HCMUS
Viet Nam National University
Ho Chi Minh City
University of Science

fit@hcmus

Sorting Algorithms

Topic: Algorithm survey and evaluation

Course: Data Structures and Algorithms

Supervisor:
MSc. Phan Thi Phuong Uyen
TA. Tran Hoang Quan

Group 07

Full Name	Student ID
Tran Ly Nhat Hao	23127187
Pham Thanh Dat	23127170
Huynh Hao Nam	23127431

Ho Chi Minh City, 28th June 2024

Contents

List of Figures

Student Information

1. **Name:** Tran Ly Nhat Hao - 23CLC08
   - **Student ID:** 23127187
   - **Github:** tranlynhathao
   - **Course:** Data Structures and Algorithms - 23CLC08

2. **Name:** Pham Thanh Dat - 23CLC08
   - **Student ID:** 23127170
   - **Github:** thanhdatttt
   - **Course:** Data Structures and Algorithms - 23CLC08

3. **Name:** Huynh Hao Nam - 23CLC08
   - **Student ID:** 23127431
   - **Github:** Hownameee
   - **Course:** Data Structures and Algorithms - 23CLC08

4. **Github Repository:** Sorting_Algorithms

Folder Structure

```
.
├── 07.exe
├── Checklist.xlsx
├── Report.pdf
└── SOURCE
    ├── CommandLine.cpp
    ├── CommandLine.h
    ├── DataGenerator.cpp
    ├── DataGenerator.h
    ├── Lib.h
    ├── SortWithCompare.cpp
    ├── SortWithCompare.h
    ├── SortWithTime.cpp
    ├── SortWithTime.h
    └── main.cpp

2 directories, 13 files
```

1 Introduction

This report is prepared for the Data Structures and Algorithms project by group 07 under the supervision of MSc. Phan Thi Phuong Uyen and TA. Tran Hoang Quan.

The aim of this report is to provide a comprehensive survey and evaluation of the running time and the number of comparisons of various sorting algorithms, highlighting their core concepts, step-by-step explanations, complexity analyses, and possible optimizations.

Sorting algorithms are fundamental to computer science, playing a critical role in data organization, searching efficiency, and overall computational performance. Understanding these algorithm is essential for solving the problems efficiently.

This report covers an extensive range of sorting algorithms including 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. Each algorithm is discussed in terms of its core concepts, detailed explanations, complexity analyses, and any notable variants or optimizations. The report also encompasses a comprehensive survey and evaluation of the mentioned sorting algorithms focusing on the time of their performance and their number of comparisons.

2 Sorting Algorithms

2.1 Selection Sort

2.1.1 Core concept

Selection sort is comparison-based sorting algorithm. The algorithm repeatedly selects the smallest (or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted part. This process is repeated for the remaining unsorted portion until the entire list is sorted. [1]

2.1.2 Explanation

One way to explain the algorithm: [2]

- **Initialization:** Start with the first element in the list (index 0) as the current minimum.
- **Find the Maximum:** Traverse the unsorted part of the list to find the maximum element.
- **Swap:** Swap the found maximum element with the first element of the unsorted part.
- **Move the Boundary:** Move the boundary between the sorted and unsorted parts one element to the left.
- **Repeat:** Repeat for the next position in the list until the entire list is sorted.

Step-by-step:

- Consider the following array as an example: 29, 10, 14, 37, 13
- Beginning with five integers as unsorted part, you select the largest which is 37 and swap it with the last integer 13, then 37 is now in sorted part.
- After first swap: 29, 10, 14, 13, 37
- Next you select the largest integer 29 from among the first four integers in the array and swap it with the next-to-last integer in the array 13. 29 is now the first element of sorted part.
- After second swap: 13, 10, 14, 29, 37
- Notice that 14 is the largest element in unsorted part and it also is in the proper position in sorted part, but the algorithm ignores this fact and performs a swap of 14 with itself.
- After third swap: 13, 10, 14, 29, 37
- Finally, select the 13 and swap it with the item in the second position of the array—10.
- After fourth swap: 10, 13, 14, 29, 37
- The array is now sorted into ascending order.

2.1.3 Complexity analysis

Time complexity: At each iteration, the array is divided into two subarrays, the left part is sorted, and the right part is unsorted.
Iteration 1: Perform $n-1$ comparisons in n numbers, choose the smallest element to move to $a[0]$
Iteration 2: Perform $n-2$ comparisons in $(n-1)$ numbers, choose the smallest element to move to $a[1]$
...
Iteration $n-1$: Perform 1 comparison in 2 numbers $a[n-2]$ and $a[n-1]$, choose the smallest element to move to $a[n-1]$, then $a[n-1]$ becomes the largest number, ending the algorithm.
Let $f(n)$ be the cost function of the algorithm. We have:

$$f(n) = (n-1) + (n-2) + \ldots + 3 + 2 + 1 = \frac{n(n-1)}{2} \in O(n^2)$$

- **Best Case:** $O(n^2)$ – Selection sort always performs $O(n^2)$ comparisons and swaps regardless of the initial order of the elements.
- **Average Case:** $O(n^2)$ – Same as the best case, selection sort does not change its behavior based on the input data.
- **Worst Case:** $O(n^2)$ – Selection sort performs $O(n^2)$ comparisons and swaps in all cases.

Space complexity:

- $O(1)$ – Selection sort is an in-place sorting algorithm, meaning it requires only a constant amount of additional memory space.

2.1.4 Variants and optimizations

An optimized variant is heap sort which we will discuss it later. This sorting algorithm is an improvement of selection sort. Heap sort reduces the time to find the smallest or largest element in unsorted part by building a max heap or min heap. Heap is a data structure which is used to find the minimum (min heap) or maximum (max heap) element more efficiently. This reduces the time complexity to $O(n \log n)$.

2.2 Insertion Sort

2.2.1 Core Concepts

Insertion sort orders a list of values by repetitively inserting a particular value into a sorted subset of the list. More specifically, consider the first item to be a sorted sublist of length 1, insert second item into sorted sublist, shifting first item if needed. Insert third item into sorted sublist, shifting items 1-2 as needed. Repeat until all values have been inserted into proper positions. [3]

2.2.2 Step-by-step Explaination

- Step 1: Assume the first element is already sorted. Pick the next element and find its correct position among the already sorted elements.
- Step 2: Shift all the elements greater than the picked element to one position ahead to make space for the picked element.
- Step 3: Insert the picked element into its correct position.
- Step 4: Repeat the above steps for all elements in the array.

2.2.3 Complexity Analysis:

At each iteration, the array is divided into two subarrays, the left part is sorted, and the right part is unsorted. Accelerate the search using binary search.
Iteration 1: Perform 1 comparison to find the insertion position for $a[1]$
Iteration 2: Perform at least 1 comparison, at most 2 comparisons to find the insertion position for $a[2]$
...
Iteration $n-1$**:** Perform at least 1 comparison, at most $(n-1)$ comparisons to find the insertion position for $a[n-1]$, ending the algorithm
In each iteration, the number of assignments is equal to the number of comparisons made.
Let $f(n)$ be the cost function of the algorithm (based on the number of comparisons and assignments):

$$f(n) = (n_1 + n_2 + \ldots + n_{n-1}) \times 2 \quad \text{where:} \quad n_1 = 1, \quad n_2 \in [1,2], \quad n_3 \in [1,3], \ldots, \quad n_{n-1} \in [1, n-1]$$

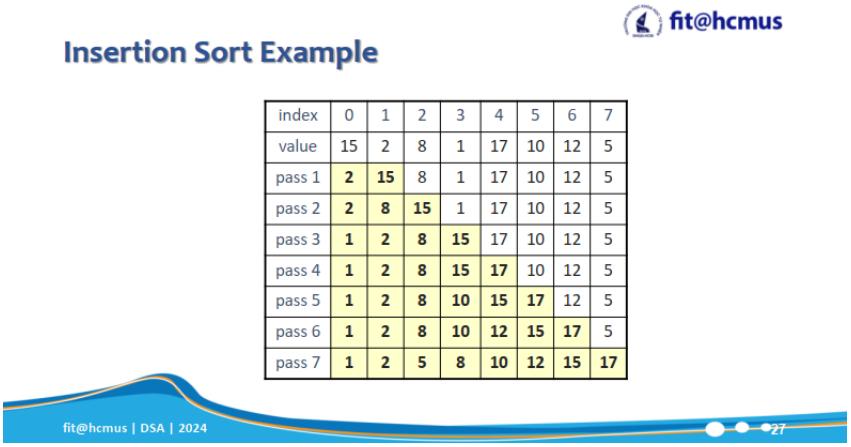
Time Complexity:

- Best-case: $O(n)$
- Average-case: $O(n^2)$

- Worst-case: $O(n^2)$

Space complexity: O(1) (in-placed)

An example of Insertion sort [3]



index	0	1	2	3	4	5	6	7
value	15	2	8	1	17	10	12	5
pass 1	**2**	**15**	8	1	17	10	12	5
pass 2	**2**	**8**	**15**	1	17	10	12	5
pass 3	**1**	**2**	**8**	**15**	17	10	12	5
pass 4	**1**	**2**	**8**	**15**	**17**	10	12	5
pass 5	**1**	**2**	**8**	**10**	**15**	**17**	12	5
pass 6	**1**	**2**	**8**	**10**	**12**	**15**	**17**	5
pass 7	**1**	**2**	**5**	**8**	**10**	**12**	**15**	**17**

Figure 2.1: Insertion Sort Demo

2.3 Shell Sort

2.3.1 Core concepts

Shell sort is an enhanced version of insertion sort designed to address the problem of elements being far from their correct positions. Shell sort works by initially sorting elements that are far apart from each other, then progressively reducing the gap between elements to be compared. When gap is 1, it is an insertion-sort with nearly sorted array which makes the final pass much faster. This approach efficiently reduces the problem of elements being far from their correct positions, resulting in a quicker sorting process overall.

