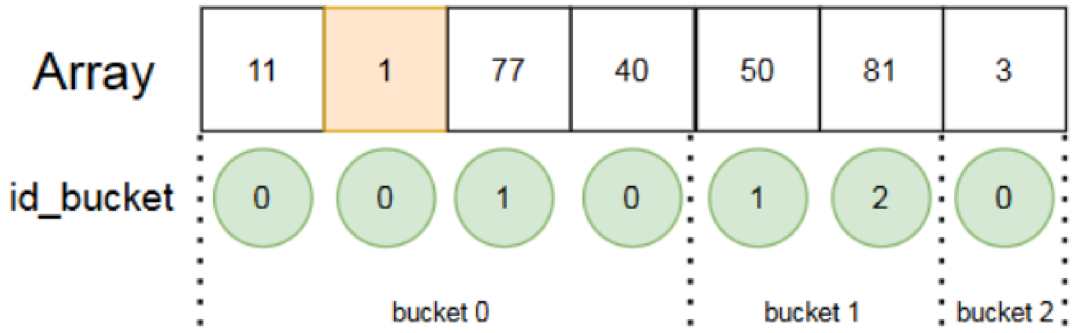
3.12.4 Visualization

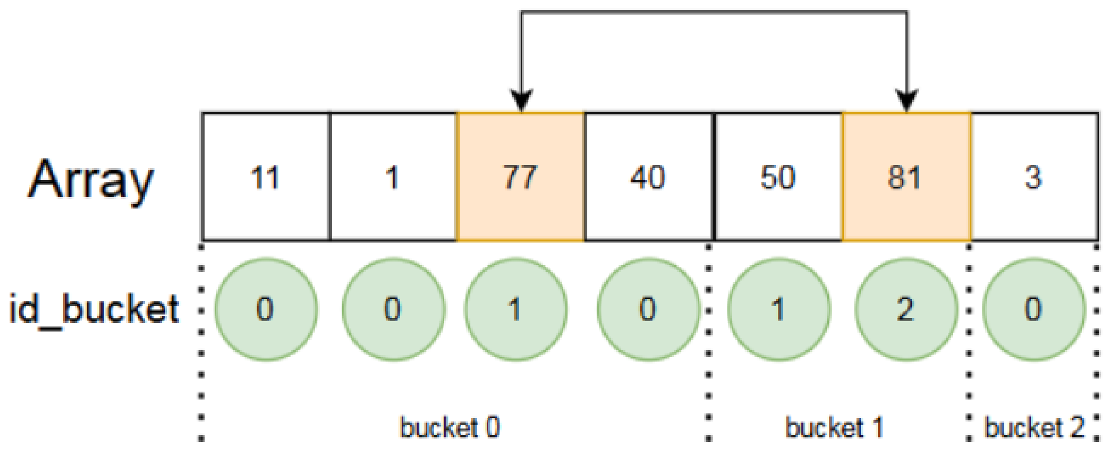
- **Input Array**: A = 50, 1, 77, 40, 11, 81, 3 . We have max_val = 81, min_val = 1; so the bucket size is 3.
- Re-arrange values of array into buckets:

