VNUHCM UNIVERSITY OF SCIENCE

Faculty of Information Technology

DATA STRUCTURE AND ALGORITHMS

LAB 3 PROJECT: SORTING

23CLC03

1653012 - Phạm Viết Minh Đạo
23127157 - Nguyễn Hoàng Gia Bảo
23127396 - Lương Linh Khôi
23127494 - Nguyễn Huỳnh Tiến

Ho Chi Minh city, 7/2024

Table of contents

List of table & chart

1 Introduction

Sorting is the process of arranging data into a meaningful order to facilitate effective analysis. A sorting algorithm consists of instructions designed to sort specific data. This report will analyze 11 sorting algorithms: Selection Sort, Insertion Sort, Bubble Sort, Shaker Sort, Shell Sort, Heap Sort, Merge Sort, Quick Sort, Counting Sort, Radix Sort, and Flash Sort. The report includes an experiment to measure the runtime and comparison counts of each sorting algorithm. The experiment was conducted using a personal laptop operating on a 64-bit Windows system. Specific details about the laptop are as follows:

- Processor: AMD Ryzen 5 6600H with Radeon Graphics, 3.30 GHz
- Installed RAM: 16.0 GB (15.2 GB usable)
- System type: 64-bit operating system, x64-based processor

2 Algorithm Presentation

2.1 Selection Sort

Idea: Selection sort is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.

Step-by-step Description:

- Step 1: Set the increment variable $i = 1$
- Step 2: Find the smallest element in $a[i \dots n]$, and then swap it with $a[i]$. Increase i by 1 and go to Step 3
- Step 3: Check whether the end of the array is reached by comparing i with n. If $i < n$ then go to Step 2 (The first i elements are in place.) Otherwise, stop the algorithm

Here is pseudocode of this algorithm, which, for simplicity, assumes that the list is implemented as an array:

Algorithm

```
// Sorts a given array by selection sort
// Input: An array A[0...n - 1] of orderable elements
// Output: Array A[0...n - 1] sorted in nondecreasing order
for i ← 0 to n - 2 do
    min ← i
    for j ← i + 1 to n - 1 do
        if A[j] < A[min] then
            min ← j
    swap A[i] and A[min]
```

Complexity:

Time complexity:

The analysis of selection sort is straightforward. The input size is given by the number of elements n; the basic operation is the key comparison $A[j] < A[min]$. The number of times it is executed depends only on the array size and is given by the following sum:

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

Thus, selection sort is a $O(n^2)$ algorithm on all inputs. Note, however, that the number of key swaps is only $O(n)$, or, more precisely, $n - 1$ (one for each repetition of the i loop). This property distinguishes selection sort positively from many other sorting algorithms.

Space complexity:

Best case, Average case, and Worst case is $O(1)$.

Characteristics:

- In-place algorithm: Selection sort is an in-place algorithm because it doesn't require auxiliary memory to process the sorting.
- Comparison sorting algorithm: Selection sort is a comparison sort.
- Stable sorting algorithm: Selection sort is not stable because it can change the relative order of equal elements during sorting.

Applications:

- A small list is to be sorted
- Cost of swapping does not matter
- Checking of all the elements is compulsory
- Cost of writing to memory matters, like in flash memory (number of writes/swaps is $\Theta(n)$, as compared to $\Theta(n^2)$ of bubble sort)

2.2 Insertion Sort

Idea: Insertion sort is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list.

Step-by-step Description:

- Step 1: Set the increment variable $i = 2$.
- Step 2: Find the correct position pos in $a[1 \ldots i-1]$ to insert $a[i]$, i.e., where $a[pos-1] \leq a[i] \leq a[pos]$. Set $x = a[i]$, move forward $a[pos \ldots i-1]$ one element, set $a[pos] = x$. Increase i by 1 and go to Step 3.
- Step 3: Check whether the end of the array is reached by comparing i with n. If $i \leq n$ then go to Step 2. Otherwise, stop the algorithm.

Here is pseudocode of this algorithm:

Algorithm:

```
// Sorts a given array by insertion sort
// Input: An array A[0..n - 1] of n orderable elements
// Output: Array A[0..n - 1] sorted in nondecreasing order
for i ← 1 to n - 1 do
    v ← A[i]
    j ← i - 1
    while j >= 0 and A[j] > v do
        A[j + 1] ← A[j]
        j ← j - 1
    A[j + 1] ← v
```

Complexity:

Time complexity:

The basic operation of the algorithm is the key comparison $A[j] > v$ and the number of key comparisons in this algorithm obviously depends on the nature of the input.

Worst case:

In the worst case, $A[j] > v$ is executed the largest number of times, i.e., for every $j = i-1, \ldots, 0$. Since $v = A[i]$, it happens if and only if $A[j] > A[i]$ for $j = i-1, \ldots, 0$. (Note that we are using the fact that on the ith iteration of insertion sort all the elements preceding $A[i]$ are the first i elements in the input, albeit in the sorted order.) Thus, for the worst-case input, we get $A[0] > A[1]$ (for $i = 1$), $A[1] > A[2]$ (for $i = 2$), ..., $A[n-2] > A[n-1]$ (for $i = n-1$). In other words, the worst-case input is an array of strictly decreasing values. The number of key comparisons for such an input is:

$$C_{\text{worst}}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2).$$

Thus, in the worst case, insertion sort makes exactly the same number of comparisons as selection sort.

Best case:
In the best case, the comparison $A[j] > v$ is executed only once on every iteration of the outer loop. It happens if and only if $A[i-1] \leq A[i]$ for every $i = 1, \ldots, n-1$, i.e., if the input array is already sorted in nondecreasing order. Thus, for sorted arrays, the number of key comparisons is:

$$C_{\text{best}}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

Average-case:
For the no-sentinel version, the number of key comparisons to insert $A[i]$ before and after $A[0]$ will be the same. Therefore, the expected number of key comparisons on the ith iteration of the no-sentinel version is:

$$\frac{1}{i+1}\sum_{j=1}^{i} j + \frac{i}{i+1} = \frac{1}{i+1}\left(\frac{i(i+1)}{2}\right) + \frac{i}{i+1} = \frac{i}{2} + \frac{i}{i+1}$$

Hence, for the average number of key comparisons, $C_{\text{avg}}(n)$, we have:

$$C_{\text{avg}}(n) = \sum_{i=1}^{n-1}\left(\frac{i}{2} + \frac{i}{i+1}\right) - \frac{1}{2}\sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1}\frac{i}{i+1}$$

We have a closed-form formula for the first sum:

$$\frac{1}{2}\sum_{i=1}^{n-1} i = \frac{1}{2}\left(\frac{(n-1)n}{2}\right) = \frac{n^2 - n}{4}$$

The second sum can be estimated as follows:

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

Where $H_n = \sum_{j=1}^{n}\frac{1}{j} \approx \ln(n)$, we have:

$$C_{\text{avg}}(n) \approx \frac{n^2 - n}{4} + n - H_n \approx \frac{n^2}{4} \in \Theta(n^2)$$

Space complexity:
Best case, Average case, and Worst case is $O(1)$.

Characteristics:

- In-place algorithm: Selection sort is an in-place algorithm because it doesn't require auxiliary memory to process the sorting.
- Comparison sorting algorithm: Selection sort is a comparison sort.
- Stable sorting algorithm: Selection sort is a stable sorting algorithm.

Applications:

- The list is small or nearly sorted.
- Simplicity and stability are important.

Improvement (Binary Insertion Sort)
Idea: Use binary search to find the correct position for insertion.
Algorithm:

```
// Sorts a given array by insertion sort
// Input: An array x[0..n - 1] of n orderable elements
// Output: Array x[0..n - 1] sorted in nondecreasing order
for i ← 2, 3..., n do
    j ← BINARY-SEARCH(x[1:(i-1)], x[i])
    k ← i - 1
    while k > j do
        Swap x[k] and x[k-1]
        k ← k - 1

function BINARY-SEARCH (a[0..n - 1], b):
    low ← 1
    high ← m
    while low < high do
        mid ← (low + high) / 2
        if a[mid] = b then
            return mid
        else if a[mid] > b then
            low ← mid + 1
        else
            high ← mid - 1
    return low
```

Complexity:

- The number of swaps is the same as in the standard version of Insertion Sort.
- In the worst case, we perform approximately comparisons for $\log_2(i-1)$ each $i = 3, 4, 5, \ldots$, and do exactly one comparison for $i = 2$:

$$1 + \sum_{i=3}^{n} \log_2(i-1) = 1 + \log_2\left(\prod_{i=3}^{n} i\right) = 1 + \log_2\left(\frac{n!}{2}\right) = \log_2(n!)$$

- Using Stirling's approximation, we get:

$$\log_2(n!) = n\log_2 n - n\log_2 e + \log_2 n \in O(n \log n)$$

  So, we conclude that the number of comparisons Binary Insertion Sort performs is log-linear in n. However, since the number of swaps is $\Theta(n^2)$, both algorithms are asymptotically the same in the worst case. That's also true of their average-case.

2.3 Bubble Sort

Idea: Bubble Sort works by repeatedly swapping the adjacent elements if they are in the wrong order.

Step-by-step Description: Consider the array of n elements, $a[1 \ldots n]$.

- **Step 1:** Set the increment variable $i = 1$.

- **Step 2:** Swap any pair of adjacent elements in $a[1 \dots n - i + 1]$ if they are in the wrong order. Set the increment variable $j = 1$. If $a[j] > a[j + 1]$ then swap $a[j]$ with $a[j + 1]$. Increase j by 1 and repeat Step 2 until the end of the unsorted region. Increase i by 1 and go to Step 3.

- **Step 3:** Check whether the data is sorted by comparing i with n. If $i < n$ then go to Step 2 (The last i elements are in place). Otherwise, stop the algorithm.