2.3.2 Step-by-step Explanations

- Step 1:Find the gap suitable for array, start by gap is equal to 1, gap multiplicate with 3 then plus 1, repeat this step if gap smaller than n. Then divide gap by 3 to find the largest gap that smaller than n.
- Step 2: Sort the elements in certain gap, use the idea of insertion-sort to sort the elements in certain gap (start from i is equal to gap compare a[i] with a[i – gap]).
- Step 3: Divide gap by 3 then repeat step 2 until gap is equal to 1, it is an insertion-sort with the array is nearly sorted.

2.3.3 Complexity Analysis

Time complexity analysis:

- Best Case: $O(n \log n)$
- Average Case: Typically around $O(n^{5/4})$ to $O(n^{4/3})$
- Worst Case: $O(n^{3/2})$

Space complexity analysis: O(1) because it is an inplaced sorting algorithm

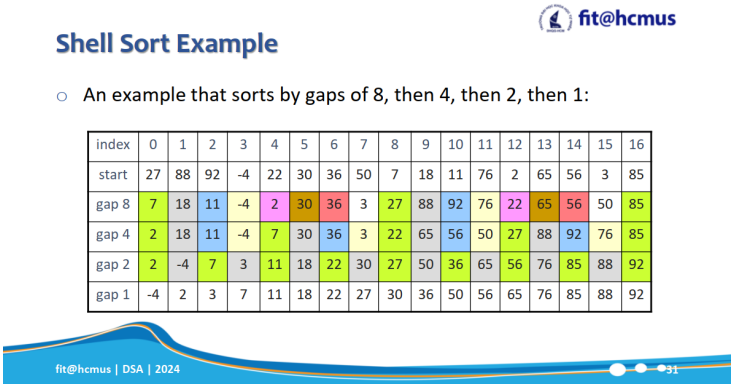
2.3.4 Variants and Optimizations

Optimizations: Depend on the data and the number of elements in array, we can choose suitable gap for having better performance. Two common ways to choose:

Gap divide by 3: $..., 121, 40, 13, 4, 1$

Gap divide by 2: $..., 31, 15, 7, 3, 1$

An example of Shell sort [3]



index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
start	27	88	92	-4	22	30	36	50	7	18	11	76	2	65	56	3	85
gap 8	7	18	11	-4	2	30	36	3	27	88	92	76	22	65	56	50	85
gap 4	2	18	11	-4	7	30	36	3	22	65	56	50	27	88	92	76	85
gap 2	2	-4	7	3	11	18	22	30	27	50	36	65	56	76	85	88	92
gap 1	-4	2	3	7	11	18	22	27	30	36	50	56	65	76	85	88	92

Figure 2.2: Shell Sort Demo

2.4 Bubble Sort

2.4.1 Core concept

Bubble sort is also comparison-based sorting algorithm. The bubble sort compares adjacent items and exchanges them if they are out of order. This sort usually requires several passes over the data. [2]

2.4.2 Explanation

One way to explain this algorithm:
Traverse from left and compare adjacent elements and the higher one is placed at right side. In this way, the largest element is moved to the rightmost end at first. This process is then continued to find the second largest and place it and so on until the data is sorted. [4]

Lets consider the following array as an example: 29, 10, 14, 37, 13 [2]

The first pass: Compare the items in the first pair 29 and 10 and exchange them because they are out of order. Next you consider the second pair 29 and 14 and exchange these items because they are out of order. The items in the third pair 29 and 37 are in order, and so you do not exchange them. Finally, you exchange the items in the last pair 37 and 13.

The second pass: During the second pass of the bubble sort, return to the beginning of the array and consider pairs of items in exactly the same manner as the first pass. Do not, however, include the last and largest—item of the array. That is, the second pass considers the first n – 1 items of the array. After the second pass, the second-largest item in the array will be in its proper place in the next-to-last position of the array.
Now, ignoring the last two items, which are in order, continue with subsequent passes until the array is sorted.
The first two passes of a bubble sort of an array of five integers [2]

2.4.3 Complexity analysis

Time complexity: In each iteration of variable i, the smallest element in the segment $a[i \dots n-1]$ will "bubble up" and be placed at $a[i]$. Therefore, there are a total of $(n-1)$ iterations for variable i.
Iteration 1: there are $(n-1)$ comparisons between $a[j]$ and $a[j-1]$, with j going from $n-1$ to 1.

Iteration 2: there are $(n-2)$ comparisons between $a[j]$ and $a[j-1]$, with j going from $n-2$ to 1.
...
Iteration $n-1$: there is 1 comparison between $a[1]$ and $a[0]$.
Let $f(n)$ be the number of comparisons and also the number of basic operations of the Bubble Sort algorithm, we have:

$$f(n) = (n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n-1)}{2}$$

regardless of the distribution of the array elements.

- Best case: $O(n)$ - Occurs when the original data is already sorted, it uses one pass, during which n – 1 comparisons and no exchanges occur. [2]
- Average case: $O(n^2)$ - The algorithm always performs n-1 passes and involves n(n-1)/2 comparisons on average.
- Worst case: $O(n^2)$ - This occurs when the list is sorted in reverse order, requiring the maximum number of swaps.

Space complexity: $O(1)$ – Bubble sort is an in-place sorting algorithm, meaning it requires only a constant amount of additional memory space.

2.4.4 Variants and optimizations

- One common optimization is to introduce a flag that checks whether any swaps were made during a pass. If no swaps were made, the list is already sorted, and the algorithm can terminate early, improving the best-case time complexity to $O(n)$.
- Shaker sort is an extension of bubble sort, we will also discuss it later. It sorts in both directions in each pass through the list. By doing so, it can more quickly bring small values to the beginning and large values to the end, which helps reduce the number of passes needed compared to standard bubble sort.

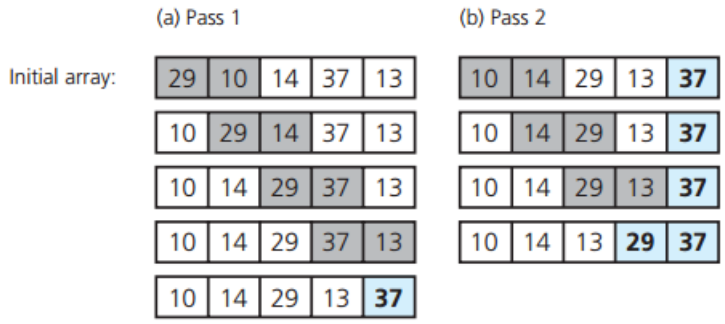


Figure 2.3: Bubble Sort Demo

2.5 Heap Sort

2.5.1 Core concept

Heap sort, as we have mentioned before, is an improvement of selection sort. It is comparison-based sorting technique based on heap data structure. We use heap to find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements.

2.5.2 Explanation

Heap definition (array-based representation) [3]

Heap is a collection of n elements $(a_0, a_1, \ldots, a_{n-1})$ in which every element (at position i) in the first half is greater than or equal to the elements at position $2i+1$ and $2i+2$.

$$\text{for every } i \ (0 \leq i \leq n/2 - 1)$$

$$a_i \geq a_{2i+1}$$

$$a_i \geq a_{2i+2}$$

Heap in above definition is called max-heap. (We also have min-heap structure)

Heap will reduce the time to find the minimum or maximum element when sorting the array which is an improvement of selection sort.
Main steps:

- **Build a (max or min) Heap**: Convert the input array into a heap.
- **Heapify**: Rearrange the heap to maintain the max-heap property.
- **Extract Max Element**: Swap the first element (maximum) with the last element and reduce the heap size by one. Heapify the root to maintain the max-heap property.
- **Repeat**: Repeat the extraction process until the heap is empty.

The most important step is build a heap. Here is pseudo code for one way to build a max-heap. [3]

Algorithm 1 Heap Construct

```
1. Initialize index = floor(n/2) - 1
2. While index >= 0
     a. Call heapRebuild(index, a, n)
     b. index = index - 1
```

Algorithm 2 Heap Rebuild

```
1. Initialize k = pos, v = a[k], isHeap = false
2. While not isHeap and 2k + 1 < n
     a. j = 2k + 1
     b. If j < n - 1
          i. If a[j] < a[j + 1]
               A. j = j + 1
     c. If a[k] >= a[j]
          i. isHeap = true
        else
          i. Swap a[k] and a[j]
          ii. k = j
```

2.5.3 Complexity analysis

Time complexity:

- Build max heap takes O(n) time and each of the n extraction takes O(log2 n)
- $O(nlog_2n)$ for all case

Space complexity: $O(1)$, if we don't consider the recursive stack space. If we consider the recursive stack space then, in the worst case Heap sort could make $O(n)$.

2.5.4 Variants and optimizations

Optimizing the heapify process can improve performance. For example, bottom-up heap construction can be more efficient than top-down heap construction.

2.6 Merge Sort

2.6.1 Core concept

Merge sort uses the powerful strategy of divide and conquer. The array is repeatedly split into smaller subarrays then merging them together. During the merging process, the elements are compared and sorted to create larger sorted subarrays repeat this process until all subarrays are merged back into the original array, resulting in a fully sorted array. By breaking the problem into smaller and then combining the sorted pieces, merge sort efficiently sorts the entire array.

2.6.2 Step-by-step Explanations

- Step 1: Divide the Array into Subarrays by calling Recursive to split the array into smaller subarrays until each subarray has only 1 or 2 elements. This is achieved by continuously dividing the array into two halves.
- Step 2: Start merging the subarrays back together. Create two temporary arrays to hold the elements of the subarrays being merged. Compare the elements from the two temporary arrays and place the smaller element back into the original array. Continue merging and comparing until all elements are sorted and merged back into the original array.

2.6.3 Complexity Analysis

Time complexity analysis:

- Best-case: $O(nlog(n))$
- Average-case: $O(nlog(n))$
- Worst-case: $O(nlog(n))$

Space complexity analysis: $O(n)$ because we have to create the many arrays with sum of these elements is equal to n elements of original array to merge.