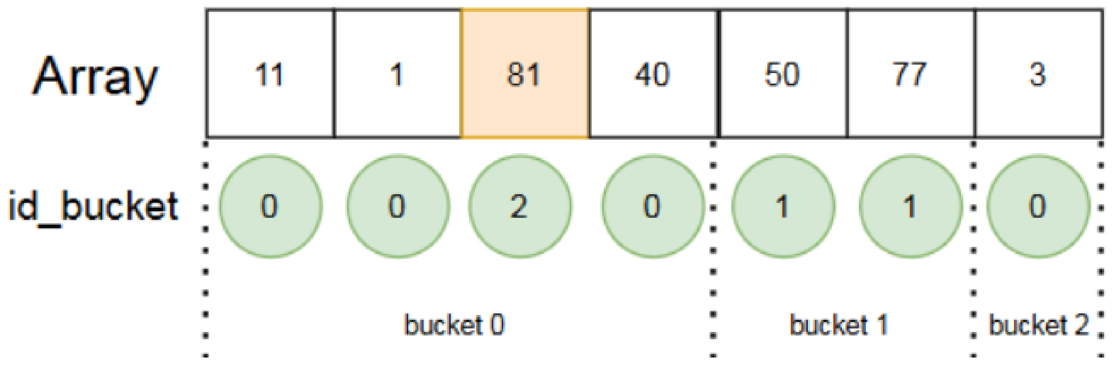

Array
50
1
77
40
11
81
3
id_bucket
1
0
1
0
0
2
0
bucket 0
bucket 1
bucket 2

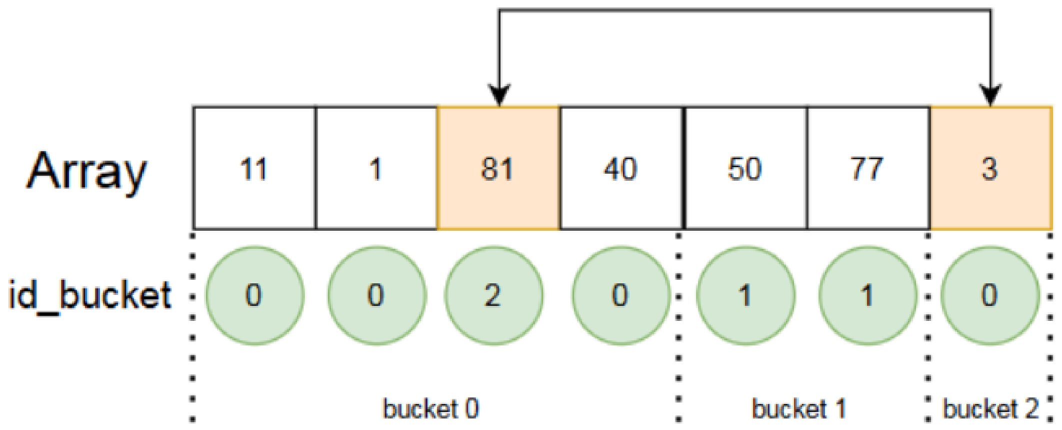

Array
50
1
77
40
11
81
3
id_bucket
1
0
1
0
0
2
0
bucket 0
bucket 1
bucket 2

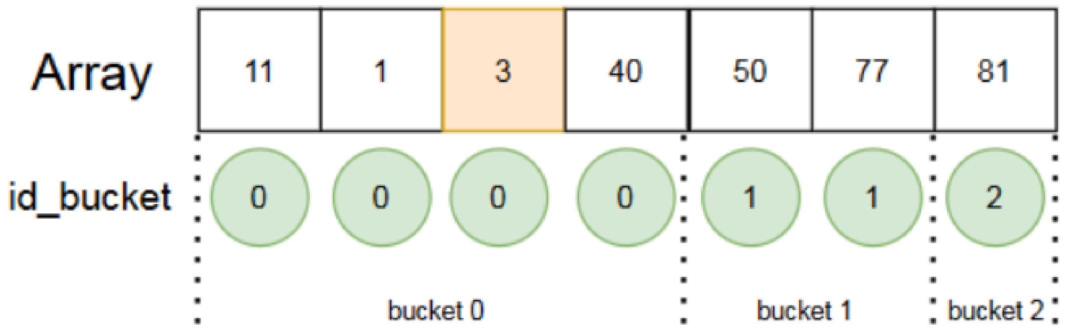

Array
11
1
77
40
50
81
3
id_bucket
0
0
1
0
1
2
0
bucket 0
bucket 1
bucket 2


Array
11
1
77
40
50
81
3
id_bucket
0
0
1
0
1
2
0
bucket 0
bucket 1
bucket 2


Array
11
1
77
40
50
81
3
id_bucket
0
0
1
0
1
2
0
bucket 0
bucket 1
bucket 2


Array
11
1
81
40
50
77
3
id_bucket
0
0
2
0
1
1
0
bucket 0
bucket 1
bucket 2


Array
11
1
81
40
50
77
3
id_bucket
0
0
2
0
1
1
0
bucket 0
bucket 1
bucket 2


Array
11
1
3
40
50
77
81
id_bucket
0
0
0
0
1
1
2
bucket 0
bucket 1
bucket 2

- **Sort bucket:**

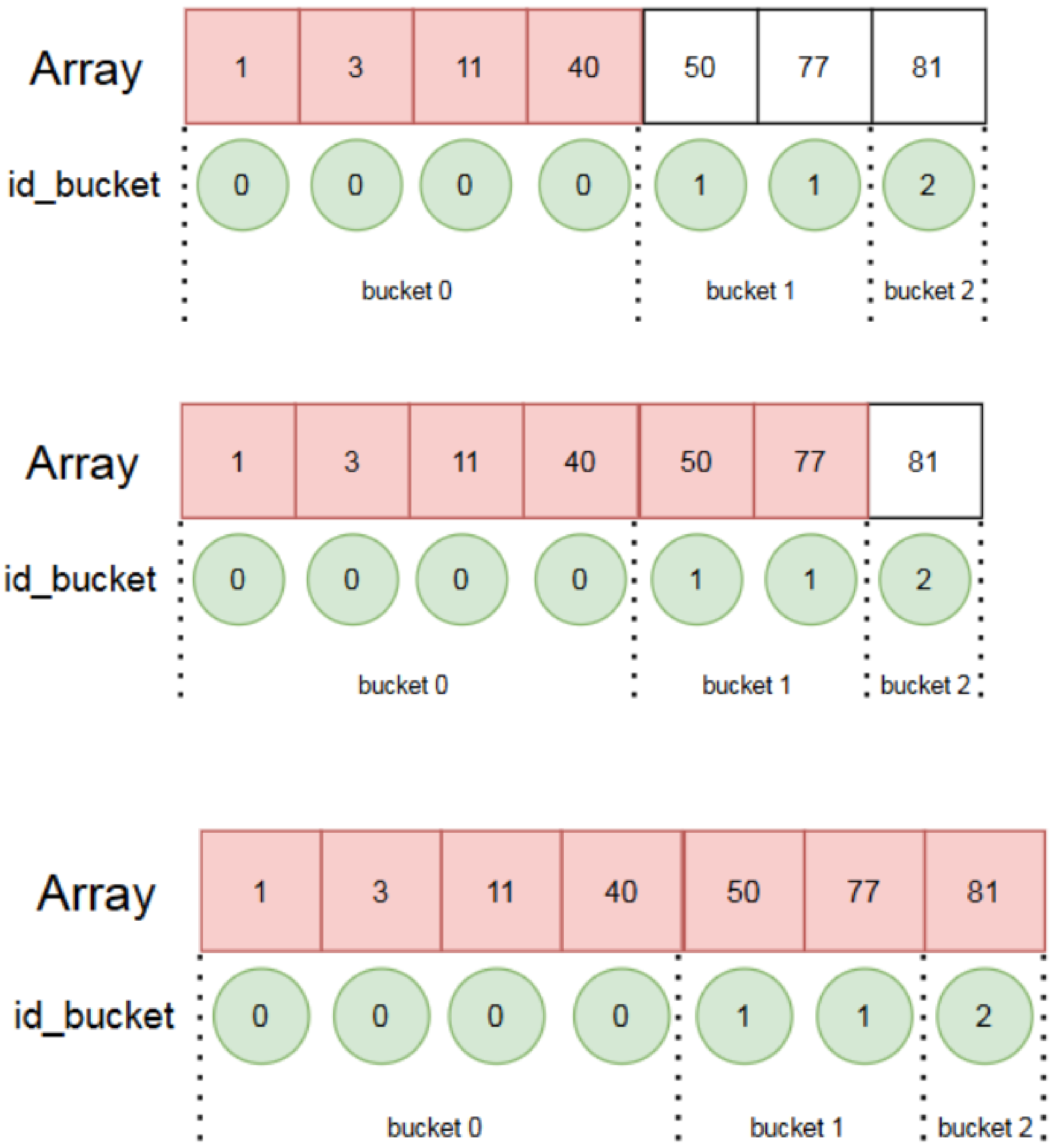


Figure 3.12.1: Flash Sort Visualization

3.12.5 Complexity Analysis

- **The worst case:** Occurs when the values in the array are very close to each other, resulting in fewer buckets compared to the number of elements. In this scenario, the insertion sort step dominates, leading to a time complexity of $O(n^2)$.
- **Space complexity:** $O(m)$, m is number of buckets. This is because the algorithm requires additional space to store the bucket counts.

4 Experiment Results and Comment

4.1 Randomized Data

4.1.1 Experiment Results and Graphs

Data order: Randomized						
Data size	10000		30000		50000	
Result statics	Run time	Comparisons	Run time	Comparisons	Run time	Comparisons
Selection Sort	194	100010001	1707	900030001	4763	2500050001
Insertion Sort	70	50189552	631	452087568	1699	1247833009
Shell Sort	1	639247	5	2263971	8	4361700
Bubble Sort	190	100010001	2024	900030001	5796	2500050001
Heap Sort	1	631258	5	2087360	9	3660575
Merge Sort	3	583586	9	1937257	14	3382680
Quick Sort	1	330434	3	1138355	5	1947030
Radix Sort	0	140056	2	510070	4	850070
Counting Sort	0	60002	0	180001	0	282770
Binary Insertion Sort	32	25444851	285	227236945	800	625975967
Shaker Sort	157	66900482	1525	602234745	4267	1663791431
Flash Sort	0	88833	1	285905	1	450677

Table 7: Experiment results for randomized data order (part 1)

Data order: Randomized						
Data size	100000		300000		500000	
Result statics	Run time	Comparisons	Run time	Comparisons	Run time	Comparisons
Selection Sort	21721	10000100001	192954	90000300001	644818	250000500001
Insertion Sort	8013	4995965849	62390	45005948111	194107	124837247401
Shell Sort	24	9919703	79	34095425	151	62847579
Bubble Sort	28932	10000100001	283678	90000300001	813127	250000500001
Heap Sort	21	7746679	97	25178860	165	43410959
Merge Sort	31	7166028	124	23383647	217	40381394
Quick Sort	12	4374344	48	15262029	96	30714993
Radix Sort	8	1700070	31	5100070	54	8500070
Counting Sort	1	532770	3	1532770	6	2532770
Binary Insertion Sort	3748	2502213613	41108	22516156612	111888	62441321068
Shaker Sort	18659	6664286688	205045	59977044549	441537	166401198527
Flash Sort	3	847497	16	2605110	21	4749661

Table 8: Experiment results for randomized data order (part 2)

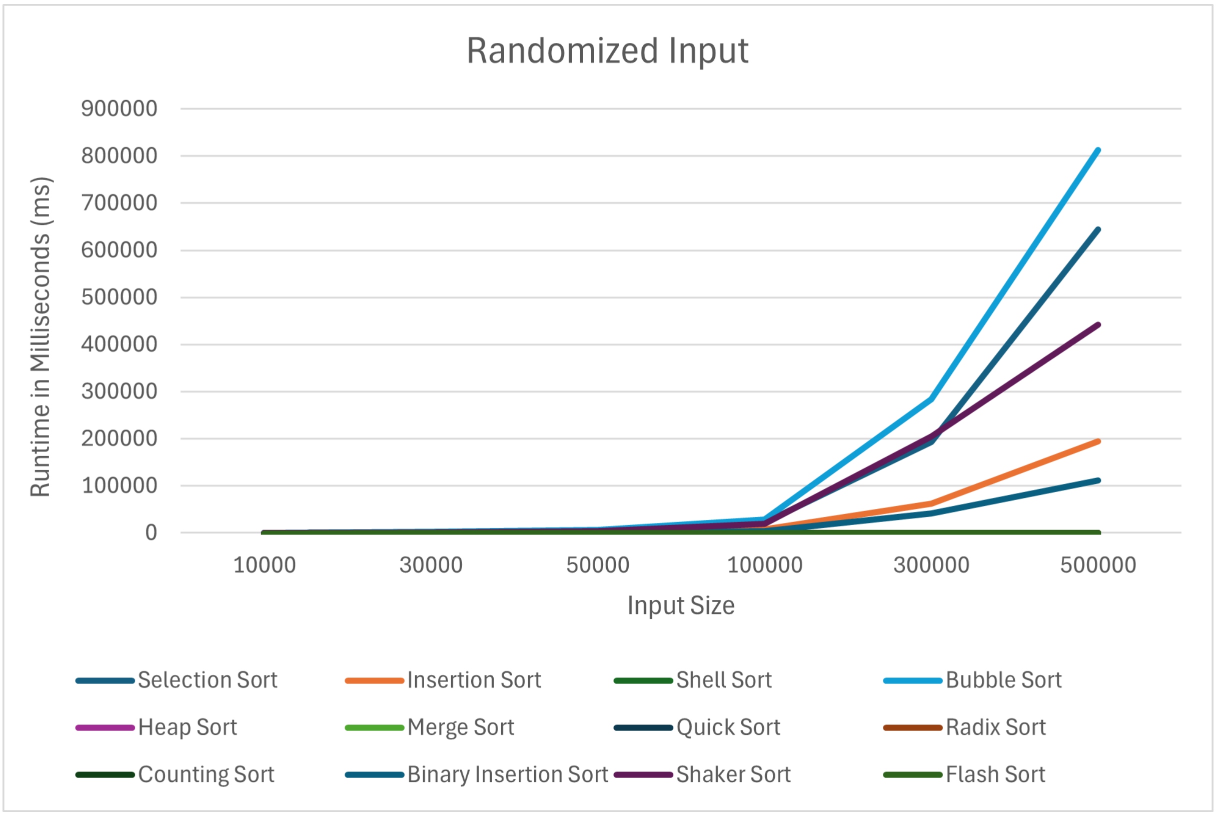


Figure 4.1.1: Line chart illustrating the running time of sorting algorithms with random input.