Here is pseudocode of this algorithm:
Algorithm

```
// Sorts a given array by bubble sort
// Input: An array A[0...n - 1] of orderable elements
// Output: Array A[0...n - 1] sorted in nondecreasing order
for i ← 0 to n - 2 do
    for j ← 0 to n - 2 - i do
        if A[j + 1] < A[j]
            swap A[j] and A[j + 1]
```

Complexity:
Time complexity: The number of key comparisons for the bubble-sort version given above is the same for all arrays of size n; it is obtained by a sum that is almost identical to the sum for selection sort:

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

The number of key swaps, however, depends on the input. In the worst case of decreasing arrays, it is the same as the number of key comparisons:

$$S_{\text{worst}}(n) = C(n) = \frac{(n - 1)n}{2} \in \Theta(n^2)$$

Space complexity: Best case, Average case, and Worst case is $O(1)$.
Characteristics:

- In-place algorithm: Bubble sort is an in-place algorithm because it doesn't require auxiliary memory to process the sorting.

- Comparison sorting algorithm: Bubble sort is a comparison sort.

- Stable sorting algorithm: Bubble sort is a stable sorting algorithm.

Applications:

- Complexity does not matter.

- Short and simple code is preferred.

Improvement (with Boolean flag)
Idea: The process stops if no exchanges occur during any pass. A Boolean variable can signal when an exchange occurs in a pass.
Algorithm:

```
// The algorithm sorts array A[0...n-1] by improved bubble sort
// Input: An array A[0...n-1] of orderable elements
// Output: Array A[0...n-1] sorted in ascending order
count ← n - 1       // number of adjacent pairs to be compared
sflag ← true        // swap flag
while sflag do
    sflag ← false
    for j ← 0 to count - 1 do
        if A[j + 1] < A[j] then
            swap A[j] and A[j + 1]
            sflag ← true
    count ← count - 1
```

Complexity: In the best case, the complexity of bubble sort becomes $O(n)$.

2.4 Shaker Sort

Idea For each pass, after moving the smallest element to the beginning of the sequence, the algorithm will move the largest element to the end of the sequence.

Step-by-step description

1. Loops through the array from right to left, every pair of elements is compared.
2. In each pair, if the element on the left is greater than the right one, then elements are swapped. By the end of this pass, the smallest element of the list will be placed at the beginning of the array.
3. Loops through the array from left to right, starting from the element just before the most recently sorted element and moving towards the end of the array. Every pair of elements is compared.
4. In each pair, if the right element is smaller than the left element then, we swap those elements. By the end of this pass, the largest element of the list will be placed at the end of the array.
5. Repeat the process until all array elements are sorted.

Complexity

- **Time Complexity:**
  - Best Case: $O(n)$ - when the array is already sorted.
  - Average Case: $O(n^2)$ - when the array elements are in random order.
  - Worst Case: $O(n^2)$ - when the array is sorted in reverse order.
- **Space Complexity:** $O(1)$ - in all cases.

Characteristics

- Shaker sort is an in-place algorithm.
- Shaker sort is a comparison sort.
- Shaker sort is stable.

2.5 Heap Sort

Idea Construct a max heap from the given array and repeatedly move the largest element in the heap to the end of the array.

Step-by-step description

1. Build a max heap from the given input array.
2. Swap the root element of the heap (which is the largest element) with the last element of the heap.
3. Remove the last element of the heap (which is now in the correct position).
4. Heapify the remaining elements of the heap.
5. Repeat steps 2 to 4 until the heap contains only one element.

Complexity

- **Time Complexity:**
  - Best Case: $O(n \log n)$ - when the array is already sorted.
  - Average Case: $O(n \log n)$ - when the array elements are in random order.
  - Worst Case: $O(n \log n)$ - when the array is sorted in reverse order.
- **Time Complexity Analysis**

  The heap sort algorithm includes two stages:
  - **Heap Construction:** Constructing the heap takes $O(n)$ time.
  - **Sifting Down (Heapify):** There are $\lfloor \frac{n}{2} \rfloor$ sifts, each of which runs in $O(\log_2 n)$ time.
  - **Sorting:** Sorting the heap takes $O(n \log_2 n)$ time.
    - It executes $n - 1$ steps where each step runs in $O(\log_2 n)$ time.

  Therefore, heap sort has a time complexity of $O(n \log n)$ in all cases.
- **Space Complexity:** $O(1)$ - in all cases.

Characteristics

- Heap sort is an in-place algorithm.
- Heap sort is a comparison sort.
- Heap sort is not stable.

Variants

- **Ternary Heapsort:** Uses a ternary heap instead of a binary heap. It is more complicated to program but performs fewer swap and comparison operations.
- **Smoothsort:** A variant of heap sort that uses a different data structure called the Leonardo heap.

2.6 Shell Sort

Idea
Exchange items that are far apart h steps in the array.

Step-by-step description

1. Initialize the value of gap size, say h.
2. Divide the list into smaller sub-part. Each must have equal intervals to h.
3. Sort these sub-lists using insertion sort.
4. Repeat this step 2 until the list is sorted.

Complexity

- **Time Complexity:**
  - Best Case: $O(n \log n)$
  - Average Case: $O(n \log n)$
  - Worst Case: $\leq O(n^2)$
- **Space Complexity:** $O(1)$ - in all cases.

Characteristics

- Shell sort is an in-place algorithm.
- Shell sort is a comparison sort.
- Shell sort is not stable.

2.7 Merge Sort

Idea

Recursively divide the array into halves, sort each half, and then merge the sorted halves into one sorted array.

Step-by-step description

1. Divide the list or array recursively into two halves until it can no more be divided.
2. Each subarray is sorted individually using the merge sort algorithm.
3. The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.

Complexity

- **Time Complexity:**
  - Best Case: $O(n \log n)$ - when the array is already sorted.
  - Average Case: $O(n \log n)$ - when the array elements are in random order.
  - Worst Case: $O(n \log n)$ - when the array is sorted in reverse order.
- **Space Complexity:** $O(n)$ - in all cases.
- **Time Complexity Analysis**
  - At most $n-1$ comparisons to merge the two segments whose total number of elements is n.
  - n moves from the original array to the temporary array.
  - n moves for copying the data back.
  - Thus, each merge requires $3n - 1$ major operations.

Each call to mergeSort recursively calls itself twice.

- The levels of recursive calls:
  - $k = \log_2 n$ (if $n = 2^k$)
  - $k = 1 + \lfloor \log_2 n \rfloor$ (if $n \neq 2^k$)
- At level 0, the original call to mergeSort calls merge once: $3n - 1$ operations.
- At level 1, two calls to mergeSort, and hence two merges occur: $2 \times (3\left(\frac{n}{2} - 1\right)) = 3n - 2$ operations.
- At level m, 2^m calls to merge occur: $2^m \times 3\left(\frac{n}{2^m} - 1\right) = 3n - 2m$ operations.
- Each level of recursion requires $O(n)$ operations.
- There are either $k = \log_2 n$ (if $n = 2^k$) or $k = 1 + \lfloor \log_2 n \rfloor$ levels.

Therefore, merge sort has a time complexity of $O(n \log n)$ in all cases.

Characteristics

- Merge sort is an out-of-place algorithm.
- Merge sort is a comparison sort.
- Merge sort is stable.

Variants

- **Parallel Merge Sort:** Breaks the sorting operation into smaller sub-tasks and sorts them concurrently utilizing multiple processor cores.
- **Hybrid Merge Sort:** Combines Merge Sort with another sorting algorithm, such as Insertion Sort or Quick Sort, based on the size of the sub-arrays.
- **In-Place Merge Sort:** Reorders components inside of the original array without utilizing additional memory.

2.8 Quick Sort

Idea:
QuickSort is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

Step-by-step Description

1. Consider an initial array, `a[first..last]`.
2. Pick the pivot $p = a[k]$, where $k = \frac{\text{first}+\text{last}}{2}$.
3. Identify pairs of elements that are not in their correct positions and swap them.
   (a) Set the increment variables, $i = \text{first}$ and $j = \text{last}$.
   (b) While $a[i] < p$ do increase i by 1. While $a[j] > p$ do decrease j by 1.
   (c) If $i \leq j$ then swap $a[i]$ with $a[j]$, increase i by 1 and decrease j by 1.
   (d) Go to Step 3.
4. Check whether the two smaller subarrays overlap.
   (a) If $i < j$ then go to Step 2.
   (b) Otherwise, recursively go to Step 1 with `a[first..j]` and `a[i..last]`.

Complexity Evaluations

Variation	Time Complexity	Space Complexity
Best case	$O(n \log n)$	$O(\log n)$
Worst case	$O(n^2)$	$O(n)$
Average case	$O(n \log n)$	$O(\log n)$

Time Complexity:

- **The best case** occurs when we select the pivot as the mean element.

  Explain:

$$T(N) = 2 \cdot T\left(\frac{N}{2}\right) + N \cdot \text{constant}$$

  Now

$$T\left(\frac{N}{2}\right) = 2 \cdot T\left(\frac{N}{4}\right) + \frac{N}{2} \cdot \text{constant}$$

  So,

$$T(N) = 2\left(2 \cdot T\left(\frac{N}{4}\right) + \frac{N}{2} \cdot \text{constant}\right) + N \cdot \text{constant}$$

$$= 4 \cdot T\left(\frac{N}{4}\right) + 2 \cdot \text{constant} \cdot N$$

$$T(N) = 2^k \cdot T\left(\frac{N}{2^k}\right) + k \cdot \text{constant} \cdot N$$

then, $2^k = N$

$$k = \log_2 N$$

So, we can say that

$$T(N) = N \cdot T(1) + N \cdot \log_2 N$$

Therefore, the time complexity is $O(N \cdot \log N)$.

- **The worst case** will occur when our array will be sorted and we select the smallest or largest indexed element as the pivot.