An example of merge sort: [5]

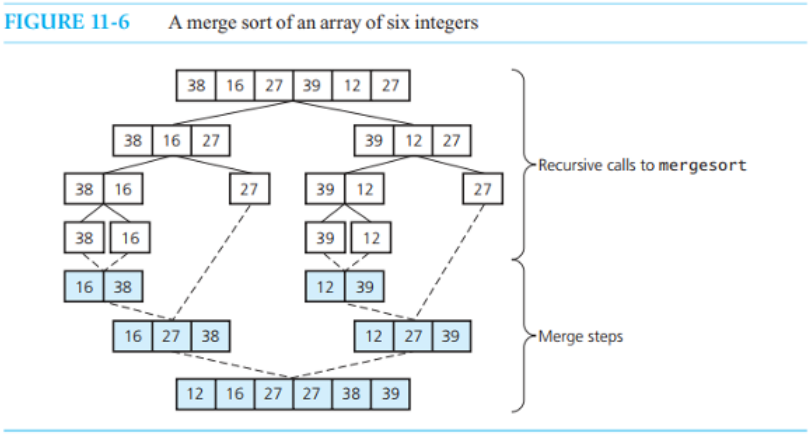


Figure 2.4: Merge Sort Demo

2.7 Quick Sort

2.7.1 Core concept

Quick sort 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. It one of best sorting algorithms using key comparisons.

2.7.2 Explanation

The algorithm consists of the following three steps: [3]

- Divide: Partition the list.
  - To partition the list, we first choose some element from the list for which we hope about half the elements will come before and half after. Call this element the pivot.
  - Then we partition the elements so that all those with values less than the pivot come in one sub-list and all those with greater values come in another.
- Recursion: Recursively sort the sub-lists separately.
- Conquer: Put the sorted sub-lists together.

One partition technique [3]

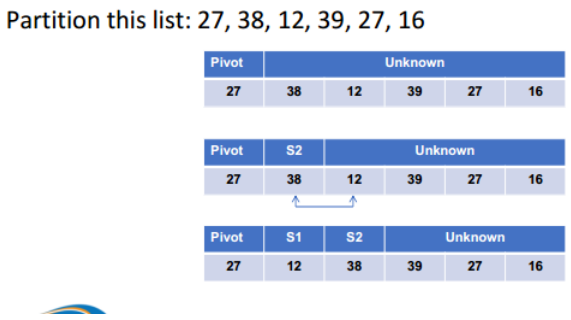


Figure 2.5: Partition technique

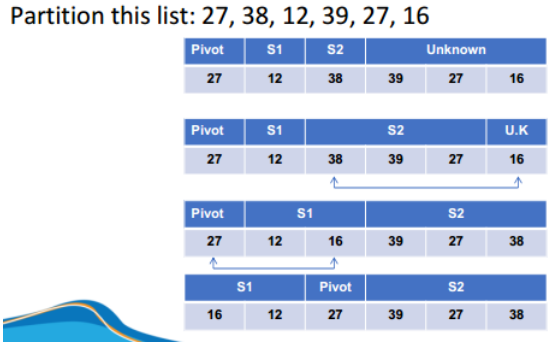


Figure 2.6: Partition technique

Another partition technique [2]

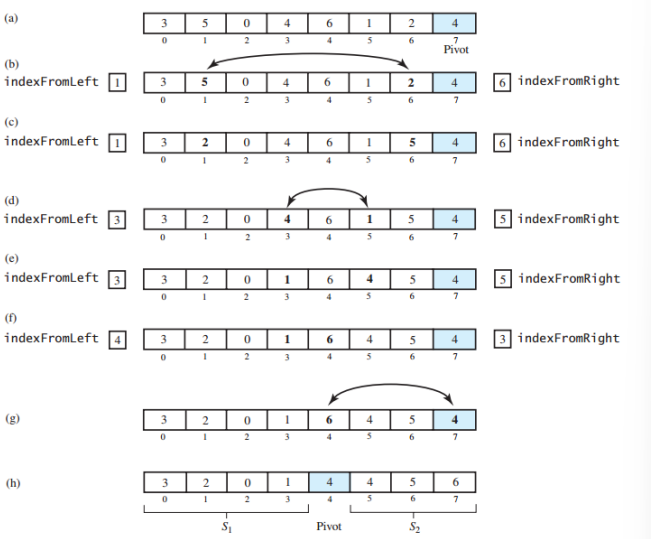


Figure 2.7: Partition technique 2

As the partition process is done recursively, it keeps on putting the pivot in its actual position in the sorted array. Repeatedly putting pivots in their actual position makes the array sorted.

2.7.3 Complexity analysis

Time complexity:

- Best case: O(n log2n) - occur when the pivot chosen at the each step divides the array into roughly equal halves, make balanced partitions, leading to efficient sorting.
- Average case: O(n log2n)
- Worst case: $O(n^2)$ - This rarely occurs when the pivot selection is poor, leading to unbalanced partitions (e.g., when the array is already sorted, and the first or last element is chosen as the pivot).

Space complexity: $O(1)$, if we don't consider the recursive stack space. If we consider the recursive stack space then, in the worst case quicksort could make $O(n)$. [6]

2.7.4 Variants and optimizations

- Median of three pivot selection: This pivot selection scheme assumes that the array has at least three entries. If you have only three entries, the pivot selection sorts them, so there is no need for the partition method or for a quick. [2]
- Tail call optimization: partition function is in-place, but we need extra space for recursive function calls. Optimizing the recursive calls to minimize the depth of the recursive stack. By always sorting the smaller subarray first and using tail recursion for the larger subarray, the stack depth can be reduced.

2.8 Radix Sort

2.8.1 Core Concepts

Radix sort is an algorithm that does not use comparisons to sorted the array. To be specific, treats each data item as a character string, groups the data items according to the rightmost character then put these groups into order with respect to this rightmost character, combine all the groups and move to the next left position. At the end, the sort operation will be in sorted. [3]

2.8.2 Step-by-step Explaination

- Step 1: find max value by calling getMax to determine the maximum number in the array.
- Step 2: Start with the rightmost significant number. Apply countSort to sort the array based on the current number. Move to the next significant digit by multiplying exp by 10. Repeat the process until all digits have been processed (when m / exp becomes 0).

2.8.3 Complexity Analysis

Time Complexity:

- Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping the keys by the individual digits which share the same significant position and value. It has a time complexity of $O(d \cdot (n + b))$, where d is the number of digits, n is the number of elements, and b is the base of the number system being used.
- In practical implementations, radix sort is often faster than other comparison-based sorting algorithms, such as quicksort or merge sort, for large datasets, especially when the keys have many digits. However, its time complexity grows linearly with the number of digits, and so it is not as efficient for small datasets. [7]

Space complexity: Radix sort also has a space complexity of $O(n + b)$, where n is the number of elements and b is the base of the number system. This space complexity comes from the need to create buckets for each digit value and to copy the elements back to the original array after each digit has been sorted.
An example of radix sort: [5]

FIGURE 11-14 A radix sort of eight integers

0123, 2154, 0222, 0004, 0283, 1560, 1061, 2150	Original integers
(1560, 2150) (1061) (0222) (0123, 0283) (2154, 0004)	Grouped by fourth digit
1560, 2150, 1061, 0222, 0123, 0283, 2154, 0004	Combined
(0004) (0222, 0123) (2150, 2154) (1560, 1061) (0283)	Grouped by third digit
0004, 0222, 0123, 2150, 2154, 1560, 1061, 0283	Combined
(0004, 1061) (0123, 2150, 2154) (0222, 0283) (1560)	Grouped by second digit
0004, 1061, 0123, 2150, 2154, 0222, 0283, 1560	Combined
(0004, 0123, 0222, 0283) (1061, 1560) (2150, 2154)	Grouped by first digit
0004, 0123, 0222, 0283, 1061, 1560, 2150, 2154	Combined (sorted)

Figure 2.8: Radix Sort Demo

2.9 Counting Sort

2.9.1 Core Concepts

Counting sort is an efficient algorithm for sorting integers. It works by counting the number of occurrences of each distinct value in the input array using a separate count array. These counts are then used to place the elements back into the original array in sorted order, ranging from the smallest to the largest value. This method ensures that the array is sorted correctly and efficiently. [8]

2.9.2 Step-by-step Explanations

- Step 1: find max value of the array
- Step 2: create an array with max value of elements to count appearances
- Step 3: iterate through the original array a. For each element in a, increment the corresponding index in the count array b. This effectively counts the occurrences of each element.
- Step 4: assign the value back to original array

2.9.3 Complexity Analysis

Time Complexity:

- Best-case: $O(n + k)$
- Average-case: $O(n + k)$
- Worst-case: $O(n + k)$
- where n is the number of elements in the array, and k is the range of the input.

Space Complexity: $O(n + k)$ because it requires additional space for the count array and the output array.

2.9.4 Variants and Optimizations

Negative number: if there are negative numbers in input array and there is no negative index so we can plus to all elements of the array by the smallest value to make sure that smallest is 0 then running counting and the final sorted array, we minus all elements by the value that we plus in first step.