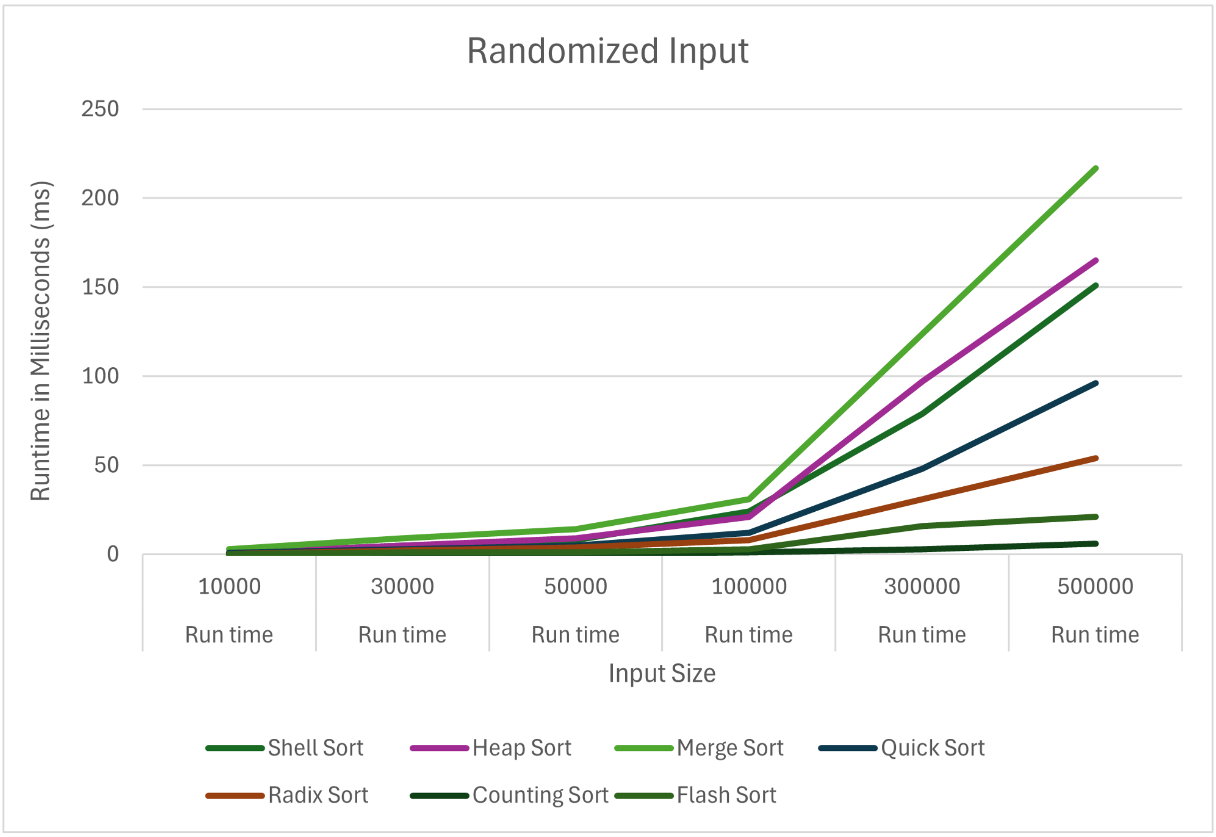


Figure 4.1.2: Line chart illustrating the runtime (ms) of sorting algorithms within a high-performance range (under 250 ms) for random input.

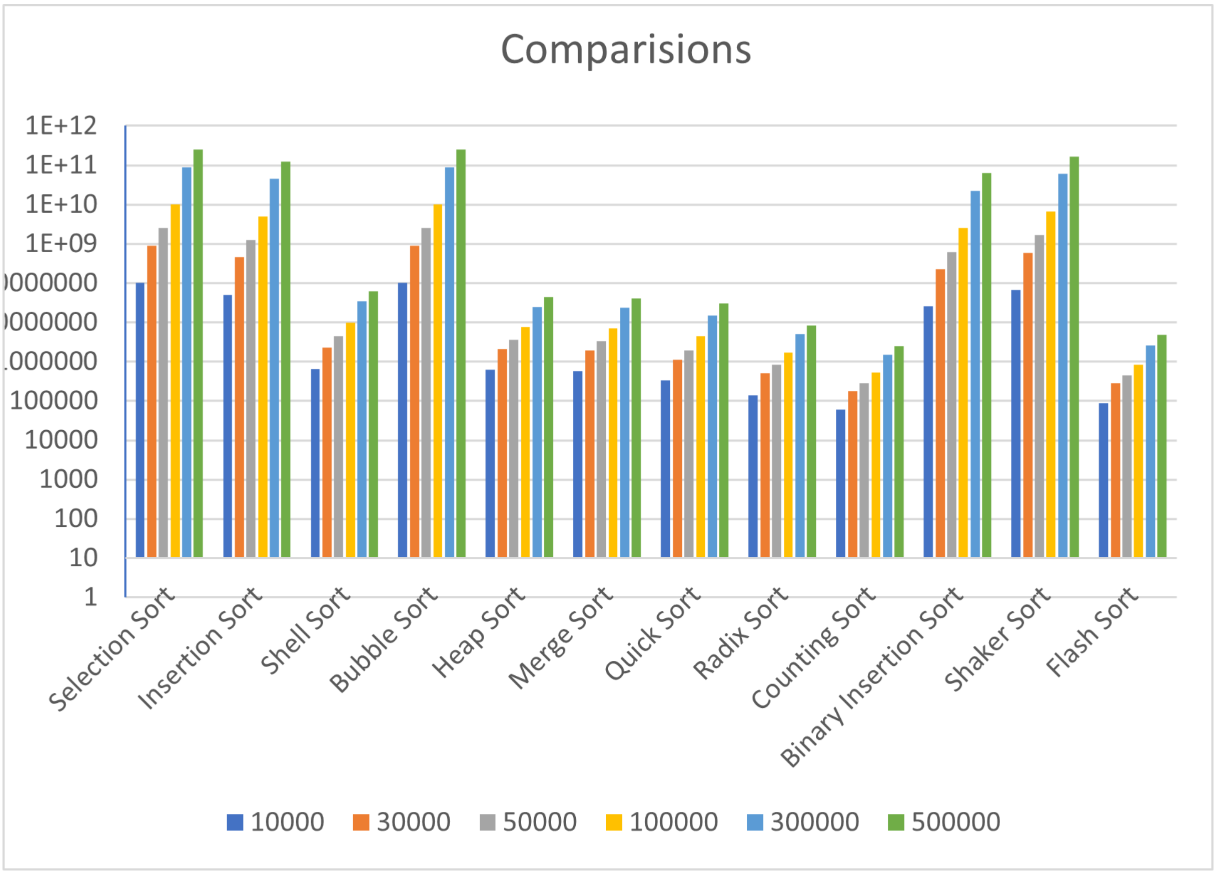


Figure 4.1.3: Bar chart illustrating the number of comparisons made by sorting algorithms with random input.

4.1.2 Comments

- According to Table 7, Table 8 and Figure 4.1.1, **Bubble Sort, Selection Sort**, and **Shaker Sort** exhibit the longest execution times among all tested input sizes.This upward trend becomes especially pronounced as the dataset size increases, clearly highlighting their inefficiency in handling large input.Their performance is greatly impacted by the excessive number of comparisons and swaps, which lead to their overall ineffectiveness.
- Notably, the graph clearly depicts that **Insertion Sort** and **Binary Insertion Sort**, despite sharing the same time complexity of $O(n^2)$ as the aforementioned algorithms, outperform them significantly due to their lower number of comparisons. **Binary Insertion Sort**, in particular, takes advantage of binary search to locate the insertion point, thereby reducing the comparison overhead. As a result, it achieves nearly **twice the speed of standard Insertion Sort** on larger datasets.
- In contrast, the remaining algorithms with time complexities less than **quadratic time complexities** $O(n^2)$ showcase exceptional efficiency. Most impressively, **non-comparative algorithms** like **Counting Sort, Radix Sort**, and **Flash Sort** achieve superior execution times, especially on large datasets. As Figure 4.1.2 illustrates, **Counting Sort** delivers the fastest performance overall, sorting **500,000 elements in under 10 milliseconds**
- Visually, algorithms with a time complexity of $O(n^2)$ and long runtimes also have a very large number of comparisons, for example, **Selection Sort, Insertion Sort, Shaker Sort and Bubble Sort**. Especially **Selection Sort** and **Shaker Sort**; for a sequence length of 500,000, both of these algorithms result in nearly 200 billion comparisons, which is 400,000 times the length of the sequence. Therefore, these algorithms are not suitable for problems with large sequence lengths.
- Next are the algorithms **Merge Sort, Heap Sort, Quick Sort**, and **Shell Sort**. Most of these have a time complexity of $O(nlogn)$, so the number of comparisons for these algorithms is greatly reduced compared to those with $O(n^2)$ complexity, falling in the range of 20 million to 60 million comparisons. Notably, **Shell Sort**, an improvement of **Insertion Sort**, reduces the number of comparisons from 124,837,247,401 (Insertion) to 62,847,579 (Shell) for the largest dataset.

- The two algorithms with the fewest comparisons are **Flash Sort** and **Counting Sort** (4,749,661 and 2,532,770, respectively, for the largest dataset).