  Explain:

$$T(N) = T(N-1) + N \cdot \text{constant}$$

$$T(N) = T(N-2) + (N-1) \cdot \text{constant} + N \cdot \text{constant}$$

$$= T(N-2) + 2 \cdot N \cdot \text{constant} - \text{constant}$$

$$T(N) = T(N-3) + 3 \cdot N \cdot \text{constant} - 2 \cdot \text{constant} - \text{constant}$$

...

$$T(N) = T(N-k) + k \cdot N \cdot \text{constant} - \text{constant} \cdot \left(\frac{k(k-1)}{2}\right)$$

$$T(N) = T(0) + N \cdot N \cdot \text{constant} - \text{constant} \cdot \left(\frac{N(N-1)}{2}\right)$$

$$T(N) = \frac{N^2}{2} + \frac{N}{2}$$

  If we put $k = N$ in the above equation, then the worst case complexity is $O(N^2)$.

Variants/Improvements of the Algorithm

Idea: The Median-of-Three Quick Sort is a variant of the Quick Sort algorithm that improves the selection of the pivot element.

1. **Step 1: Choose the Median-of-Three Pivot**
   - Select the first, middle, and last elements.
   - Sort those elements.
   - Use the middle value as the pivot.
2. **Step 2: Partition the Array**
3. **Step 3: Recur at the Left Subarray and Right Subarray**

Stability

Quick sort is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in the case of quick sort, because we are swapping elements according to the pivot's position without considering their original positions.

Comparison or Non-comparison

Comparison.

In-place or Out-of-place

In-place. *Explain: Quick sort sorts the array without requiring additional memory proportional to the input size.*

2.9 Counting Sort

Idea: The basic idea behind Counting Sort is to count the frequency of each distinct element in the input array and use that information to place the elements in their correct sorted positions.

Step-by-step Description

1. Find out the maximum element from the given array.
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.
3. In the `countArray[]`, store the count of each unique element of the input array at their respective indices.
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.
5. Iterate from the end of the input array, because traversing the input array from the 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] ] - 1`.

Complexity Evaluations

- **Time Complexity:** $O(N + M)$, where N and M are the sizes of `InputArray[]` and `countArray[]` respectively.
  - Worst-case: $O(N + M)$.
  - Average-case: $O(N + M)$.
  - Best-case: $O(N + M)$.
- **Space Complexity:** $O(N + M)$.

Variants/Improvements of the Algorithm There are no variants of the algorithm.

Stability Counting Sort is a stable sort as the relative order of elements with equal values is maintained.

Comparison or Non-comparison Non-comparison.

In-place or Out-of-place Counting Sort is not an in-place sorting algorithm as it requires additional space $O(k)$.

2.10 Radix Sort

Ideas

Radix sort is a sorting algorithm that sorts the elements by first grouping the individual digits of the same place value. Then, it sorts the elements according to their increasing/decreasing order.

Step-by-step Description

1. Step 1: Get the maximum value in the array.
2. Step 2: Find the maximum number of digits.
3. Step 3: Take the least significant digit of each element.

4. Step 4: Form groups of numbers and combine groups.
5. Step 5: Repeat Step 3 for the next least significant digits until all the digits in the elements are sorted.
6. Step 6: The final list of elements achieved after the k-th loop is the sorted output.

Complexity Evaluations

- **Time Complexity:** $O(n \cdot k)$
  - n is the number of elements.
  - k is the number of digits in the largest number.
  - Time Analysis:
    - The loop in Step 2 takes $O(k)$ time.
    - The loop in Step 3 takes $O(n)$ time.

    Thus, the time complexity in all cases is $O(n \cdot k)$.
- **Space Complexity:** $O(n + d)$
  - d is the range of the input.

Variants/Improvements of the Algorithm Idea: The Radix Sort algorithm makes use of the Counting Sort algorithm while sorting in every phase.

1. Step 1: Check whether all the input elements have the same number of digits. If not, identify numbers with the maximum number of digits in the list and add leading zeroes to those that do not.
2. Step 2: Take the least significant digit of each element.
3. Step 3: Sort these digits using the logic of Counting Sort and reorder elements based on the output obtained. For example, if the input elements are decimal numbers, each digit can range from 0 to 9, indexing the digits based on these values.
4. Step 4: Repeat Step 2 for the next least significant digits until all the digits in the elements are sorted.
5. The final list of elements achieved after the k-th loop is the sorted output.

Stability
Radix Sort is a stable sort because it maintains the relative order of elements with equal values.
Comparison or Non-comparison
Non-comparison.
In-place or Out-of-place
Radix sort is not an in-place sorting algorithm as it requires additional space to store counts and output arrays for each pass over the digits.

2.11 Flash Sort

Ideas
The main idea behind the flash sort algorithm is to estimate the position of each element in the sorted array and move the elements towards their estimated positions in a single pass through the array.
Step-by-step Description

1. Step 1: Find the minimum and maximum values in the array.
2. Step 2: Linearly divide the array into m buckets.
3. Step 3: Count the number of elements L_b for each bucket b.
4. Step 4: Convert the counts of each bucket into the cumulative sum.
5. Step 5: Rearrange all the elements of each bucket b in positions A_i where $L_{b-1} < i < L_b$.
6. Step 6: Sort each bucket using the insertion sort.

Complexity Evaluations

- **Time Complexity:**
  - Best Case: $O(n)$. Occurs when elements are uniformly distributed, and the maximum displacement of any element from its final sorted position is small.
  - Worst Case: $O(n^2)$. Occurs when elements are not uniformly distributed, leading to inefficient handling of permutations.
  - Average Case: $O(n)$.
- **Space Complexity:** $O(m)$, where m is the number of buckets, which is typically $0.45 \times n$.

Variants/Improvements of the Algorithm

There are no variants of the algorithm.

Stability

Flash Sort is not a stable sorting algorithm because it does not guarantee that the order of similar elements in the sorted array will be maintained.

Comparison or Non-comparison

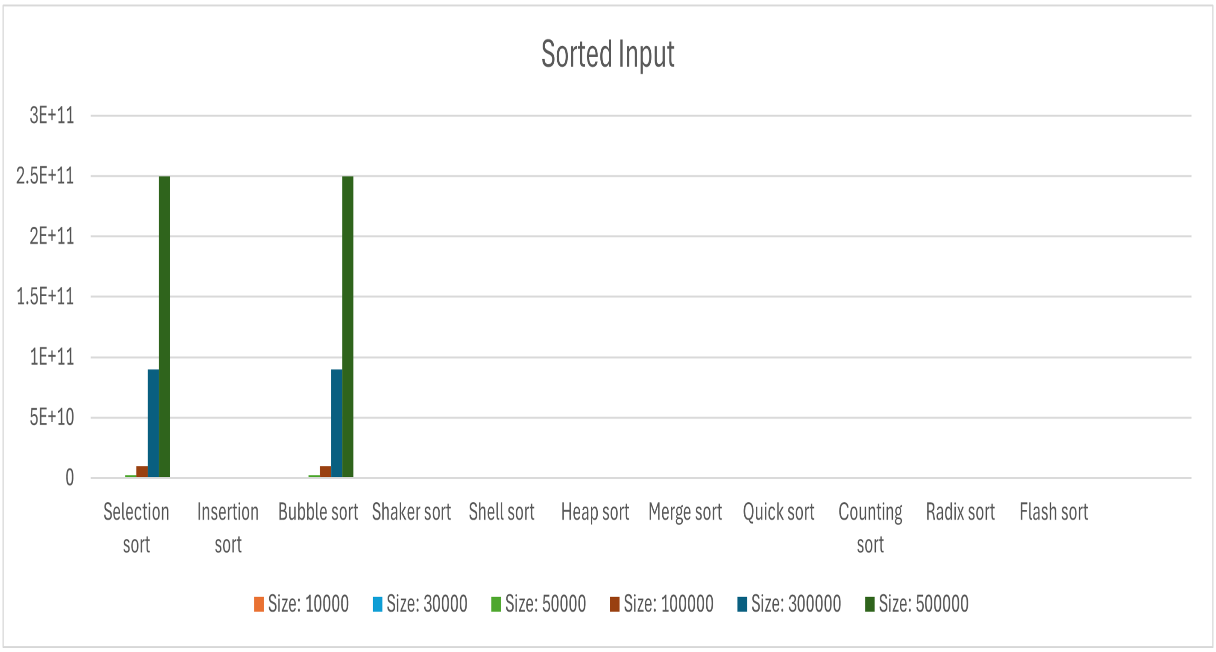
Both.

In-place or Out-of-place

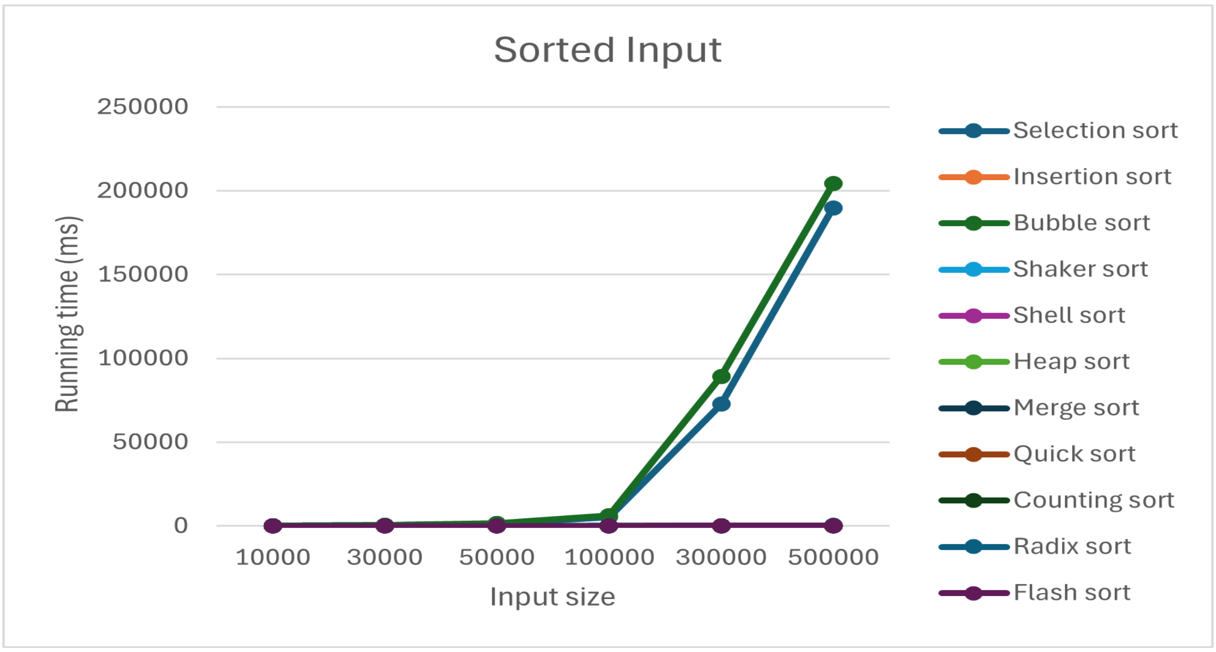
Flash Sort is an in-place sorting algorithm because it doesn't require any extra space to sort the data.