2.10 Binary Insertion Sort

2.10.1 Core concept

Binary Insertion sort is a variation of insertion sort. the core idea is similar to insertion sort, but instead of using linear search to find the position where an element should be inserted, this algorithm uses binary search. Thus, it reduce the comparative value of inserting a single element. [9]

2.10.2 Explanation

The algorithm is similar to insertion sort algorithm, but here we use binary search to find the position:

- **Initialization:** Start with the second element (index 1) of the array, as the first element is considered sorted.
- **Binary Search for Position:** Use binary search to determine the correct position in the sorted portion of the array for the current element.
- **Shift Elements:** Shift elements to the right to make room for the current element.
- **Insert Element:** Insert the current element into its correct position.
- **Repeat:** Repeat the process for each element in the array until the entire array is sorted

Step-by-step:

- Lets use the array as an example 29, 10, 14, 37, 13
- Consider 29 is in sorted part, start from 10, we use binary search to find the position in sorted part to insert 10. 10 smaller than 29 so the position is 0. Then we shift and insert 10.
- After first insertion 10, 29, 14, 37, 13
- Continue to find the location for 14 which is between 10 and 29. We shift and insert 14 to the correct position.
- After second insertion 10, 14, 29, 37, 13
- We continue to the same technique for 37 and 13. Finally, we get a sorted list
- After third and fourth insertions 10, 13, 14, 29, 37

2.10.3 Complexity analysis

Time complexity: Consider the array $a[]$ sorted in increasing order.
Iteration 1: requires 1 comparison
Iteration 2: requires $\log 2 + 1$ comparisons
...
Iteration $n-1$**:** requires $\log(n-1) + 1$ comparisons
In all iterations, since the array is already sorted in increasing order, the elements do not need to be moved.

$$f(n) = 1 + \log 2 + 1 + \log 3 + 1 + \ldots + \log(n-1) + 1 = n - 1 + \sum_{i=2}^{n-1} \log i$$

Consider $\sum_{i=2}^{n-1} \log i \approx \sum_{i=1}^{n} \log i$, then:

$$\sum_{i=n/2}^{n} \log \frac{n}{2} \leq \sum_{i=1}^{n} \log i \leq \sum_{i=1}^{n} \log n \Rightarrow \frac{n}{2} \log \frac{n}{2} \leq f(n) \leq n \log n$$

Thus, $f(n) \in \Omega(n \log n)$ and $f(n) \in O(n \log n)$, therefore $f(n) \in \Theta(n \log n)$.
Therefore, if we only apply simple binary search, in the best case, the complexity of BinaryInsertionSort is $O(n \log n)$. This can be improved by comparing $a[i]$ and $a[i-1]$ first when performing the binary search, which can reduce it to $O(n)$.

- Best Case: $O(nlogn)$ – Binary search reduces the number of comparisons, but shifting elements still takes linear time.
- Average Case: $O(n^2)$ – Despite using binary search, the time complexity remains quadratic due to the shifting of elements.
- Worst Case: $O(n^2)$ – Similar to the average case, the need to shift elements results in quadratic time complexity.

Space complexity: O(1) - binary insertion sort is similar to insertion sort. It is an in-place sorting algorithm, meaning it requires only a constant amount of additional memory space.

2.10.4 Variants and optimizations

Tim Sort is a hybrid sorting algorithm derived from merge sort and insertion sort. It was designed to perform well on many kinds of real-world data. [10]

2.11 Shaker Sort

2.11.1 Core concept

Shaker sort, also known as cocktail sort or bidirectional bubble sort, is a variation of bubble sort that sorts the array in both directions on each pass through the list. This means it traverses the array from left to right and then from right to left, ensuring that the largest elements "bubble" to the end and the smallest elements "bubble" to the beginning of the array.

2.11.2 Explanation

The algorithm is broken up into 2 stages: [11]

- The first stage loops through the array from left to right, just like the Bubble Sort. During the loop, adjacent items are compared and if the value on the left is greater than the value on the right, then values are swapped. At the end of the first iteration, the largest number will reside at the end of the array.
- The second stage loops through the array in opposite direction- starting 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.

Example for the first pass [11]
Let us consider an example array: (5 1 4 2 8 0 2)
First Forward Pass:

```
(5 1 4 2 8 0 2) → (1 5 4 2 8 0 2), Swap since 5 > 1
(1 5 4 2 8 0 2) → (1 4 5 2 8 0 2), Swap since 5 > 4
(1 4 5 2 8 0 2) → (1 4 2 5 8 0 2), Swap since 5 > 2
(1 4 2 5 8 0 2) → (1 4 2 5 8 0 2)
(1 4 2 5 8 0 2) → (1 4 2 5 0 8 2), Swap since 8 > 0
(1 4 2 5 0 8 2) → (1 4 2 5 0 2 8), Swap since 8 > 2
```

After the first forward pass, the greatest element of the array will be present at the last index of the array.

First Backward Pass:

```
(1 4 2 5 0 2 8) → (1 4 2 5 0 2 8)
(1 4 2 5 0 2 8) → (1 4 2 0 5 2 8), Swap since 5 > 0
(1 4 2 0 5 2 8) → (1 4 0 2 5 2 8), Swap since 2 > 0
(1 4 0 2 5 2 8) → (1 0 4 2 5 2 8), Swap since 4 > 0
(1 0 4 2 5 2 8) → (0 1 4 2 5 2 8), Swap since 1 > 0
```

After the first backward pass, the smallest element of the array will be present at the first index of the array.

2.11.3 Complexity analysis

Time complexity:

- Best Case: $O(n)$ – This occurs when the array is already sorted. Only one full pass through the array is needed.
- Average Case: $O(n^2)$ – Each element needs to be compared with many other elements in the array.
- Worst Case: $O(n^2)$ – Similar to the average case, the worst-case scenario involves multiple passes through the array with numerous swaps.

Space complexity: $O(1)$ – Shaker sort is an in-place sorting algorithm, meaning it requires only a constant amount of additional memory space.

2.12 Flash Sort

2.12.1 Core Concepts

Flash sort is sorting that divide elements in to different buckets which the gap of value is smallest and make sure that the largest value in previous bucket is smaller that lowest value in next bucket. Then we use basic sorting algorithms to sort buckets, often use insertion sort because it work well with nearly sorted array. [12]

2.12.2 Step-by-step Explanations

First, calculate the number of buckets by multiplying the size of the input array by 0.45. Then, create an array to count the number of elements in each bucket. Iterate through the input array, assigning each element to the appropriate bucket. Next, adjust the bucket counts to reflect the final positions of elements. Reorder the elements so each bucket contains the correct values. Finally, apply insertion sort within each bucket to sort the elements, resulting in a fully sorted array.

2.12.3 Complexity Analysis

Time Complexity:

- Best-case: $O(n)$
- Average-case: $O(n + k)$
- Worst-case: $O(n + k)$
- Where n is the number of elements, and k is the number of buckets.

Space Complexity: $O(n + k)$ because it requires additional space for the count array and the class boundaries.

2.12.4 Variants and Optimizations

There are different way to count the number of buckets based on the different data so just test and find suitable buckets to improve performance, sometimes number of buckets equal to $0.45 * nor 0.42 * nork * n$. We can use quick-sort instead of insertion-sort if the array is larger.

3 Survey and Evaluation

3.1 Randomize Data

3.1.1 Time

Execution time unit: milliseconds

Algorithm	10,000	30,000	50,000	100,000	300,000	500,000
selection-sort	90.9034	819.165	2144.86	8596.38	76158.2	210088
insertion-sort	5.4657	49.4609	138.977	547.006	4937.82	13827.8
shell-sort	20.9348	204.922	550.198	2260.1	18522.9	53320.1
bubble-sort	97.1671	1116.17	3208.09	13295.4	121053	339611
heap-sort	0.909	3.1648	5.3967	11.5898	39.6501	68.36
merge-sort	3.2503	9.3788	15.4051	28.9049	87.508	146.323
quick-sort	0.5708	1.7461	3.1067	6.3324	20.1092	34.9733
radix-sort	1.8813	0.5193	0.7778	1.8149	4.8387	8.4334
counting-sort	0.07	0.1945	0.2539	0.4312	0.902	1.3383
binary-insertion-sort	4.8785	37.591	102.974	410.325	3620.11	10357.5
shaker-sort	85.7846	842.283	2380.6	9535.75	86159.8	239150
flash-sort	0.1696	0.6379	0.9506	1.8552	7.8633	16.1854

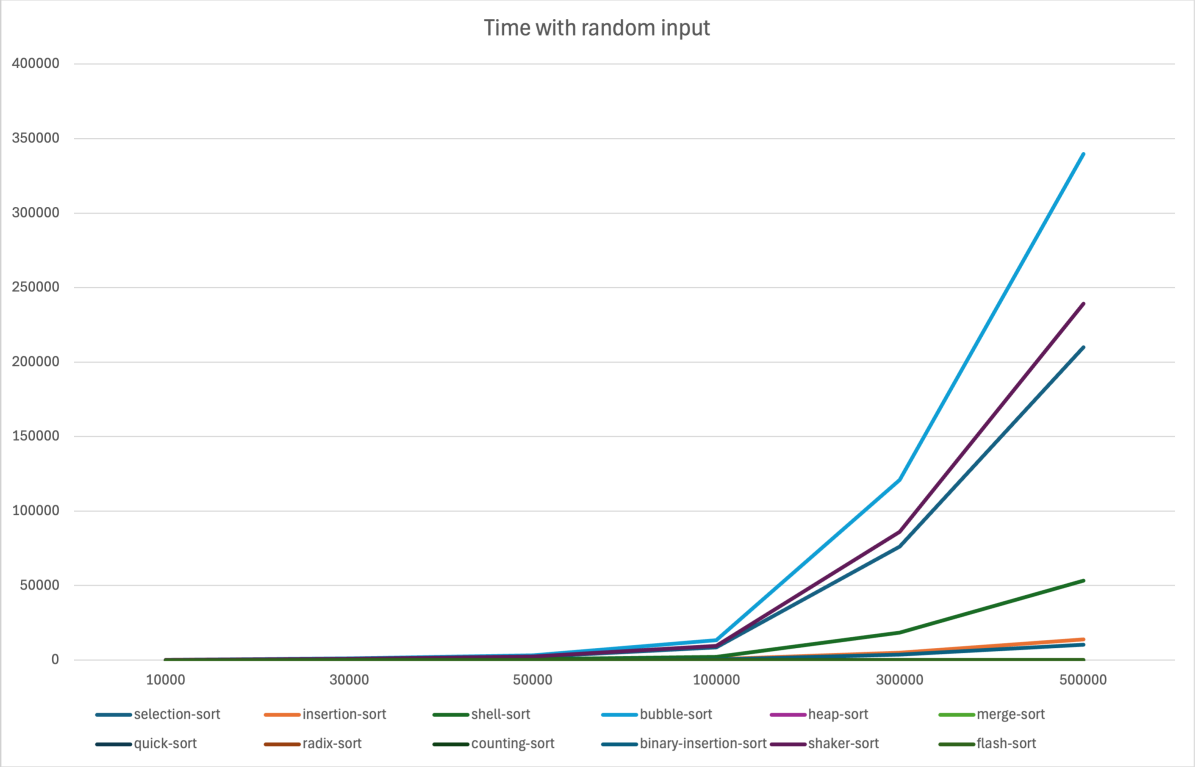


Figure 3.1: Execution time for Randomize data

Comments on the running time of sorting algorithms with random data

1. **Fastest algorithms:**
   - `Counting Sort`, *Quick Sort*, *Heap Sort*, and *Flash Sort* are the fastest algorithms as the data size increases. Notably, *Quick Sort* and *Flash Sort* have very low running times compared to other algorithms, even with large datasets like 500,000 elements.