4.2 Nearly Sorted Data

4.2.1 Experiment Results and Graphs

Data order: Nearly sorted						
Data size	10000		30000		50000	
Result statics	Run time	Comparisons	Run time	Comparisons	Run time	Comparisons
Selection Sort	86	100010001	765	900030001	2148	2500050001
Insertion Sort	0	152890	1	699882	0	439046
Shell Sort	0	407272	2	1356098	2	2252419
Bubble Sort	83	100010001	754	900030001	2059	2500050001
Heap Sort	1	697465	4	2291075	8	4032039
Merge Sort	2	510496	8	1641108	11	2812862
Quick Sort	18	22984222	84	107482343	445	582823228
Radix Sort	0	140056	2	510070	4	850070
Counting Sort	0	60002	0	180002	0	300002
Binary Insertion Sort	0	453286	2	1615223	3	2469351
Shaker Sort	0	158418	1	698366	0	514937
Flash Sort	0	113038	0	339742	1	548507

Table 9: Experiment results for nearly sorted data order (part 1)

Data order: Randomized						
Data size	100000		300000		500000	
Result statics	Run time	Comparisons	Run time	Comparisons	Run time	Comparisons
Selection Sort	8492	10000100001	76203	90000300001	213617	250000500001
Insertion Sort	0	696246	1	1333174	2	1938854
Shell Sort	4	4669833	15	15446703	26	25652926
Bubble Sort	8286	10000100001	74236	90000300001	207101	250000500001
Heap Sort	17	8543122	56	27752171	97	47647291
Merge Sort	23	5873799	68	18743072	115	32105621
Quick Sort	4976	6020466574	64882	77025549895	225451	227956368140
Radix Sort	8	1700070	28	6000084	63	10000084
Counting Sort	1	600002	3	1800002	9	3000002
Binary Insertion Sort	5	5176196	16	16617412	46	28621029
Shaker Sort	1	825945	1	1635313	4	2426965
Flash Sort	2	1066771	7	3137147	16	5202062

Table 10: Experiment results for nearly sorted data order (part 2)

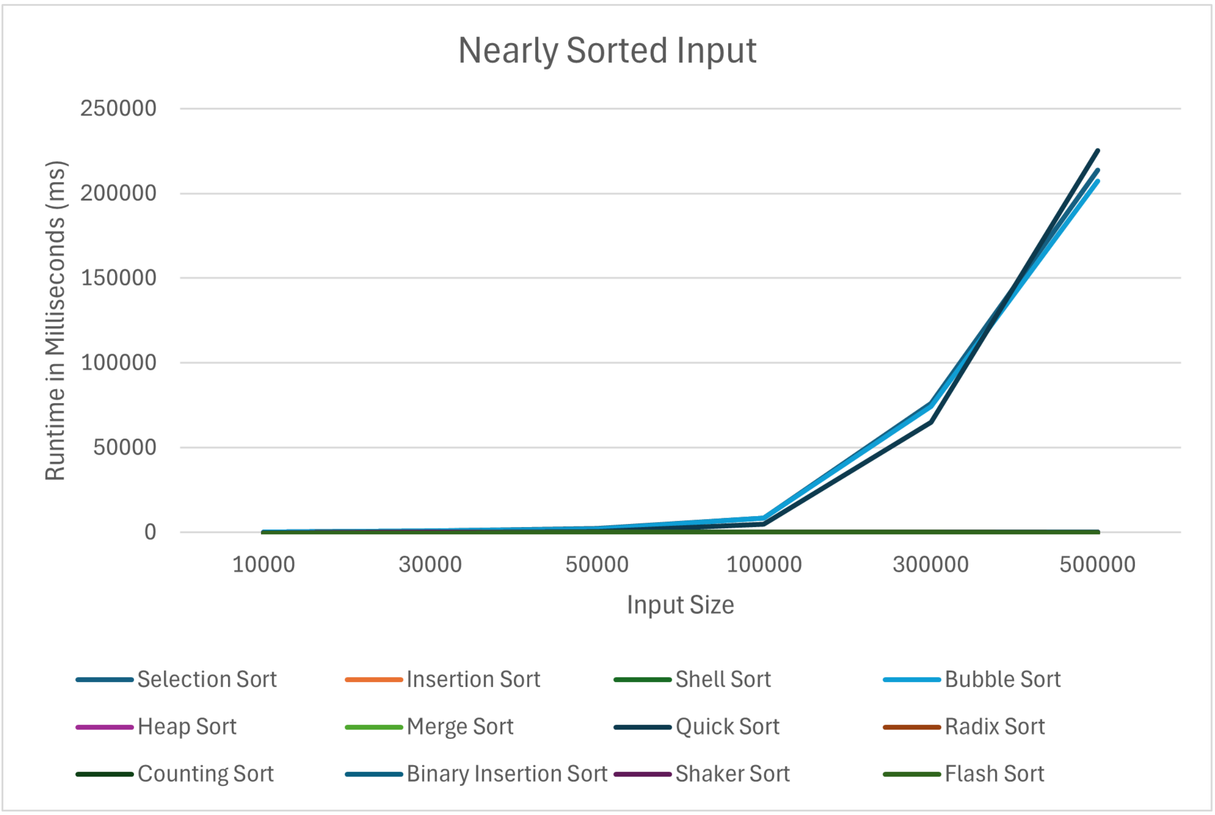


Figure 4.2.1: Line chart illustrating the running time of sorting algorithms with nearly sorted input.

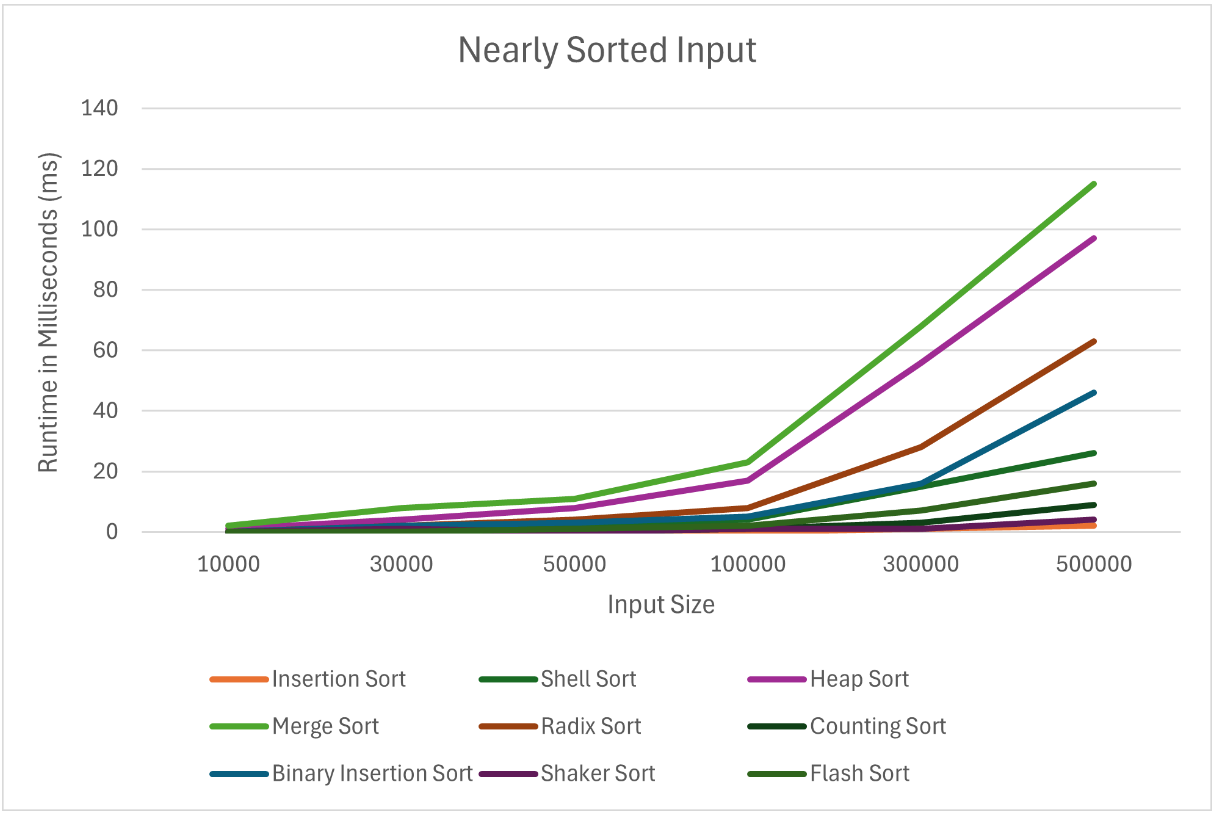


Figure 4.2.2: Line chart illustrating the runtime (ms) of sorting algorithms within a high-performance range (under 250 ms) for nearly sorted input.