3 Experimental

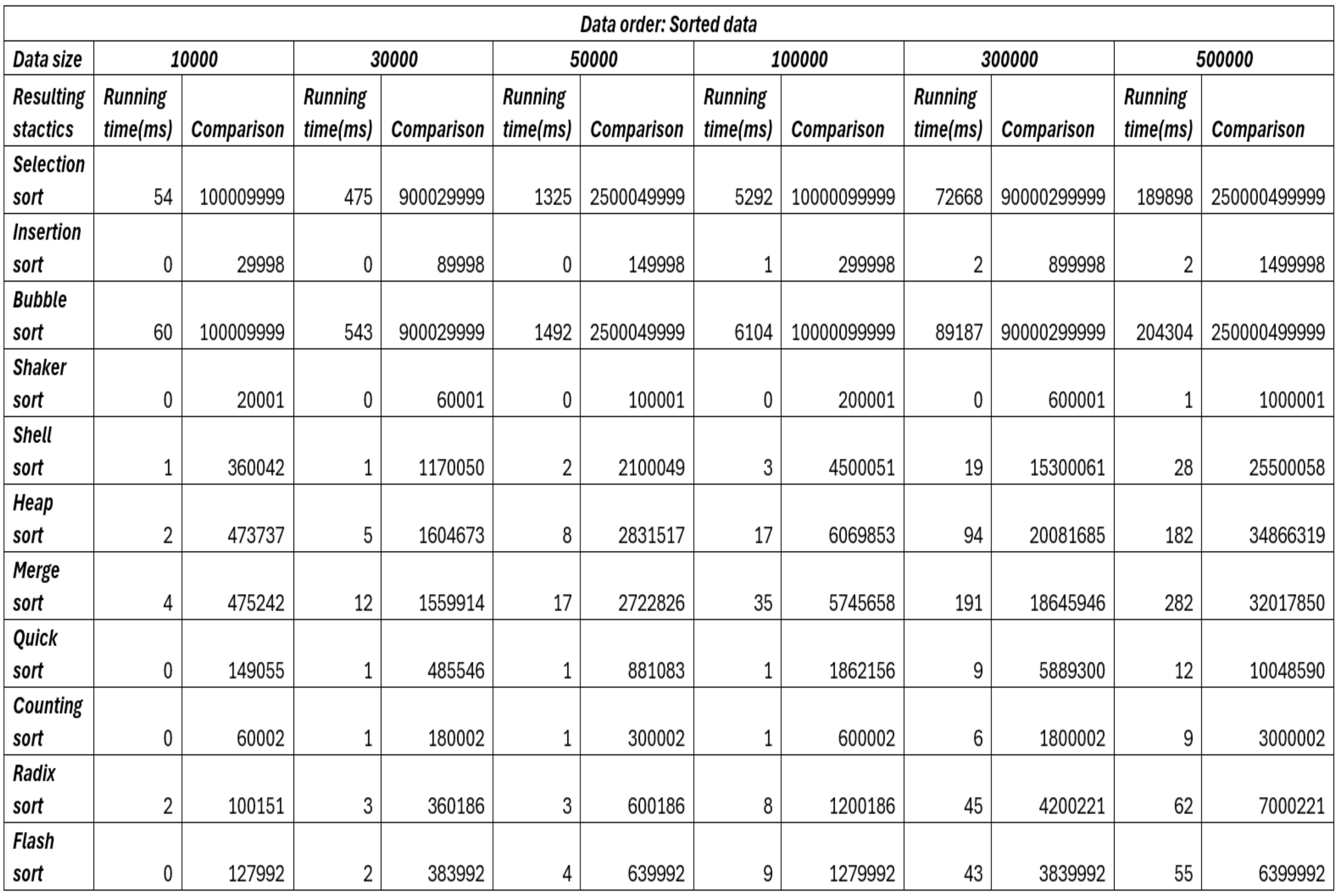
3.1 Sorted Data



Picture 1: Sorted Order Running Time - Bar



Picture 2: Sorted Order Comparison - Line Chart



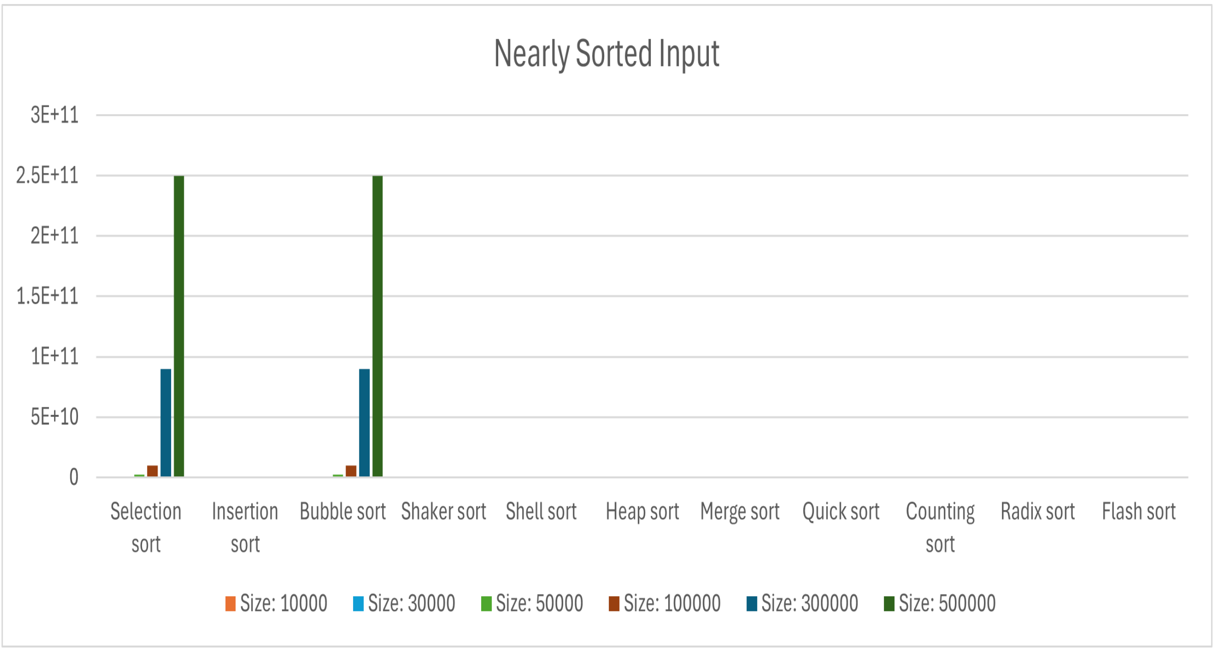
Data order: Sorted data												
Data size	10000		30000		50000		100000		300000		500000	
Resulting stactics	Running time(ms)	Comparison	Running time(ms)	Comparison	Running time(ms)	Comparison	Running time(ms)	Comparison	Running time(ms)	Comparison	Running time(ms)	Comparison
Selection sort	54	100009999	475	900029999	1325	2500049999	5292	10000099999	72668	90000299999	189898	250000499999
Insertion sort	0	29998	0	89998	0	149998	1	299998	2	899998	2	1499998
Bubble sort	60	100009999	543	900029999	1492	2500049999	6104	10000099999	89187	90000299999	204304	250000499999
Shaker sort	0	20001	0	60001	0	100001	0	200001	0	600001	1	1000001
Shell sort	1	360042	1	1170050	2	2100049	3	4500051	19	15300061	28	25500058
Heap sort	2	473737	5	1604673	8	2831517	17	6069853	94	20081685	182	34866319
Merge sort	4	475242	12	1559914	17	2722826	35	5745658	191	18645946	282	32017850
Quick sort	0	149055	1	485546	1	881083	1	1862156	9	5889300	12	10048590
Counting sort	0	60002	1	180002	1	300002	1	600002	6	1800002	9	3000002
Radix sort	2	100151	3	360186	3	600186	8	1200186	45	4200221	62	7000221
Flash sort	0	127992	2	383992	4	639992	9	1279992	43	3839992	55	6399992