2. **Slowest algorithms:**
   - *Bubble Sort*, *Selection Sort*, and *Insertion Sort* have the longest running times. *Bubble Sort* and

Selection Sort have rapidly increasing running times, reaching hundreds of thousands of seconds with datasets of 500,000 elements.

3. **Other algorithms:**
   - *Merge Sort*, *Shell Sort*, and *Shaker Sort* have average running times, neither too fast nor too slow.

3.1.2 Comparison

Algorithm	10,000	30,000	50,000	100,000	300,000	500,000
selection-sort	100020001	900060001	2500100001	10000200001	90000600001	250000000001
insertion-sort	49650481	448332712	1248140892	4990403035	45011022554	124964111984
shell-sort	136486703	1228074234	3411144570	13643867417	1.2279e+11	3.41082e+11
bubble-sort	100009999	900029999	2500049999	10000099999	90000299999	2.5e+11
heap-sort	638069	2149637	3772566	8044375	26494137	45968203
merge-sort	583723	1937626	3383060	7166134	23383144	40382087
quick-sort	364891	1086464	1921044	4144209	14922110	27756867
radix-sort	140051	510064	850064	1700064	510064	850064
counting-sort	60001	180001	265539	465539	1265539	2065539
binary-insertion-sort	25432305	226953151	629328887	2511144485	22498338748	62537270911
shaker-sort	66785983	599351162	1667586501	6658782743	60022078477	1.66739e+11
flash-sort	92801	285860	496012	892630	2755320	4583014

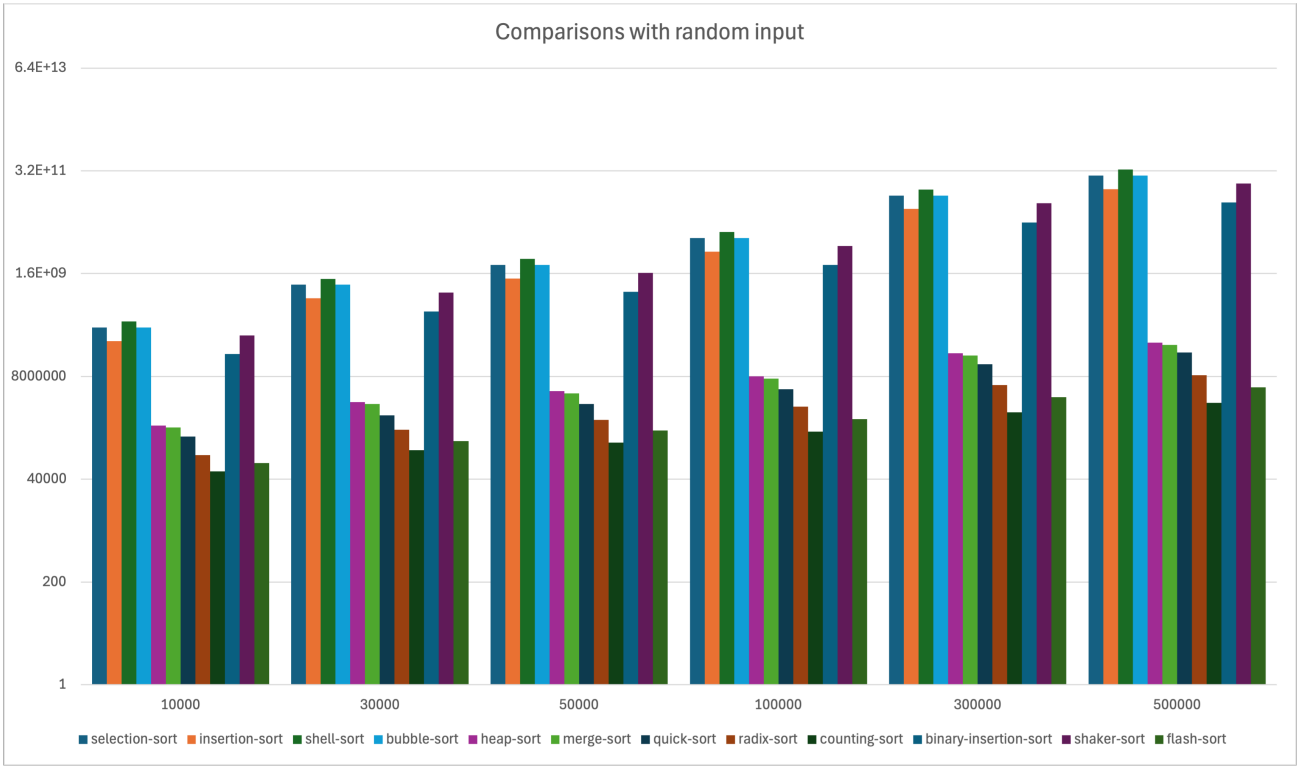


Figure 3.2: Number of comparisons of the algorithm for Randomize data

Comments on the number of comparisons of sorting algorithms with random data

1. **Fewest comparisons:**
   - *Counting Sort*, *Flash Sort*, and *Radix Sort* have the fewest comparisons, especially *Counting Sort*, which has very low comparison counts even with large datasets.

2. **Most comparisons:**
   - *Bubble Sort*, *Selection Sort*, and *Insertion Sort* have very high comparison counts, especially *Bubble Sort* and *Selection Sort*, which reach billions of comparisons with datasets of 500,000 elements.

3. **Other algorithms:**
   - *Quick Sort*, *Heap Sort*, and *Merge Sort* have relatively low comparison counts, making them more efficient compared to slower sorting algorithms.

3.1.3 Overall

Overall comments on algorithms based on all data and size

1. **Overall fastest algorithms:**
   - *Quick Sort*, *Heap Sort*, and *Flash Sort* are the overall fastest algorithms. Among them, *Quick Sort* is most commonly used due to its efficiency in handling large datasets.

2. **Overall slowest algorithms:**
   - *Bubble Sort*, *Selection Sort*, and *Insertion Sort* are the overall slowest algorithms. They are suitable only for very small datasets or educational purposes.

3. **Stable and unstable algorithms:**
   - Algorithms like *Merge Sort* and *Insertion Sort* are stable, meaning they maintain the relative order of equal elements.
   - Algorithms like *Quick Sort* and *Heap Sort* are unstable, meaning they do not maintain the relative order of equal elements.

3.2 Sorted Data

3.2.1 Time

Execution time unit: milliseconds

Algorithm	**10000**	**30000**	**50000**	**100000**	**300000**	**500000**
Selection Sort	83.5349	762.314	2093.73	8438.43	75874.9	210401
Insertion Sort	0.0043	0.013	0.0228	0.0452	0.1466	0.2558
Shell Sort	15.9199	166.915	454.441	1746.54	16108.9	72609.4
Bubble Sort	13.4474	123.507	329.481	1383.44	12187	34030.9
Heap Sort	0.691	2.2718	3.8386	8.0435	26.315	44.2006
Merge Sort	2.336	7.0626	11.8658	23.7198	73.8943	119.809
Quick Sort	78.4909	699.612	1986.74	7702.21	69509.3	198334
Radix Sort	0.1545	0.4646	0.943	1.5853	6.1696	10.6871
Counting Sort	0.016	0.0444	0.0809	0.208	0.7713	1.0371
Binary Insertion Sort	0.242	0.8476	1.3542	2.9678	10.6128	17.5126
Shaker Sort	0.0036	0.0094	0.0166	0.0418	0.1131	0.2715
Flash Sort	0.0948	0.2717	0.6822	1.1887	3.7231	5.9786

Comments on the running time of sorting algorithms with sorted data

1. **Fastest Algorithms**
   - Insertion Sort: Extremely fast across all data sizes, maintaining very low execution times, which can be attributed to the sorted input allowing it to perform minimal operations.
   - Counting Sort and Radix Sort: Show low execution times, consistent with their linear or near-linear time complexity.
   - Flash Sort: Also performs well across different sizes, indicating efficiency in handling sorted data.

2. **Slowest Algorithms**
   - Selection Sort: Performs poorly, with execution times increasing significantly with larger data sizes. This is due to its $O(n^2)$ complexity.
   - Shell Sort and Bubble Sort: Have relatively high execution times, especially for larger data sizes, though they perform better than selection sort.

3. **Moderate Performance**
   - Merge Sort, Heap Sort, Quick Sort: These algorithms show moderate execution times. Quick Sort's performance is likely impacted by pivot selection, which can lead to worse performance even on sorted input.