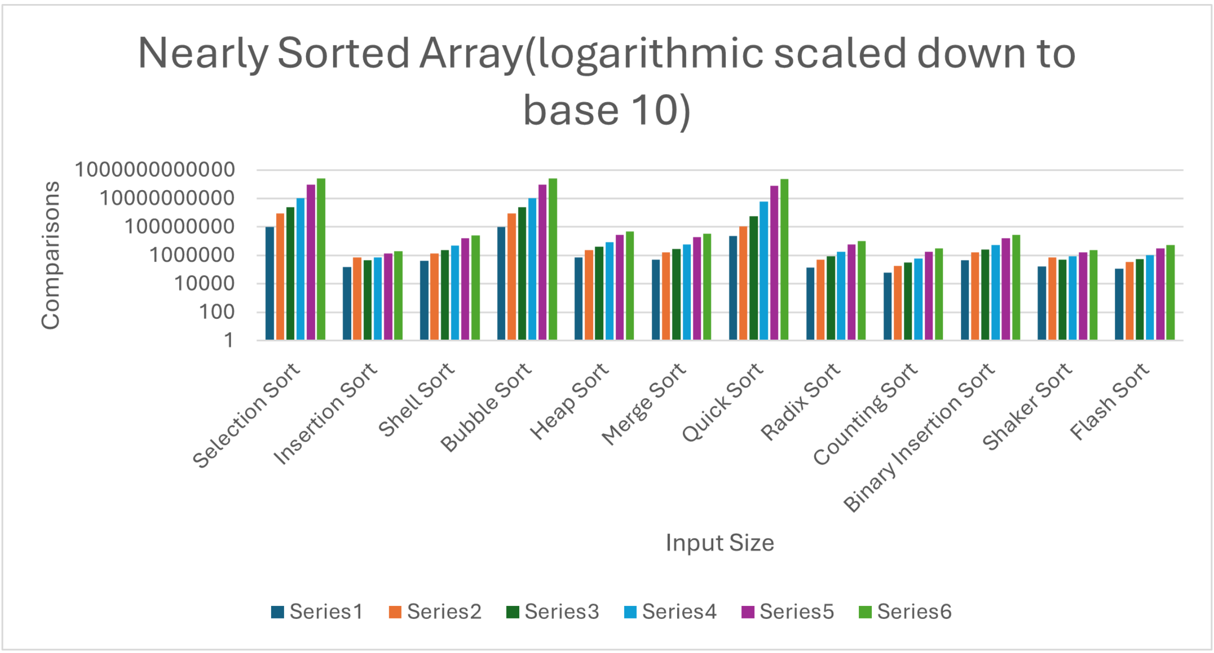


Figure 4.2.3: Bar chart illustrating the number of comparisons made by sorting algorithms with nearly sorted input.

4.2.2 Comments

- Referencing Figure 4.2.1, Table 9 and Table 10. **Bubble Sort** and **Selection Sort**, despite slight improvements over their performance on random or reversed inputs, remain far behind the rest.
- Most notably, **Insertion Sort** and **Shaker Sort** achieve excellent runtime efficiency, completing all tested input sizes, including 500,000 elements in just **4–5 milliseconds or less**. **Insertion Sort**, in particular, achieves close to $O(n)$ performance in this data order, as only a small number of element shifts are needed to complete the sorting, requiring only **2 milliseconds** to sort half a million elements. **Shaker Sort**, as a bidirectional variant of **Bubble Sort**, also benefits from early termination and minimal displacement of elements, allowing it to perform far better than its $O(n^2)$ average-case.
- **Binary Insertion Sort**, while slightly slower than its linear counterpart due to overhead from binary search operations, still performs remarkably well across all input sizes. Its runtime is comparable to that of **Shell Sort, Merge Sort**, and **Heap Sort**, all of which maintain consistently low execution times.
- While **Quick Sort**'s execution time, which does not benefit as much from the near-order of the data, increases substantially with input size, reaching approximately **225 thousand milliseconds** for 500,000 elements, **non-comparison-based algorithms** like **Counting Sort, Radix Sort**, and **Flash Sort** continue to dominate in performance with runtimes well under **70 milliseconds** for the largest input.
- According to Table 9, Table 10, it is evident that:
- **Bubble Sort** and **Selection Sort** have the most numerous comparisons, especially on a large array. Because of their simple complexity, they are less efficient than other algorithms.
- **Quick Sort** also has significant comparisons, because a nearly sorted array is one of the conditions that lead to the worst-case scenario of that algorithm. In this case, the complexity of **Quick Sort** is $O(n^2)$.
- Based on the chosen pivot, complexity of **Quick Sort** can be improved.
- In this condition, **Insertion Sort** and **Shaker Sort** perform fewer comparisons than other sorting algorithms. Both of them work very well with nearly sorted or already sorted arrays and require a minimal number of comparisons.
- The way that the gap sequence is chosen affects the complexity of **Shell Sort**. There are some efficient sequences proved by Computer Scientists. For example, Sedgewick's sequences, Knuth's

sequences, ...

- Other sorting algorithms seem to perform similarly. Depending on the specific purpose, we can choose the most appropriate one.

4.3 Sorted Data

4.3.1 Experiment Results and Graphs

Data order: Sorted						
Data size	10000		30000		50000	
Result statics	Run time	Comparisons	Run time	Comparisons	Run time	Comparisons
Selection Sort	85	100010001	771	900030001	2150	2500050001
Insertion Sort	0	29998	0	89998	0	149998
Shell Sort	0	360042	1	1170050	2	2100049
Bubble Sort	82	100010001	746	900030001	2072	2500050001
Heap Sort	1	697539	4	2289215	8	4032434
Merge Sort	2	475242	7	1559914	11	2722826
Quick Sort	77	100019998	692	900059998	1998	2500099998
Radix Sort	0	140056	2	510070	4	850070
Counting Sort	0	60002	0	180002	0	300002
Binary Insertion Sort	0	400849	1	1341697	2	2353393
Shaker Sort	0	20002	0	60002	0	100002
Flash Sort	0	103503	0	310503	1	517503

Table 11: Experiment results for sorted data order (part 1)

Data order: Sorted						
Data size	100000		300000		500000	
Result statics	Run time	Comparisons	Run time	Comparisons	Run time	Comparisons
Selection Sort	8484	10000100001	76899	90000300001	214321	250000500001
Insertion Sort	0	299998	1	899998	1	1499998
Shell Sort	4	4500051	15	15300061	26	25500058
Bubble Sort	8162	10000100001	74900	90000300001	208436	250000500001
Heap Sort	17	8544521	56	27752144	96	47647291
Merge Sort	22	5745658	92	18645946	116	32017850
Quick Sort	8227	10000199998	76381	90000599998	214817	250000999998
Radix Sort	8	1700070	28	6000084	46	10000084
Counting Sort	1	600002	7	1800002	5	3000002
Binary Insertion Sort	5	5006785	15	16427137	26	28427137
Shaker Sort	0	200002	0	600002	0	1000002
Flash Sort	2	1035003	12	3105003	12	5175003

Table 12: Experiment results for sorted data order (part 2)

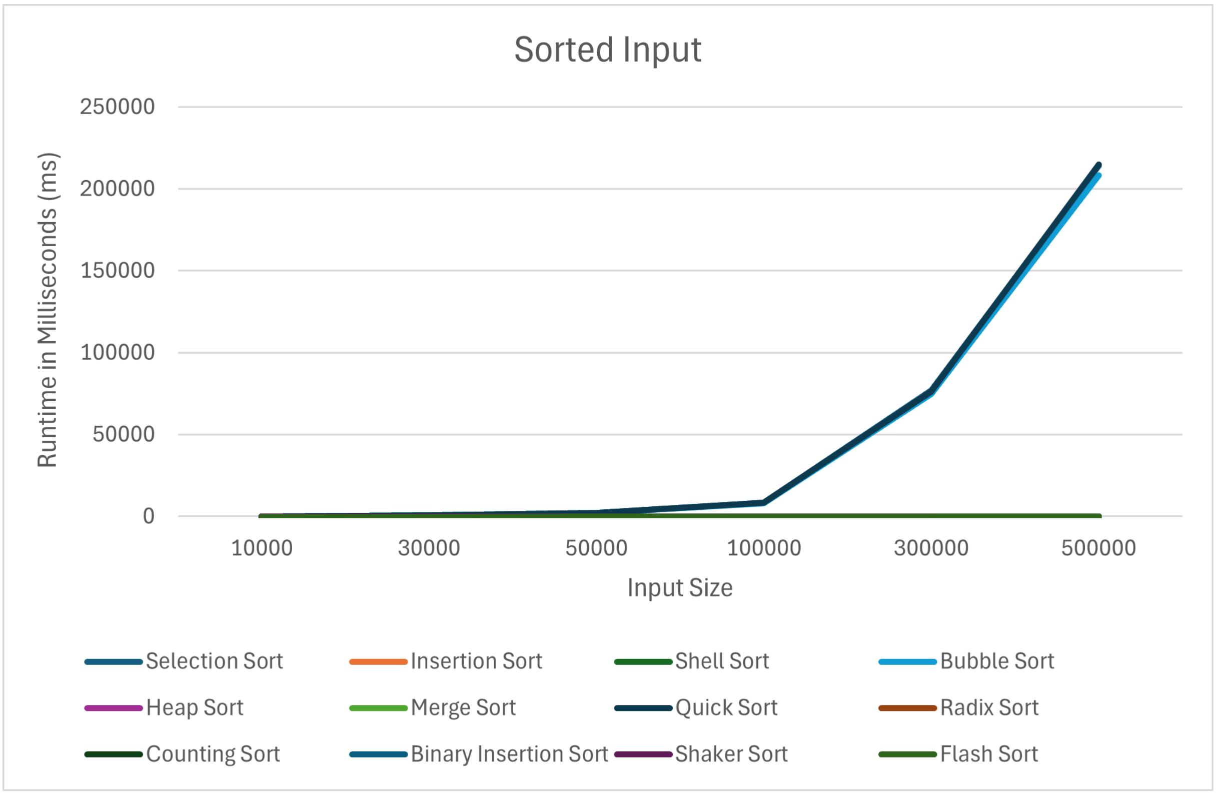


Figure 4.3.1: Line chart illustrating the running time of sorting algorithms with sorted input.