Picture 3: Sorted Order Table

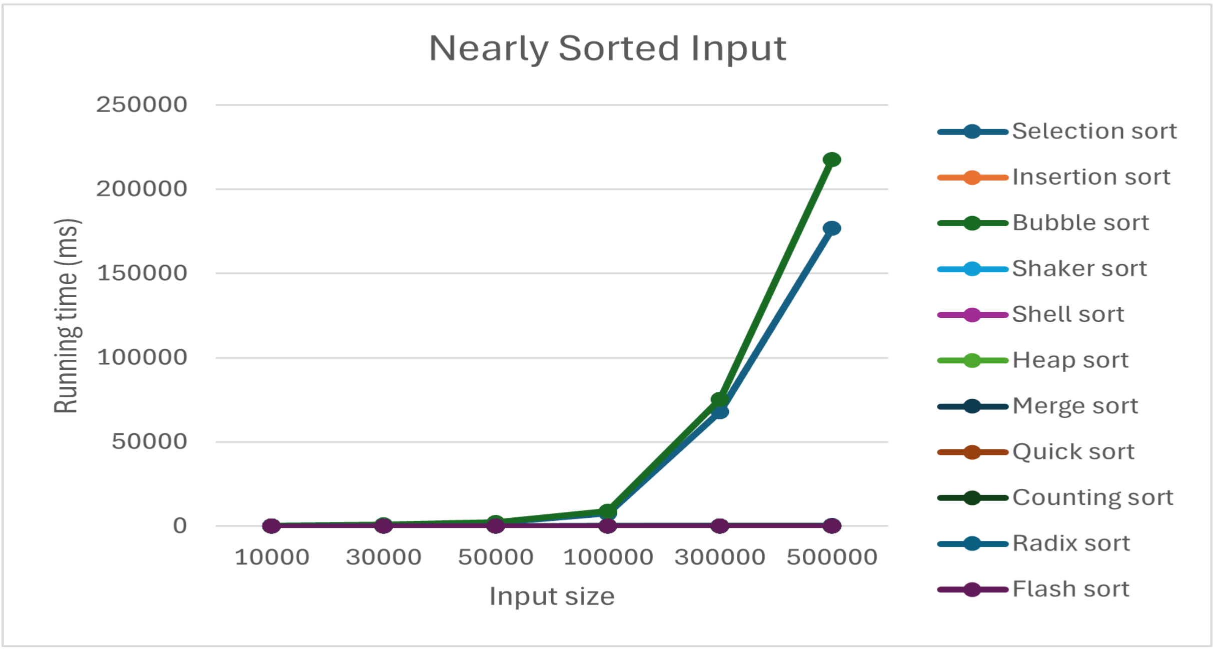
1. **The fastest sorting algorithm:** Insertion sort, Shaker sort (Time: 0-2 ms)
2. **The slowest sorting algorithm:** Bubble sort (Time: 60-204,304 ms)
3. **The sorting algorithm with the fewest comparisons:** Insertion sort (Comparisons: 29,998-1,499,998)
4. **The sorting algorithm with the most comparisons:** Bubble sort, Selection sort (Comparisons: 100,009,999-250,000,499,999)

=> Insertion sort and Shaker sort exhibit near-optimal performance for sorted data, completing in minimal time across all data sizes due to their adaptive nature.

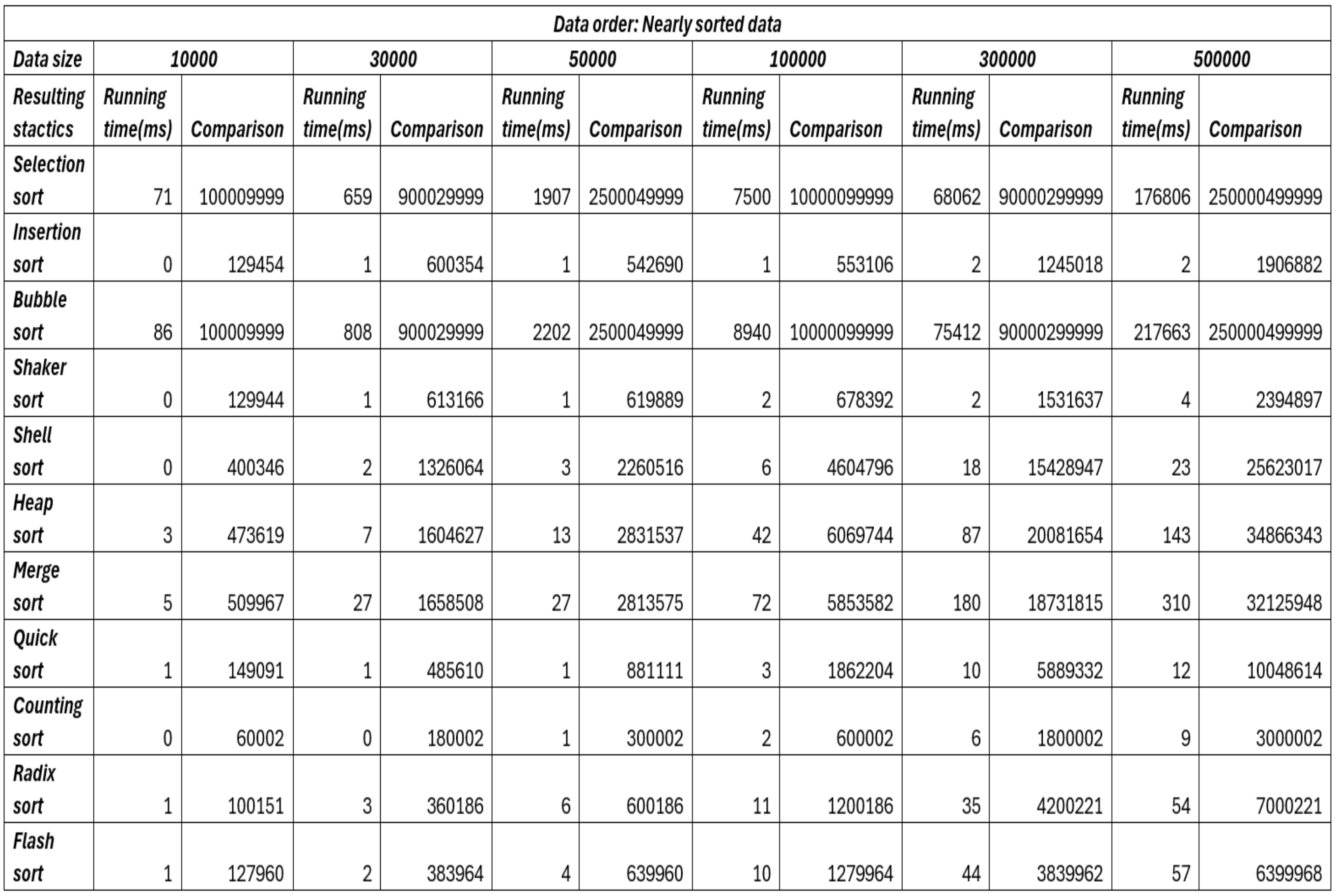
3.2 Nearly Sorted Data



Picture 4: Nearly Sorted Order Running Time - Bar



Picture 5: Nearly Sorted Order Comparison - Line Chart



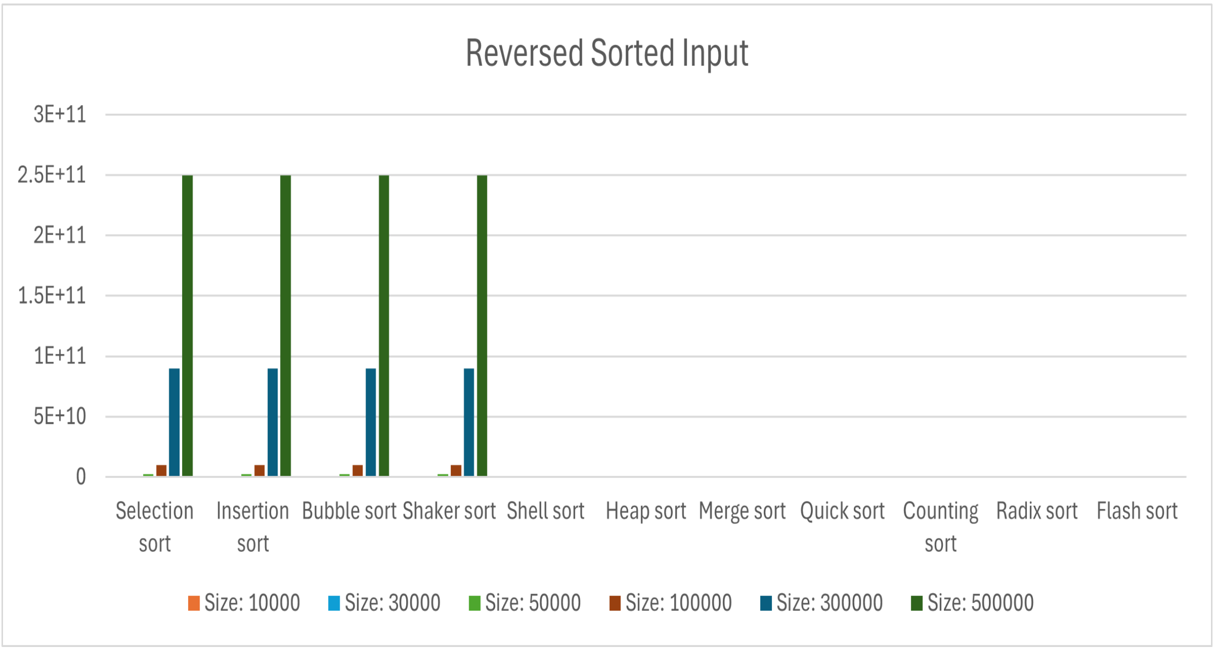
Data order: Nearly sorted data												
Data size	10000		30000		50000		100000		300000		500000	
Resulting stactics	Running time(ms)	Comparison	Running time(ms)	Comparison	Running time(ms)	Comparison	Running time(ms)	Comparison	Running time(ms)	Comparison	Running time(ms)	Comparison
Selection sort	71	100009999	659	900029999	1907	2500049999	7500	10000099999	68062	90000299999	176806	250000499999
Insertion sort	0	129454	1	600354	1	542690	1	553106	2	1245018	2	1906882
Bubble sort	86	100009999	808	900029999	2202	2500049999	8940	10000099999	75412	90000299999	217663	250000499999
Shaker sort	0	129944	1	613166	1	619889	2	678392	2	1531637	4	2394897
Shell sort	0	400346	2	1326064	3	2260516	6	4604796	18	15428947	23	25623017
Heap sort	3	473619	7	1604627	13	2831537	42	6069744	87	20081654	143	34866343
Merge sort	5	509967	27	1658508	27	2813575	72	5853582	180	18731815	310	32125948
Quick sort	1	149091	1	485610	1	881111	3	1862204	10	5889332	12	10048614
Counting sort	0	60002	0	180002	1	300002	2	600002	6	1800002	9	3000002
Radix sort	1	100151	3	360186	6	600186	11	1200186	35	4200221	54	7000221
Flash sort	1	127960	2	383964	4	639960	10	1279964	44	3839962	57	6399968