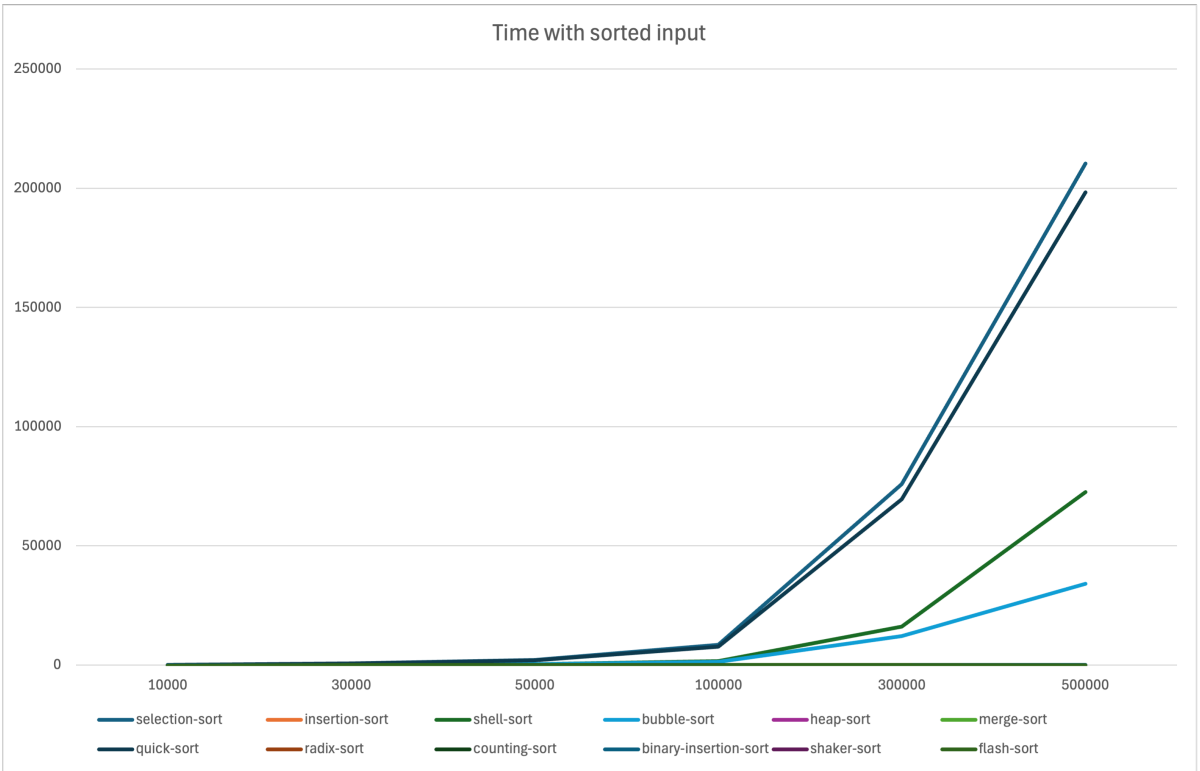


Figure 3.3: Execution time for Sorted data

3.2.2 Comparison

Algorithm	**10000**	**30000**	**50000**	**100000**	**300000**	**500000**
Selection Sort	100020001	900060001	2500100001	10000200001	90000600001	2.50001E+11
Insertion Sort	29998	89998	149998	299998	899998	1499998
Shell Sort	136486703	1228074234	3411144570	13643867417	1.2279E+11	3.41082E+11
Bubble Sort	100009999	900029999	2500049999	10000099999	90000299999	2.5E+11
Heap Sort	670329	2236648	3925351	8365080	27413230	47404886
Merge Sort	475242	1559914	2722826	5745658	18645946	32017850
Quick Sort	100019998	900059998	2500099998	10000199998	90000599998	2.50001E+11
Radix Sort	140051	510064	850064	1700064	6000077	10000777
Counting Sort	60003	180003	300003	600003	1800003	3000003
Binary Insertion Sort	370852	1251700	2203396	4706788	15527140	26927140
Shaker Sort	20002	60002	100002	200002	600002	1000002
Flash Sort	119000	357000	595000	1190000	3570000	5950000

Comments on the number of comparisons of sorting algorithms with sorted data

1. **Fewest Comparisons**
   - Counting Sort: Performs the fewest comparisons as it is a non-comparison-based sort.
   - Insertion Sort: Very few comparisons are needed due to the already sorted input, similar to its execution time performance.
   - Flash Sort: Also shows low comparisons, consistent with its efficient performance.

2. **Most Comparisons**
   - Shell Sort: Has an extremely high number of comparisons, especially at larger data sizes.
   - Bubble Sort: Also shows a high number of comparisons, though not as much as shell sort.
   - Selection Sort and Quick Sort: Both have high numbers of comparisons due to their $O(n^2)$ and $O(n \log n)$ complexities, respectively.

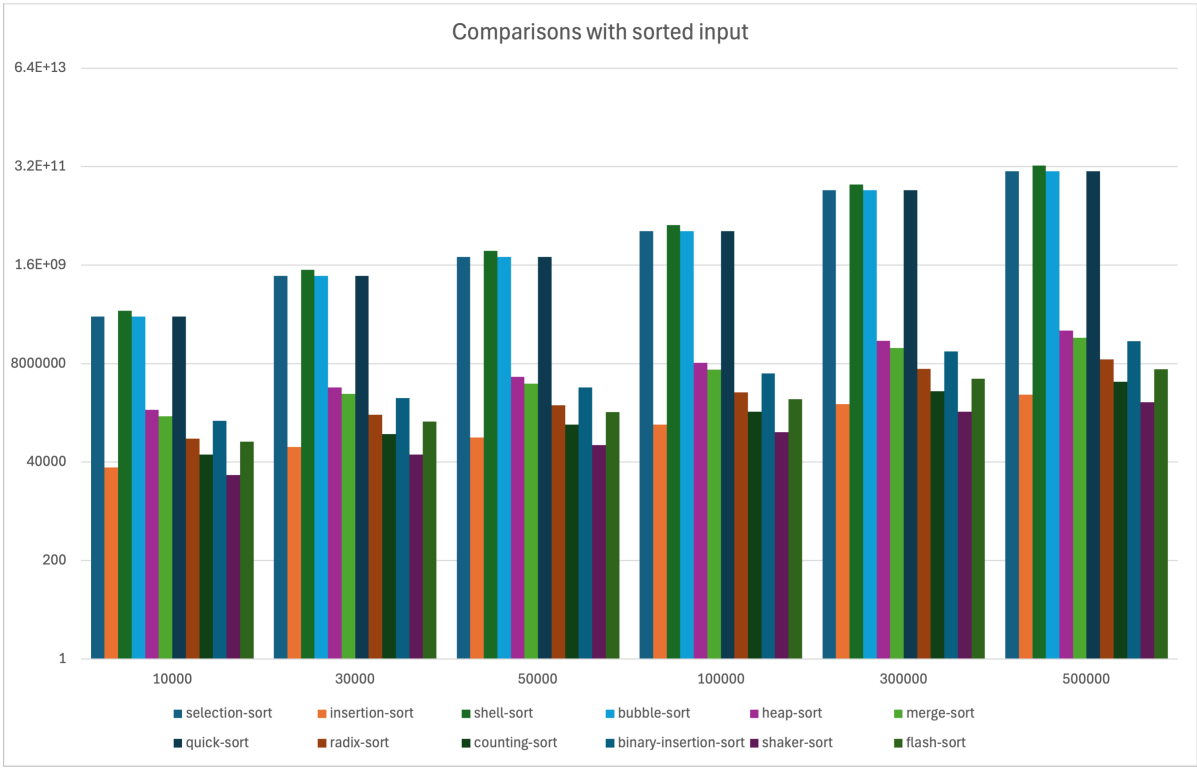


Figure 3.4: Number of comparisons of the algorithm for Sorted data

3.2.3 Overall

Overall comments on algorithms based on all data and size

1. **Data Order and Data Size**
   - For sorted data, insertion sort, counting sort, and flash sort are the best performers in terms of execution time and number of comparisons.
   - Selection sort, shell sort, and bubble sort are consistently the slowest and have the highest number of comparisons.
   - Merge sort, heap sort, and quick sort offer a balance but do not outperform the linear or near-linear algorithms for sorted data.

2. **Stability:**
   - Stable Algorithms: Insertion sort, merge sort, and bubble sort are stable sorting algorithms. Insertion sort performs the best with sorted input among the stable sorts.
   - Unstable Algorithms: Quick sort, heap sort, and shell sort are unstable. Despite quick sort and heap sort being faster than shell sort in most cases, they are still outperformed by the linear time algorithms on sorted data.

3.3 Reverse Data

3.3.1 Time

Comments on the running time of sorting algorithms with reverse data

1. **Fastest Algorithms:**
   - *Heap Sort*, *Radix Sort*, and *Counting Sort* again perform the fastest across all input sizes. Their execution times remain very low even for larger input sizes, similar to their performance on sorted input.

2. **Slowest Algorithms:**
   - *Bubble Sort*, *Shaker Sort*, and *Quick Sort* exhibit very high execution times, especially as the input size increases. For instance, bubble sort's time grows from 76.9914 seconds at 10000 elements to 195572 seconds at 500000 elements.

3. **Consistent Performers:**
   - *Merge Sort* and *Shell Sort* show relatively stable and moderate execution times, indicating reliable performance across different input sizes.

Execution time unit: milliseconds

Algorithm	**10000**	**30000**	**50000**	**100000**	**300000**	**500000**
Selection Sort	85.285	764.617	2085.39	8471.83	75944.1	210673
Insertion Sort	11.1563	100.834	276.323	1090.38	9847.49	28014.6
Shell Sort	21.0003	203.454	527.093	2115.4	18846.1	55537.9
Bubble Sort	76.9914	692.504	1928.01	7667.36	69228.6	195572
Heap Sort	0.6743	2.2238	4.1879	8.3122	26.5599	45.5919
Merge Sort	2.5089	7.4184	12.1534	23.8237	73.346	123.387
Quick Sort	44.9166	400.345	1098.21	4412.2	39600.2	112705
Radix Sort	0.133	0.4967	0.8283	1.5792	5.9777	10.3353
Counting Sort	0.016	0.0421	0.0752	0.1617	0.6948	1.2085
Binary Insertion Sort	7.9368	78.3401	204.501	795.454	7076.07	21493.8
Shaker Sort	81.9425	730.12	2068.47	8181.6	73751.9	204937
Flash Sort	0.1018	0.317	0.6843	1.3254	3.8055	6.2582