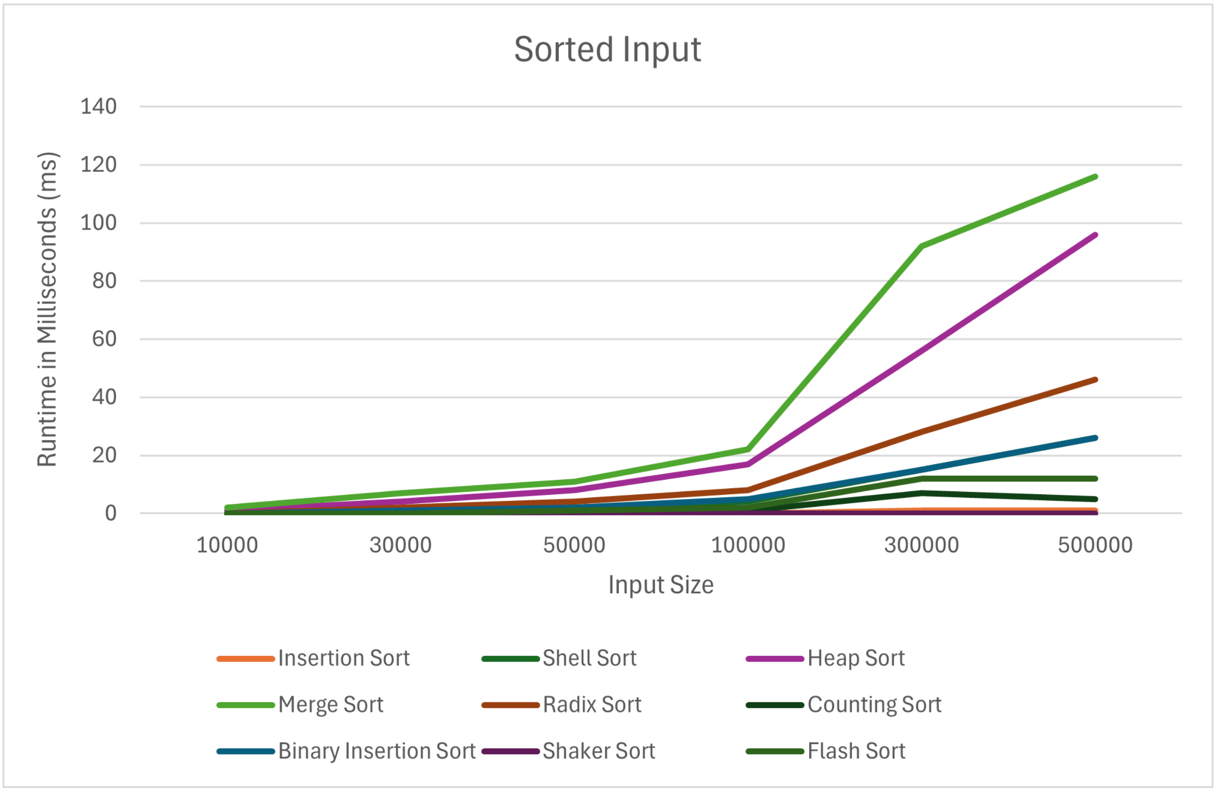


Figure 4.3.2: Line chart illustrating the runtime (ms) of sorting algorithms within a high-performance range (under 250 ms) for sorted input.

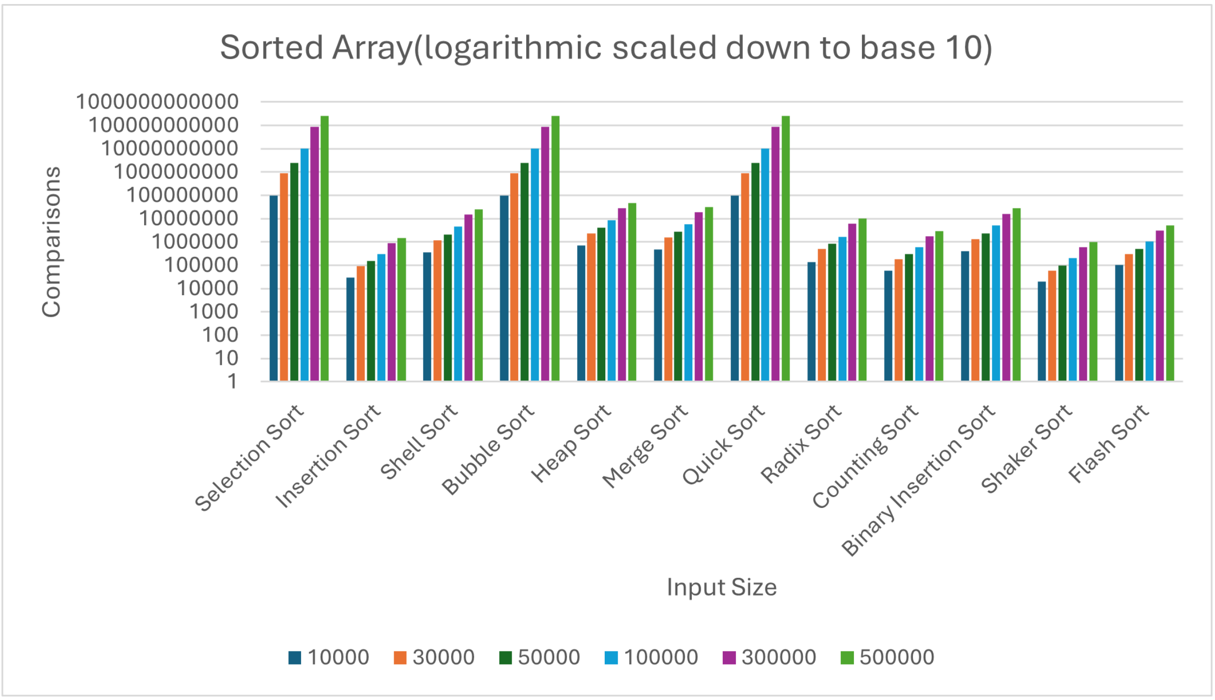


Figure 4.3.3: Bar chart illustrating the number of comparisons made by sorting algorithms with sorted input.

4.3.2 Comments

- Based on Table 11, Table 12 and Figure 4.3.1, it can be observed that **Selection Sort, Bubble Sort**, and somewhat surprisingly, **Quick Sort**, have the **worst performance** among the sorting algorithms tested on sorted input. **Selection Sort** performs a fixed number of comparisons regardless of order, and although **Bubble Sort** can terminate early, it often makes unnecessary passes. While their inefficiency is predicted due to their $O(n^2)$ complexity, **Quick Sort's** poor performance is more unexpected.
- Although **Quick Sort** belongs to the $O(nlogn)$ complexity class, it records the **slowest runtime on large, sorted datasets** in this case. This is likely due to **poor pivot choices**, which create unbalanced partitions and reduce performance to $O(n^2)$. It highlights a key weakness of Quick Sort—its sensitivity to input order and pivot selection.
- A closer inspection of algorithms that perform well on sorted input, Figure 4.3.2 reveals that **Merge Sort** and **Heap Sort** maintain **stable, moderate runtime** due to their consistent $O(nlogn)$ complexity in all cases.
- Most notably, **Counting Sort**, and especially **Insertion Sort** and **Shaker Sort**, demonstrate exceptionally fast execution times, each sorting the largest dataset (500,000 elements) in under 10 milliseconds. Both **Insertion Sort** and **Shaker Sort** benefit from partially or fully sorted input, with Insertion Sort achieving $O(n)$ performance and Shaker Sort terminating early, allowing them to outperform in best-case scenarios, despite their $O(n^2)$ worst-case complexity.
- According to Table 11, Table 12 , it is evident that:
- With an 500000-elements array, **Quick Sort** requires 250000999998 comparisons. It has the largest comparisons with the sorted input array because each time a pivot is chosen, it results in a very uneven split, one subset is empty, and the other contains n-1 elements. This leads to the worst-case scenario of this sorting algorithm.
- Additionally, **Selection Sort** and **Bubble Sort** have the same number of comparisons regardless of the initial order of the array. Both of them required 250000500001 comparisons when the array length reaches 500000 elements. Because of its numerous comparisons required, they are the two slowest sorting algorithms in this case.
- **Insertion Sort** and **Shaker Sort** demonstrate the best performance on this type of input, with running times close to 0 milliseconds. Both algorithms are considered optimal in this case because they require the fewest number of comparisons.