Picture 6: Nearly Sorted Order Table

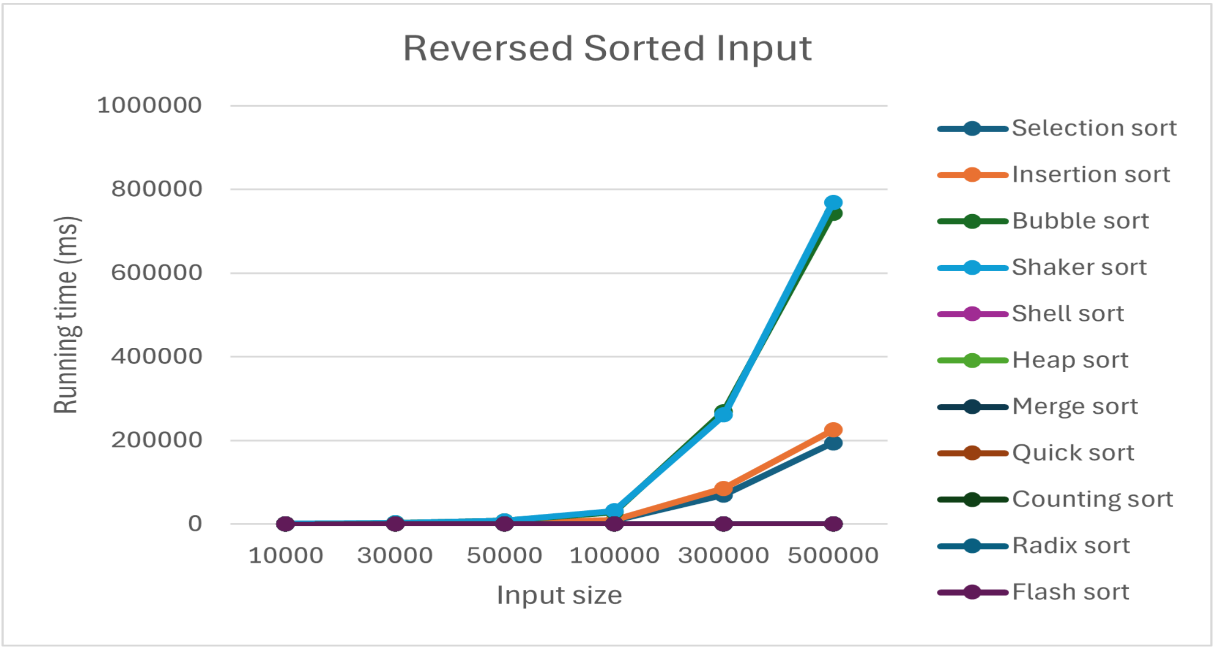
- **The fastest sorting algorithm:** Insertion sort, Shaker sort(Time: 0-4 ms)
- **The slowest sorting algorithm:** Bubble sort (Time: 86-217,663 ms)
- **The sorting algorithm with the fewest comparisons:** Counting sort (Comparisons: 60,002-3,000,002)
- **The sorting algorithm with the most comparisons:** Bubble sort, Selection sort (Comparisons: 100,009,999-250,000,499,999)

=> Insertion sort and Shaker sort continue to perform exceptionally well on nearly sorted data, maintaining very low running times.