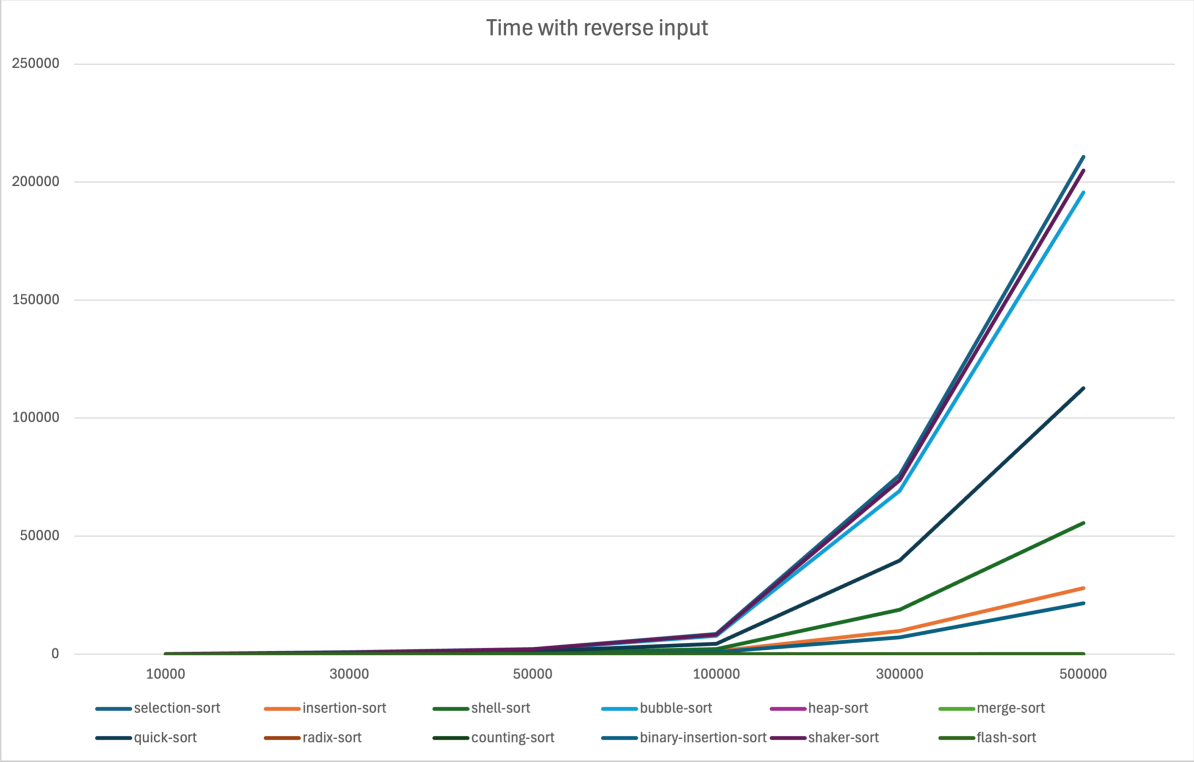


Figure 3.5: Execution time for Reverse data

3.3.2 Comparison

Comments on the number of comparisons of sorting algorithms with reverse data

1. **Fewest Comparisons:**
   - *Counting Sort* and *Radix Sort* consistently perform the fewest comparisons, highlighting their efficiency in this metric for reverse sorted inputs as well.

2. **Most Comparisons:**
   - *Selection Sort*, *Insertion Sort*, *Bubble Sort*, and *Quick Sort* show an extremely high number of comparisons, reaching up to the order of 10^{11} for larger input sizes, indicating significant inefficiency in terms of comparisons.

3. **Consistent Performers:**
   - *Heap Sort*, *Merge Sort*, and *Binary Insertion Sort* maintain a moderate and steadily increasing number of comparisons, showing balanced performance.

Algorithm	**10000**	**30000**	**50000**	**100000**	**300000**	**500000**
Selection Sort	100020001	900060001	2500100001	10000200001	90000600001	2.50001E+11
Insertion Sort	100009999	900029999	2500049999	10000099999	90000299999	2.5E+11
Shell Sort	136486703	1228074234	3411144570	13643867417	1.2279E+11	3.41082E+11
Bubble Sort	100009999	900029999	2500049999	10000099999	90000299999	2.5E+11
Heap Sort	606771	2063324	3612724	7718943	25569379	44483348
Merge Sort	476441	1573465	2733945	5767897	18708313	32336409
Quick Sort	100019998	900059998	2500099998	10000199998	90000599998	2.50001E+11
Radix Sort	140051	510064	850064	1700064	6000077	10000077
Counting Sort	60003	180003	300003	600003	1800003	3000003
Binary Insertion Sort	50348179	451187593	1252080140	5004460231	45014870410	1.25026E+11
Shaker Sort	100005001	900015001	2500025001	10000050001	90000150001	2.5E+11
Flash Sort	103753	311253	518753	1037503	3112503	5187503

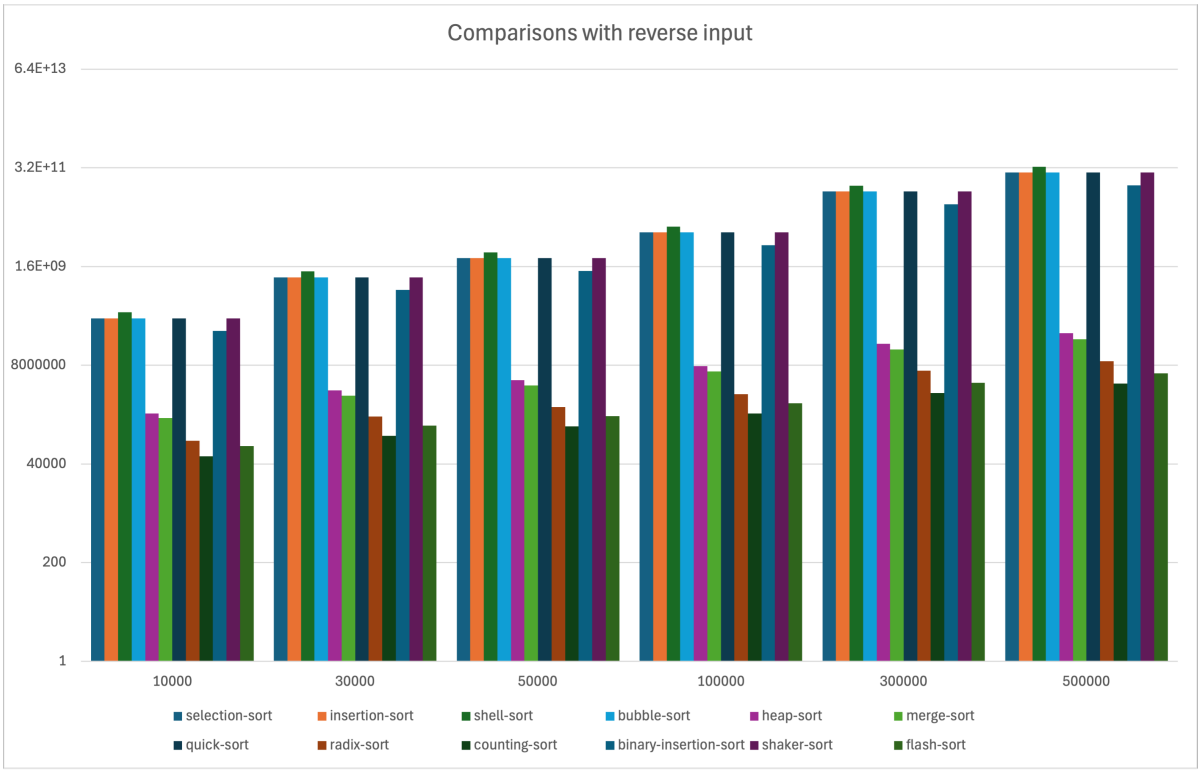


Figure 3.6: Number of comparisons of the algorithm for Reverse data

3.3.3 Overall

1. **Stable Algorithms:**
   - *Heap Sort*, *Radix Sort*, and *Merge Sort* are stable across both sorted and reverse sorted data in terms of execution time and number of comparisons. These algorithms are generally reliable and efficient.

2. **Unstable Algorithms:**
   - *Quick Sort*, *Selection Sort*, *Bubble Sort*, and *Shaker Sort* are unstable, especially with a significant increase in execution time and number of comparisons as the input size grows. These algorithms tend to be inefficient for larger datasets and in different data orders.

3. **Fastest Algorithms Overall:**
   - *Insertion Sort*, *Counting Sort*, and *Binary Insertion Sort* consistently show the fastest execution times, particularly for smaller input sizes. Their performance is highly efficient for both sorted and reverse sorted inputs.

4. **Slowest Algorithms Overall:**
   - *Selection Sort*, *Bubble Sort*, and *Shaker Sort* are the slowest overall, showing significantly higher execution times and a very high number of comparisons for larger datasets and different data orders.

3.4 Nearly Sorted Data

3.4.1 Time

Execution time unit: milliseconds

Algorithm	10000	30000	50000	100000	300000	500000
Selection Sort	83.898	754.469	2097.2	8461.66	76561.9	210139
Insertion Sort	0.02	0.0912	0.0799	0.1037	0.1879	0.3103
Shell Sort	20.4246	195.299	484.834	1795.67	15977.1	49089.4
Bubble Sort	14.3526	124.481	347.676	1339.36	12296.1	34055
Heap Sort	0.6889	2.3363	3.8533	8.278	29.1528	43.6957
Merge Sort	2.497	8.2457	12.1077	27.873	72.59	120.094
Quick Sort	19.2834	152.439	1281.55	7068.34	68668.4	197042
Radix Sort	0.1272	0.5591	0.8446	1.5646	5.8751	10.27
Counting Sort	0.0155	0.043	0.0714	0.1567	0.7502	1.4829
Binary Insertion Sort	0.271	0.9111	1.4968	3.1023	9.2867	16.3849
Shaker Sort	0.1591	0.3401	0.2698	0.327	0.3967	0.5661
Flash Sort	0.0961	0.3437	0.6098	0.992	4.0775	6.7901