- **Shell** Sort is also a good option in this case. It performs excellently and does not require any additional memory beyond the input array.
- Although **Merge Sort, Counting Sort, Radix Sort, Flash Sort** are also good and fast, they need to use external memory based on their length of input array.
- **Heap Sort** and **Binary Insertion Sort** are not bad but there are other optimal methods that are more efficient.

4.4 Reverse Sorted Data

4.4.1 Experiment Results and Graphs

Data order: Reverse sorted						
Data size	10000		30000		50000	
Result statics	Run time	Comparisons	Run time	Comparisons	Run time	Comparisons
Selection Sort	85	100010001	763	900030001	2133	2500050001
Insertion Sort	105	100009999	964	900029999	2739	2500049999
Shell Sort	0	475175	2	1554051	3	2844628
Bubble Sort	175	100010001	1570	900030001	4401	2500050001
Heap Sort	1	643776	4	2131265	7	3752824
Merge Sort	1	476441	6	1573465	11	2733945
Quick Sort	130	100019998	1196	900059998	3365	2500099998
Radix Sort	0	140056	2	510070	3	850070
Counting Sort	0	60002	0	180002	0	300002
Binary Insertion Sort	61	50365891	558	451236742	1593	1252178441
Shaker Sort	186	100005001	1674	900015001	4707	2500025001
Flash Sort	0	97755	0	293255	1	488755

Table 13: Experiment results for reverse sorted data order (part 1)

Data order: Reverse sorted						
Data size	100000		300000		500000	
Result statics	Run time	Comparisons	Run time	Comparisons	Run time	Comparisons
Selection Sort	8727	10000100001	82244	90000300001	226196	250000500001
Insertion Sort	11281	10000099999	101835	90000299999	284754	250000499999
Shell Sort	7	6089190	23	20001852	39	33857581
Bubble Sort	18008	10000100001	160018	90000300001	1247377	250000500001
Heap Sort	16	7996145	54	26276122	93	45089346
Merge Sort	23	5767897	102	18708313	127	32336409
Quick Sort	13755	10000199998	123504	90000599998	346174	250000999998
Radix Sort	8	1700070	28	6000084	47	10000084
Counting Sort	1	600002	3	1800002	5	3000002
Binary Insertion Sort	6600	5004656836	62212	45015377194	172319	125026677194
Shaker Sort	18869	10000050001	168071	90000150001	467240	250000250001
Flash Sort	2	977505	6	2932505	11	4887505

Table 14: Experiment results for reverse sorted data order (part 2)

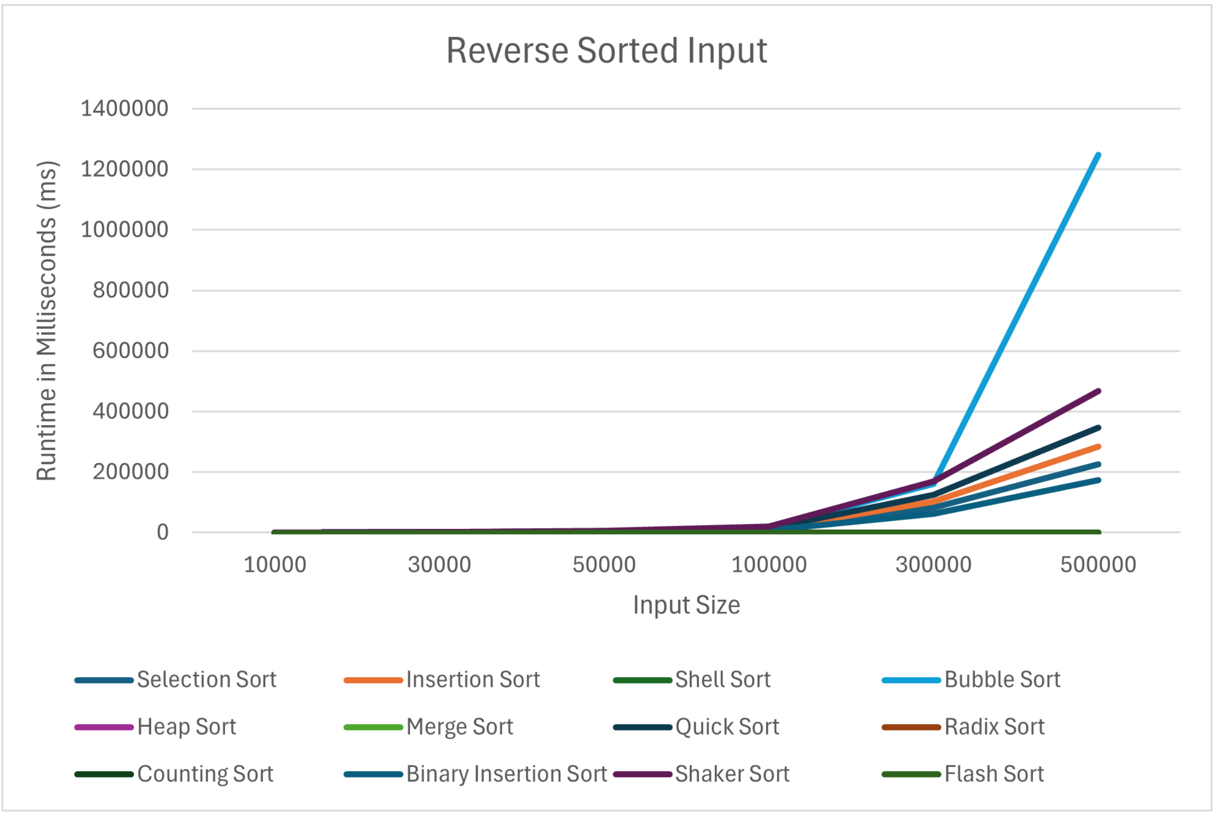


Figure 4.4.1: Line chart illustrating the running time of sorting algorithms with reverse-sorted input.

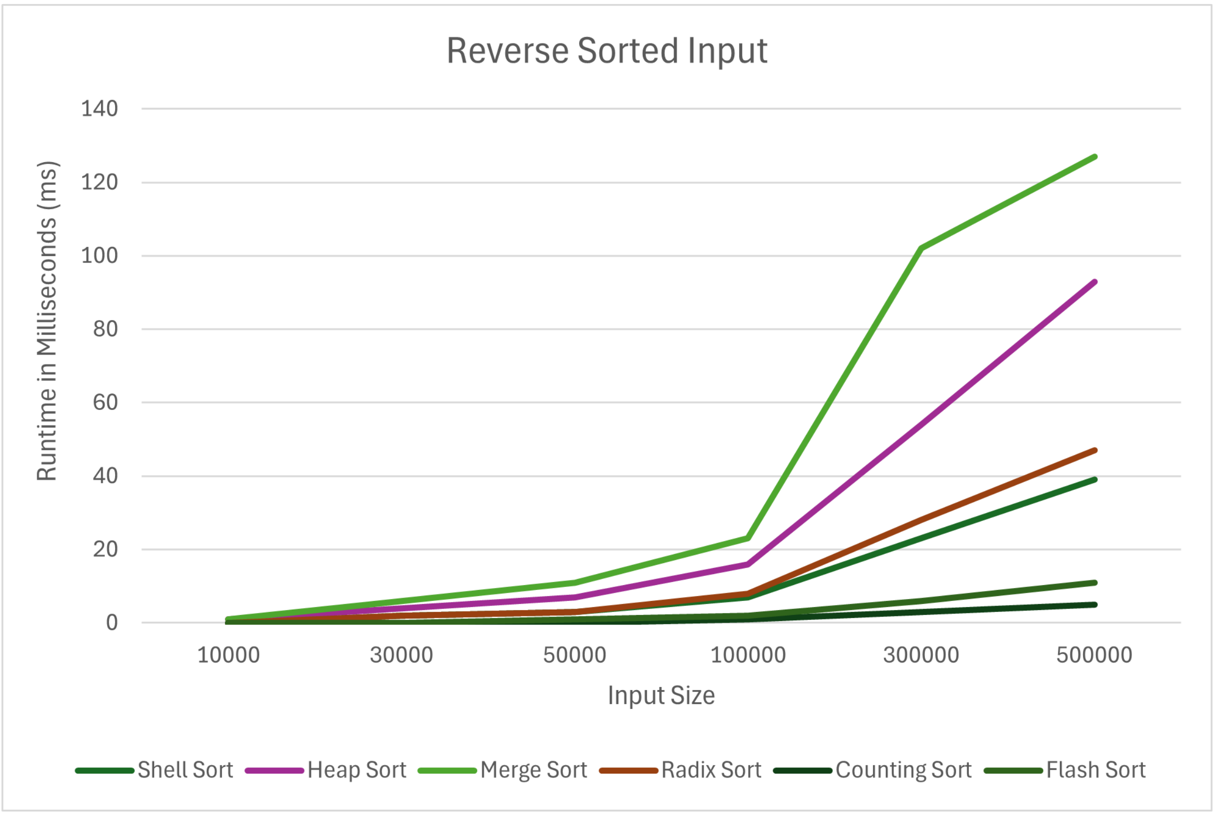


Figure 4.4.2: Line chart illustrating the runtime (ms) of sorting algorithms within a high-performance range (under 250 ms) for reverse-sorted input.