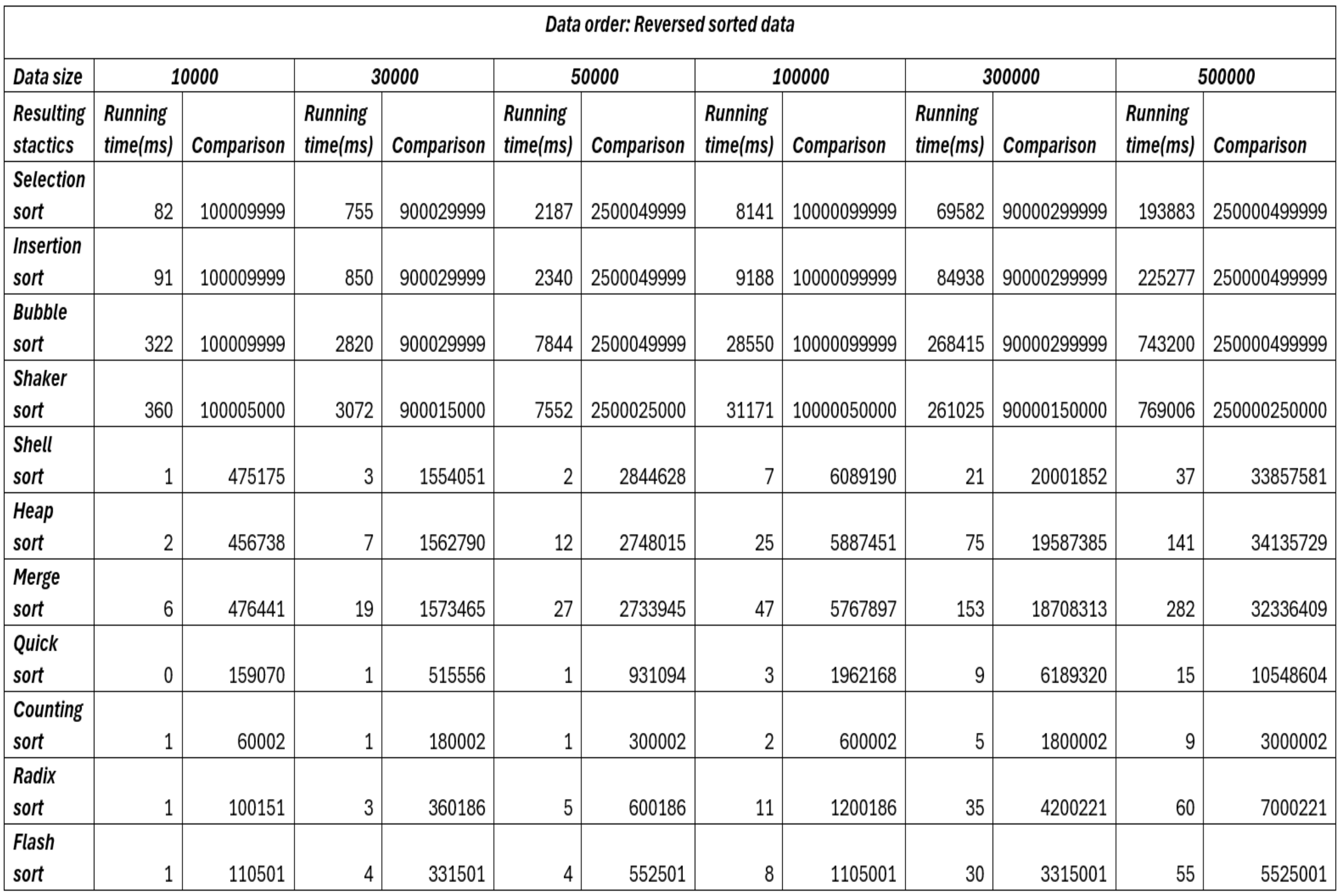
3.3 Reversed Sorted Data



Picture 7: Reversed Sorted Order Running Time - Bar



Picture 8: Reversed Sorted Order Comparison - Line Chart

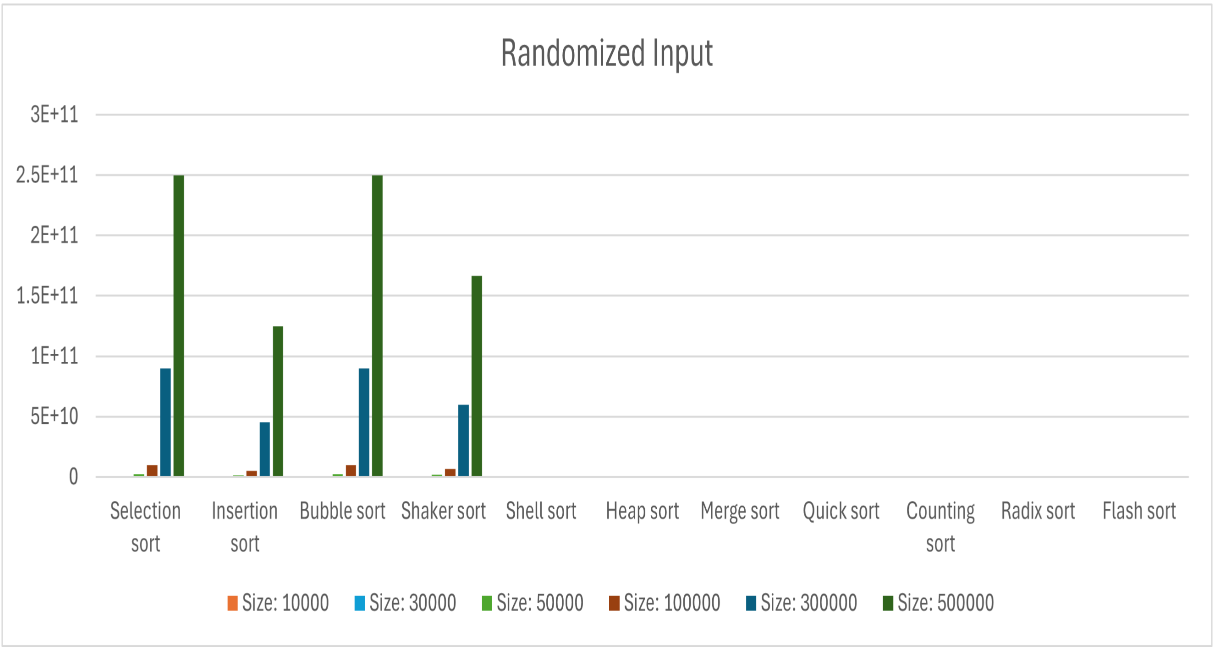


Data order: Reversed sorted data												
Data size	10000		30000		50000		100000		300000		500000	
Resulting stactics	Running time(ms)	Comparison	Running time(ms)	Comparison	Running time(ms)	Comparison	Running time(ms)	Comparison	Running time(ms)	Comparison	Running time(ms)	Comparison
Selection sort	82	100009999	755	900029999	2187	2500049999	8141	10000099999	69582	90000299999	193883	250000499999
Insertion sort	91	100009999	850	900029999	2340	2500049999	9188	10000099999	84938	90000299999	225277	250000499999
Bubble sort	322	100009999	2820	900029999	7844	2500049999	28550	10000099999	268415	90000299999	743200	250000499999
Shaker sort	360	100005000	3072	900015000	7552	2500025000	31171	10000050000	261025	90000150000	769006	250000250000
Shell sort	1	475175	3	1554051	2	2844628	7	6089190	21	20001852	37	33857581
Heap sort	2	456738	7	1562790	12	2748015	25	5887451	75	19587385	141	34135729
Merge sort	6	476441	19	1573465	27	2733945	47	5767897	153	18708313	282	32336409
Quick sort	0	159070	1	515556	1	931094	3	1962168	9	6189320	15	10548604
Counting sort	1	60002	1	180002	1	300002	2	600002	5	1800002	9	3000002
Radix sort	1	100151	3	360186	5	600186	11	1200186	35	4200221	60	7000221
Flash sort	1	110501	4	331501	4	552501	8	1105001	30	3315001	55	5525001

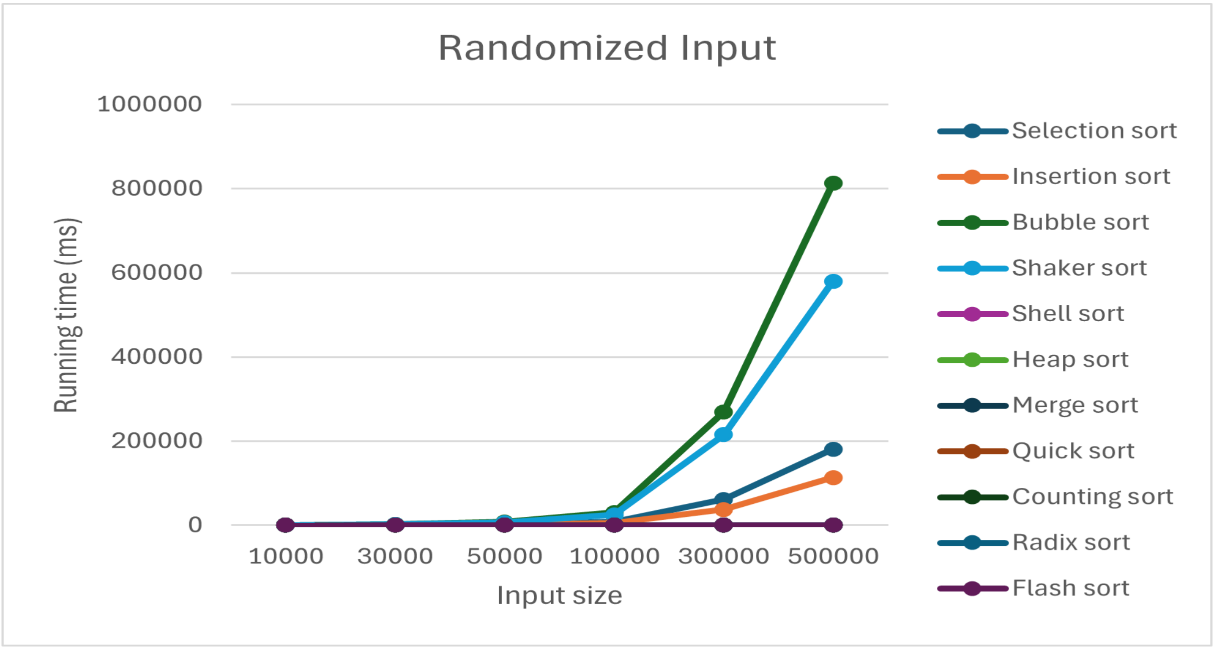
Picture 9: Reversed Sorted Order Table

- **The fastest sorting algorithm:** Counting sort (Time: 0 - 9 ms)
- **The slowest sorting algorithm:** Shaker sort (Time: 360 - 769,006 ms)
- **The sorting algorithm with the fewest comparisons:** Counting sort (Comparisons: 60,002 - 3,000,002)
- **The sorting algorithm with the most comparisons:** Insertion sort, Selection sort, Bubble sort (Comparisons: 100,009,999 - 250,000,499,999)

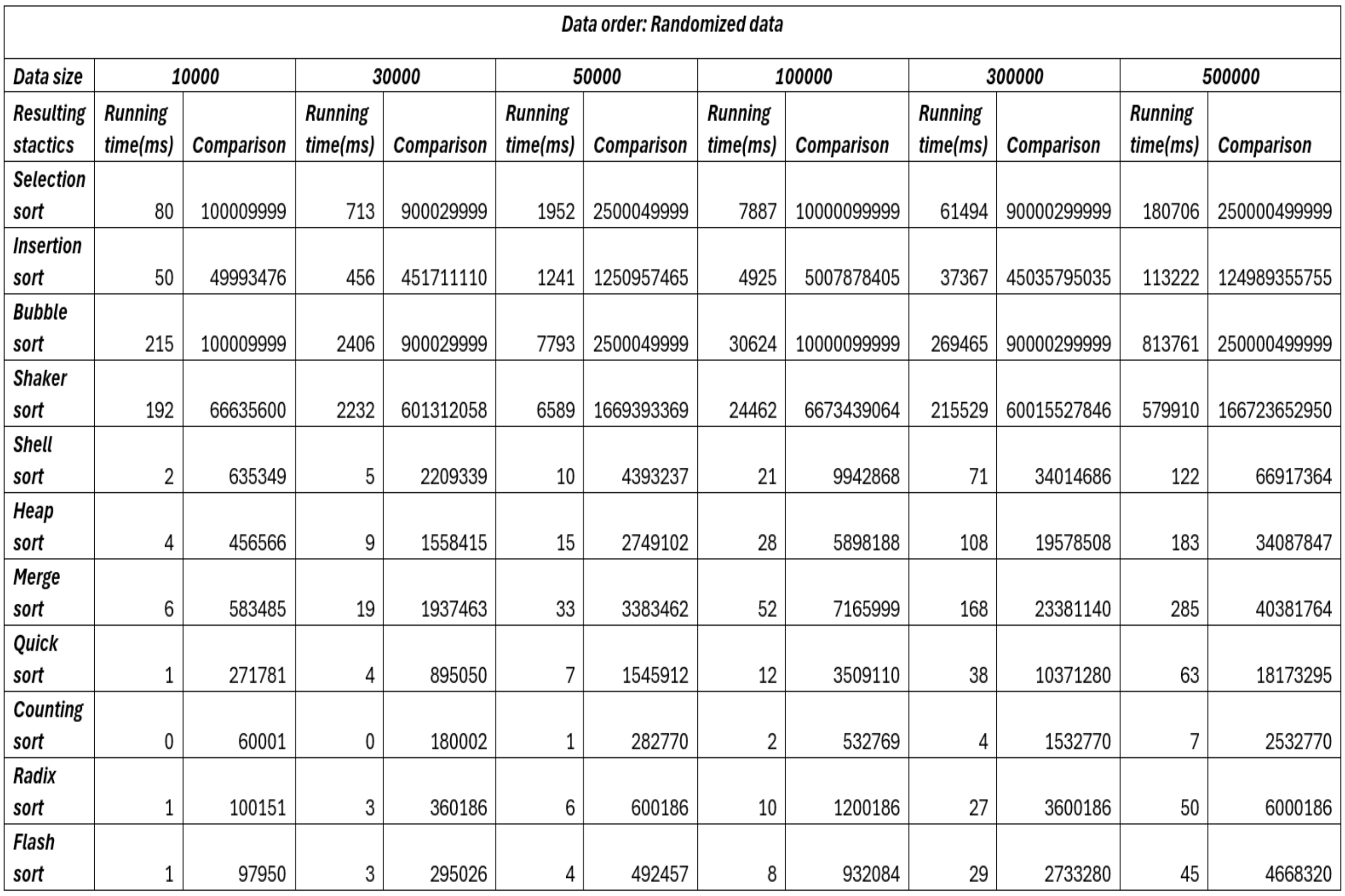
3.4 Randomized Data



Picture 10: Randomized Order Running Time - Bar



Picture 11: Randomized Order Comparison - Line Chart



Data order: Randomized data												
Data size	10000		30000		50000		100000		300000		500000	
Resulting stactics	Running time(ms)	Comparison	Running time(ms)	Comparison	Running time(ms)	Comparison	Running time(ms)	Comparison	Running time(ms)	Comparison	Running time(ms)	Comparison
Selection sort	80	100009999	713	900029999	1952	2500049999	7887	10000099999	61494	90000299999	180706	250000499999
Insertion sort	50	49993476	456	451711110	1241	1250957465	4925	5007878405	37367	45035795035	113222	124989355755
Bubble sort	215	100009999	2406	900029999	7793	2500049999	30624	10000099999	269465	90000299999	813761	250000499999
Shaker sort	192	66635600	2232	601312058	6589	1669393369	24462	6673439064	215529	60015527846	579910	166723652950
Shell sort	2	635349	5	2209339	10	4393237	21	9942868	71	34014686	122	66917364
Heap sort	4	456566	9	1558415	15	2749102	28	5898188	108	19578508	183	34087847
Merge sort	6	583485	19	1937463	33	3383462	52	7165999	168	23381140	285	40381764
Quick sort	1	271781	4	895050	7	1545912	12	3509110	38	10371280	63	18173295
Counting sort	0	60001	0	180002	1	282770	2	532769	4	1532770	7	2532770
Radix sort	1	100151	3	360186	6	600186	10	1200186	27	3600186	50	6000186
Flash sort	1	97950	3	295026	4	492457	8	932084	29	2733280	45	4668320

Picture 12: Randomized Order Table

- **The fastest sorting algorithm:** Counting sort (Time: 0 - 7 ms)
- **The slowest sorting algorithm:** Bubble sort (Time: 215 - 813,761 ms)
- **The sorting algorithm with the fewest comparisons:** Counting sort (Comparisons: 60,001 - 2,532,770)
- **The sorting algorithm with the most comparisons:** Selection sort, Bubble sort (Comparisons: 100,009,999 - 250,000,499,999)

3.5 Overall

Considering the stability of the 11 sorting algorithms based on their performance across sorted, nearly sorted, reverse sorted, and randomized data:

- **Selection Sort:** Stable - Consistently $O(n^2)$ regardless of the data arrangement.
- **Insertion Sort:** Unstable - Efficient on sorted or nearly sorted data ($O(n)$), but $O(n^2)$ on random or reverse-sorted data.
- **Bubble Sort:** Stable - Consistently $O(n^2)$ regardless of the data arrangement.
- **Shaker Sort:** Unstable - It is $O(n)$ on sorted data but $O(n^2)$ otherwise.
- **Shell Sort:** Unstable - Performance varies based on data and gap sequence, ranging from $O(n \log n)$ to $O(n^2)$.

- **Heap Sort:** Stable - Consistently $O(n \log n)$ regardless of the data arrangements.
- **Merge Sort:** Stable - Consistently $O(n \log n)$ regardless of the data arrangements.
- **Quick Sort:** Unstable - Average and best cases are $O(n \log n)$, but worst case can be $O(n^2)$ with certain data distributions.
- **Counting Sort:** Stable - Maintains $O(n + k)$ regardless of how the data is organized.
- **Radix Sort:** Stable - Performs in $O(nk)$ consistently across all data types.
- **Flash Sort:** Unstable - Varies from $O(n)$ in the best scenario to $O(n^2)$ depending on the data distribution.

In summary, the following sorting algorithms are stable as their performance doesn't change with different data types:

- Selection Sort
- Bubble Sort
- Heap Sort
- Merge Sort
- Counting Sort
- Radix Sort

In contrast, the following sorting algorithms are unstable due to their varying performance depending on the initial data arrangement:

- Insertion Sort
- Shaker Sort
- Shell Sort
- Quick Sort
- Flash Sort

4 Project organization and Programming notes

4.1 Project organization

There are a total of 5 .cpp files and 5 .h header files in the submission, each of them has a distinguished functionality:

- **main.cpp**: The main entry file, used to run the program, validate the input arguments, and determine the type of command to run.
- **Command.cpp and Command.h**: Contain functions that perform different operations based on the input arguments provided.
- **DataGenerator.cpp and DataGenerator.h**: Contain functions to initialize an array in a total of four different data arrangements (random, sorted, nearly sorted, and reversed).
- **Experiment.cpp and Experiment.h**: These files contain functions to conduct experiments on the 11 sorting algorithms with different data arrangements and sizes, focusing on comparison counting and time measurement.
- **Sort.cpp and Sort.h**: Contain 11 sorting algorithms. For each algorithm, there are 11 functions with a comparison counting argument and 11 overloaded functions without it. Additionally, there are 11 functions responsible for measuring the running time of these sorting algorithms.

To build an .exe file with g++:

1. Open Command Prompt and locate the folder containing all those .cpp and .h files.
2. Type the following command to begin building the .exe file:

```
g++ main.cpp Command.cpp DataGenerator.cpp
             Sort.cpp -o sorting.exe
```

3. An .exe file should be created, then in Command Prompt, run that file together with an appropriate command line

4.2 Programming notes

Some special libraries were used to make the program work precisely:

- `<time.h>` library (used in `DataGenerator.cpp` and `Sort.cpp`): This library was used to generate random factors while creating an array (e.g., random array or nearly sorted array) and also used to measure the running time of an algorithm.
- `<cstdlib>` library (used in `main.cpp`): The function `atoi()` in this library was used to validate and interpret a number input from the command line.

Some other notes:

- The `findSortFuncAndCalculate` function (stored in `Command.cpp`) determines the type of sorting algorithm based on the given input argument. It returns the comparison count and time measurement through pass-by-reference arguments.

- The `conductExperiment` function (stored in `Experiment.cpp`) is used to perform experiments on the 11 sorting algorithms with distinct data arrangements and sizes. By default, this function is not executed. To activate it, ensure that the `Experiment.h` header is included in the `main.cpp` file and rebuild the `.exe` file (`g++ main.cpp Command.cpp DataGenerator.cpp Sort.cpp Experiment.cpp`). After rebuilding, call the function in the `main` function within `main.cpp`.

5 References

1. Selection sort algorithm presentation, GeeksforGeeks. Available at: `https://www.geeksforgeeks.org/selection-sort-algorithm-2/`

2. Insertion sort algorithm presentation, GeeksforGeeks. Available at: `https://www.geeksforgeeks.org/insertion-sort-algorithm/`

3. Bubble sort algorithm presentation, GeeksforGeeks. Available at: `https://www.geeksforgeeks.org/bubble-sort-algorithm/`

4. Binary insertion sort algorithm presentation, Baeldung. Available at: `https://www.baeldung.com/cs/binary-insertion-sort`

5. Selection sort, Insertion sort, Bubble sort algorithm presentation, Anany Levitin (2012) "Introduction to the Design and Analysis of Algorithms" Third Edition, Pearson. Chapter 3 & 4

6. Selection sort, Insertion sort, Bubble sort, Shaker sort source code and algorithm presentation, Data structures and Algorithms, Pr Nguyen Ngoc Thao, pages 7, 9, 15, 16, 26, 26, 31. Available at: `https://drive.google.com/file/d/1GMOitvXkt1KDyAyTf_8KP_9OMYFsjdNG/view`

7. Shaker sort algorithm presentation, Wikipedia. Available at: https://vi.wikipedia.org/wiki/Thuat_toan_sap_xep_cocktail

8. Shaker sort algorithm presentation, Javatpoint. Available at: `https://www.javatpoint.com/cocktail-sort`

9. Shell sort source code and algorithm presentation, GeeksforGeeks. Available at: `https://www.geeksforgeeks.org/shellsort/`

10. Shell sort algorithm presentation, Baeldung. Available at: `https://www.baeldung.com/cs/shellsort-complexity`

11. Heap sort algorithm presentation, Javatpoint. Available at: `https://www.javatpoint.com/heap-sort`

12. Heap sort algorithm presentation, Wikipedia. Available at: `https://en.wikipedia.org/wiki/Heapsort`

13. Smooth sort algorithm presentation, GeeksforGeeks. Available at: `https://www.geeksforgeeks.org/introduction-to-smooth-sort/`

14. Merge sort algorithm presentation, Javatpoint. Available at: `https://www.javatpoint.com/merge-sort-pseudocode-cpp`

15. Heap sort source code and algorithm presentation, Data structures and Algorithms, Pr Nguyen Ngoc Thao, pages 6, 11, 9, 14, 22, 23, 25-27. Available at: `https://drive.google.com/file/d/1ePrn8yOWM3OMutNBVUjOm5L8ik474izQ/view`

16. Merge sort source code, GeekforGeeks, Available at: `https://www.geeksforgeeks.org/merge-sort`

17. Quick sort algorithm presentation, GeeksforGeeks. Available at: `https://www.geeksforgeeks.org/quick-sort-algorithm/`

18. Quick sort algorithm presentation, GeeksforGeeks. Available at:
`https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-quick-sort/`

19. Counting sort algorithm presentation, GeeksforGeeks. Available at:
`https://www.geeksforgeeks.org/counting-sort/`

20. Counting sort algorithm presentation, CodingGeek. Available at:
`https://www.codingeek.com/algorithms/counting-sort-explanation-pseudocode-and-implementation/`

21. Radix sort algorithm presentation, TutorialsPoint. Available at:
`https://www.tutorialspoint.com/data_structures_algorithms/radix_sort_algorithm.htm`

22. Radix sort algorithm presentation, Programiz. Available at:
`https://www.programiz.com/dsa/radix-sort`

23. Radix sort algorithm presentation, GeeksforGeeks. Available at:
`https://www.geeksforgeeks.org/time-and-space-complexity-of-radix-sort-algorithm/`

24. Flash sort algorithm presentation, Algorithm Examples. Available at:
`https://javascript.algorithmexamples.com/web/Sorts/flashSort.html`

25. Flash sort algorithm presentation, Educative. Available at:
`https://www.educative.io/answers/what-is-flash-sort`

26. Quick sort source code and algorithm presentation, Data structures and Algorithms, Pr Nguyen Ngoc Thao, pages 32, 33. Available at:
`https://drive.google.com/file/d/1ePrn8yOWM3OMutNBVUjOm5L8ik474izQ/view`

27. Counting sort, radix sort source code, Data structures and Algorithms, Pr Nguyen Ngoc Thao, pages 12, 13, 19. Available at:
`https://drive.google.com/file/d/13p6BfM9vS1HPpfv4lLtIRNR4Tnq4n2QT/view`

28. Flash sort source code, CodeLearn. Available at:
`https://codelearn.io/sharing/flash-sort-thuat-toan-sap-xep-than-thanh`