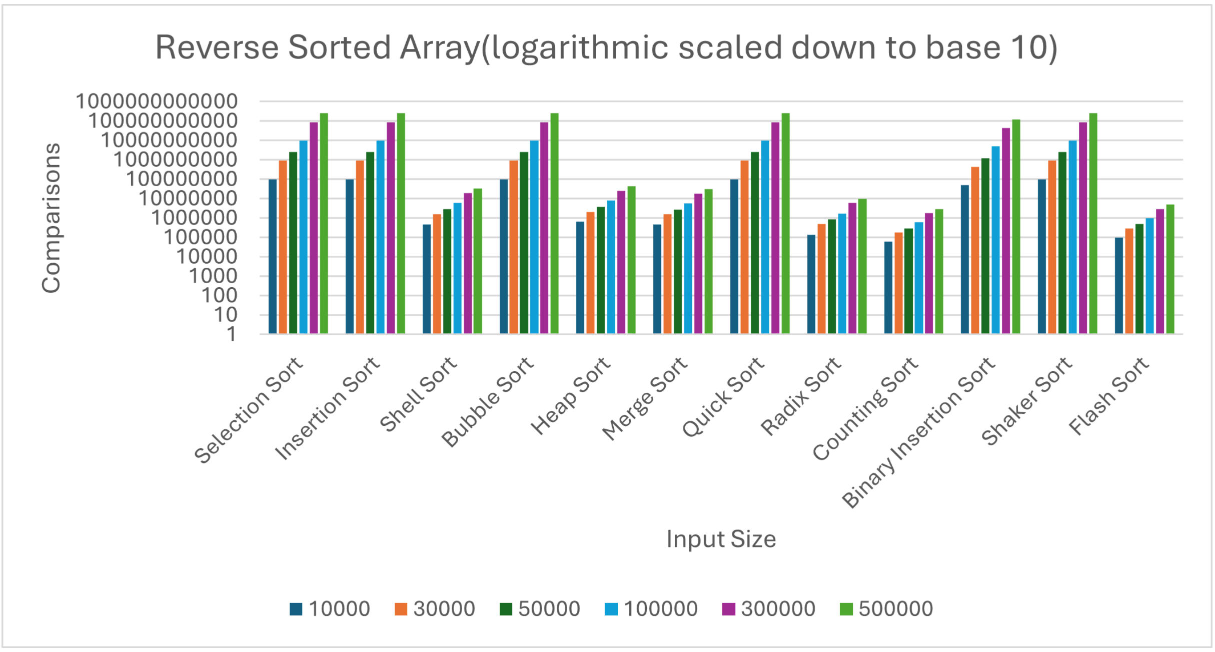


Figure 4.4.3: Bar chart illustrating the number of comparisons made by sorting algorithms with reverse-sorted input.

4.4.2 Comments

- From Figure 4.4.1, Table 13 and Table 14, it is obvious that all sorting algorithms having quadratic time complexity showcase poor performance when input size increases. **Bubble Sort**, in particular, stands out for its **extremely slow performance**, requiring nearly **1.25 million milliseconds** to sort the largest dataset—a reflection of its inefficiency when the number of **comparisons and swaps reaches its maximum**.
- **Quick Sort**, again shows **inefficient performance**, similar to the sorted input case. This reinforces its **sensitivity to pivot selection**—when a poor pivot is consistently chosen, as can happen in structured or reversed data, which potentially trigger **its worst-case behavior**.
- Figure 4.4.2 highlights the **stability** of **Merge Sort** and **Heap Sort**, which maintain **consistent performance** across all input types. **Shell Sort** and **Radix Sort** perform even better in this scenario by utilizing non-comparative strategies. At the top of the efficiency spectrum, **Counting Sort** and **Flash Sort** deliver the **fastest execution times** by a wide margin.
- According to Table 13, Table 14, it is evident that:
- **Selection Sort, Insertion Sort, Bubble Sort, Quick Sort**, and **Shaker Sort** involve a significant number of comparisons, which can lead to a worst-case complexity of $O(n^2)$. Due to this special type of input order, many sorting algorithms become less efficient.
- Based on the way the pivot is chosen, **Quick Sort** might work more efficiently. Its complexity depends on the quality of the pivot selection and the resulting balance between the partitions. The way the gap sequence is chosen affects the complexity of **Shell Sort**. There are some efficient sequences proven by computer scientists, such as Sedgewick's sequences, Knuth's sequences, etc.
- In contrast, **Counting Sort** is a sorting algorithm that have fewer comparisons than others. It is a non-comparison algorithms that works well with small and average datasets.
- **Flash Sort** is also an efficient method. It uses fewer comparisons than almost all other algorithms so that it has one of the best performance in this case.
- **Heap Sort** and **Merge Sort** both maintain a time complexity of $O(nlogn)$ regardless of the initial order of the input array, including reverse order. Therefore, the number of comparisons remains consistent for arrays of the same size. However, **Heap Sort** is a bit more challenging to implement. **Merge Sort** requires external memory beyond the input array.

4.5 Overall Comments

We classify sorting algorithms based on the **complexity** of their **average case**.

- $O(n^2)$: Selection Sort, Insertion Sort, Bubble Sort, Shaker Sort, Binary Insertion Sort.
- $O(nlogn)$: Merge Sort, Heap Sort, Quick Sort, Shell Sort.
- $O(n)$: Counting Sort, Radix Sort, Flash Sort.

And also by **stability** (records with the same key retain their relative order before and after the sort).

- **Stable:** Insertion Sort, Bubble Sort, Shaker Sort, Merge Sort, Counting Sort, Radix Sort, Binary Insertion Sort.
- **Unstable:** Selection Sort, Shell Sort, Heap Sort, Quick Sort, Flash Sort.

5 Project organization and Programming note

Here is some explanation of how I organized my source code, and some special libraries/ data structure used:

- **SORTING.h:** Header file containing:
  - Struct Result (comparisions, runtime)
  - Function protypes:
    - Sorting algorithms
    - Command-line solver functions
    - Array generation functions
- **Sorting.cpp**: This is C++ source code. It's highly probable that this file contains the implementations of various sorting algorithms (e.g., selection sort, insertion sort, quicksort, mergesort, etc.).
- **CMD.cpp:** This is a C++ source code. Based on the name, it likely contains functions related to handling commands, possibly command-line arguments that the progrm receives when it's run.
- **main.cpp**: This is a C++ source code file. By convention, this file usually contains the main() function, which is the entry point of your C++ program.
- **Readme.txt** contains some guide to help you execute the programme.
- **Input.txt, output.txt**: generated during program execution.
- **Struct Result:** Data structure for comparison count and algorithm runtime.
- Libraries **<time.h> , <chrono>** and function **ExecuteSortAndGetResult** are used to count comparisions and runtime of a sort function.

References

[1] Khan Academy. *Analysis of insertion sort.* URL: https://www.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/analysis-of-insertion-sort. Accessed April 11, 2025.

[2] Decidedlyso. *Merge-Insertion-Sort/README.md at Master · Decidedlyso/Merge-Insertion-Sort. GitHub.* URL: github.com/decidedlyso/merge-insertion-sort/blob/master/README.md. Accessed 15 Apr. 2025.

[3] Timothy M. Henry Frank M. Carrano. *Data abstraction problem solving with C++: Walls and mirrors.* Pearson, 2016.

[4] Mourad Hegazy. *Astering Merge Sort: An In-Depth Guide to Efficient Sorting.* URL: www.linkedin.com/pulse/mastering-merge-sort-in-depth-guide-efficient-sorting-mourad-hegazy-no0nf/. Accessed 15 Apr. 2025.

[5] Infopulse. *Timsort Sorting Algorithm.* URL: www.infopulse.com/blog/timsort-sorting-algorithm. Accessed 16 Apr. 2025.

[6] Syed Jafer K. *Binary Insertion Sort.* URL: https://parottasalna.com/2024/08/17/algorithm-binary-insertion-sort/. Accessed April 11, 2025.

[7] Kartik. *Selection Sort.* URL: https://www.geeksforgeeks.org/selection-sort-algorithm-2/. Accessed 25 Apr. 2025.

[8] Donald E. Knuth. *The Art of Computer Programming.* Addison-Wesley, 1998.

[9] Van Chi Nam. *DSA-03-Sorting Algorithms-Eng.*

[10] A Thornton. *ICS 46: Comparison-based sorting.* URL: https://ics.uci.edu/~thornton/ics46/Notes/ComparisonBasedSorting/. Accessed April 11, 2025.