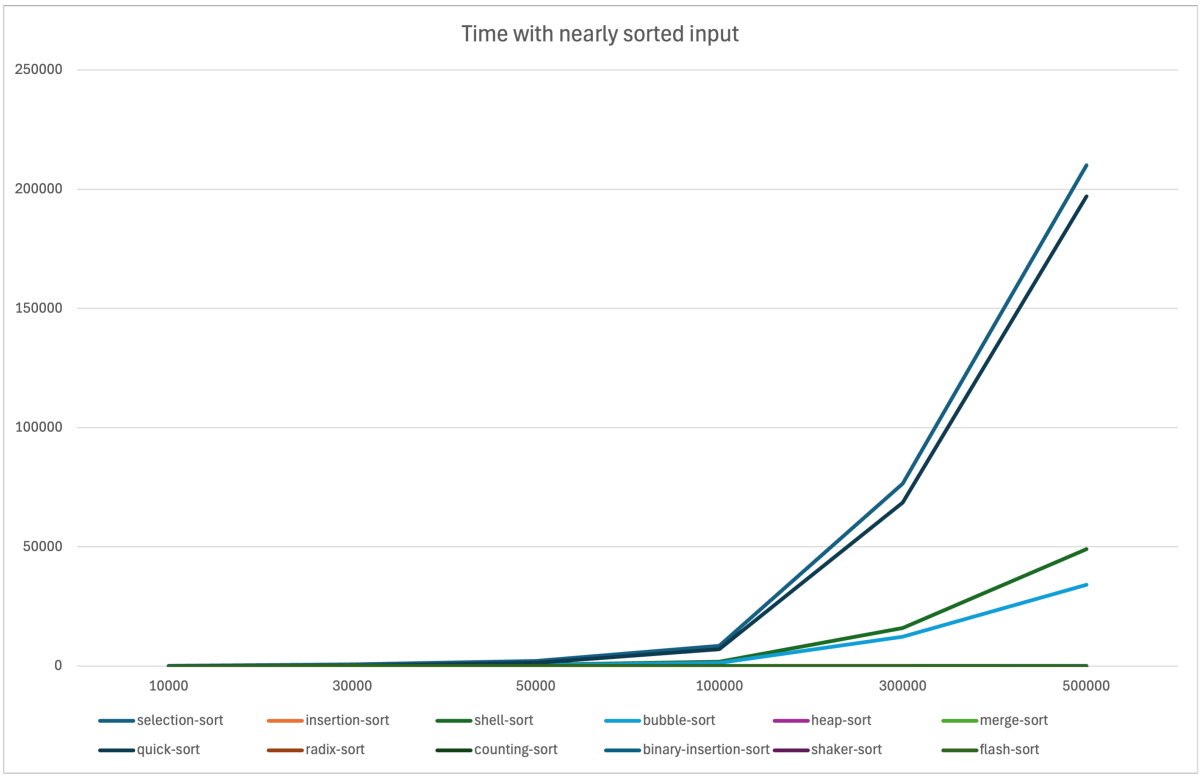


Figure 3.7: Execution time for Nearly Sorted data

Comments on the running time of sorting algorithms with nearly sorted data

1. **Fastest Algorithm:**
   - For small data sizes (10,000 - 50,000), *Insertion Sort* has the fastest execution time.
   - As data size increases (100,000 - 500,000), *Counting Sort* and *Flash Sort* become the most efficient in terms of execution time.
2. **Slowest Algorithm:** - *Selection Sort* and *Bubble Sort* are the slowest algorithms for all data sizes.
3. **General Comments:** - Algorithms like *Quick Sort*, *Merge Sort*, and *Heap Sort* have moderate execution times, better than *Selection Sort* and *Bubble Sort*, but slower than *Counting Sort* and *Flash Sort*.

3.4.2 Comparison

Algorithm	10000	30000	50000	100000	300000	500000
Selection Sort	100020001	900060001	2500100001	10000200001	90000600001	2.50001E+11
Insertion Sort	147654	744478	651114	801114	1401114	2069258
Shell Sort	136486703	1228074234	3411144570	13643867417	1.2279E+11	3.41082E+11
Bubble Sort	100009999	900029999	2500049999	10000099999	90000299999	2.5E+11
Heap Sort	669993	2236691	3925392	8364913	27413230	47404908
Merge Sort	507409	1666255	2833852	5856684	18744620	32121800
Quick Sort	29623224	252918102	1706786888	9235419190	89219734904	2.49335E+11
Radix Sort	140051	510064	850064	1700064	6000077	10000077
Counting Sort	60003	180003	300003	600003	1800003	3000003
Binary Insertion Sort	437075	1467163	2451861	4879730	15700082	27071229
Shaker Sort	183347	474192	465899	565899	965899	1475760
Flash Sort	118975	356977	594976	1189973	3569971	5949978

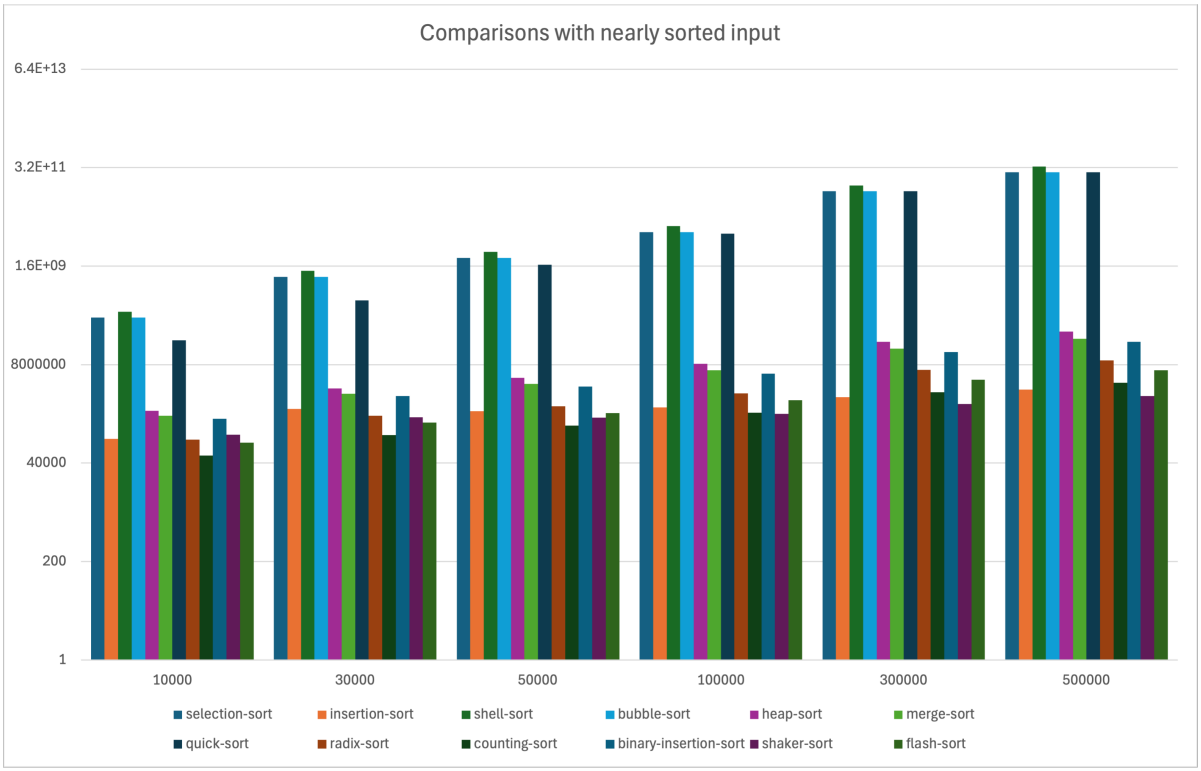


Figure 3.8: Number of comparisons of the algorithm for Nearly Sorted data

Comments on the number of comparisons of sorting algorithms with nearly sorted data

1. **Algorithm with Fewest Comparisons:**
   - *Counting Sort* and *Radix Sort* have the fewest comparisons for all data sizes.

2. **Algorithm with Most Comparisons:** - *Selection Sort*, *Bubble Sort*, and *Shell Sort* have the most comparisons for all data sizes.

3. **General Comments:**
   - Algorithms like *Merge Sort* and *Heap Sort* have relatively fewer comparisons compared to *Selection Sort* and *Bubble Sort*.

3.4.3 Overall

1. **By Data Order:**

   - *Insertion Sort* performs best when the data is nearly sorted.
   - *Selection Sort* and *Bubble Sort* remain slow and inefficient with nearly sorted data.

2. **By Data Size:**

   - For small data sizes, *Insertion Sort* and *Counting Sort* have the best execution times.
   - For large data sizes, *Counting Sort* and *Flash Sort* excel in execution time.
   - *Quick Sort* and *Merge Sort* can be considered good choices for medium to large data sizes due to their balance between execution time and comparison count.

3. **Grouping Stable/Unstable Algorithms:**

   - **Stable:**
     - *Insertion Sort*, *Merge Sort*, and *Binary Insertion Sort* are stable algorithms, ensuring the order of equal elements is preserved.
   - **Unstable:**
     - *Quick Sort*, *Heap Sort*, and *Shell Sort* are unstable algorithms and may change the order of equal elements.

4 Project Organization and Programming Note

All source codes are located in "SOURCE" folder.

4.1 Project organization

1. Header files
   - SortWithTime.h: This file contains the declaration of all original sorting algorithms which are used to measure the time of each algorithm.
   - SortWithComapre.h: This file contains the declaration of all sorting algorithms which are used to counting the number of comparisons of each algorithm.
   - CommandLine.h: This file contains the declarations of below things:
     – Declare all commands which can be used when executing the file exe
     – Declare necessary functions to run the commands.
   - DataGenerator.h: This file contains the below things:
     – Declare the functions to generate data in different orders and different sizes.
     – Declare the functions to read files to get data and write data into files.
     – Declare the function to duplicate an array.
     – Declare the functions to measure the time and counting the comparison of given algorithm.
     – The given template and given function for the template.
   - Lib.h: This file contains all the necessary libraries.
2. .cpp files
   - The cpp files in folder contains all definitions of the functions that are declared in the header files. "main.cpp" contains the `main()` function.
3. Generative Text files
   - There are 5 input text files and 1 output text files which is created when running the commands.

NOTE: Using these text files is optional, and alternatively, you can create new files in this format.

4.2 Programming note

Chrono library: use for measuring running time of sorting functions

- `<chrono>` is a `C++` header that provides a collection of types and functions to work with time. It is a part of the `C++` Standard Template Library (STL) and it's included in `C++11` and later versions. [13]
- `<chrono>` provides three main types of clocks:
  - `system_clock`, `steady_clock`, and `high_resolution_clock`.
  - These clocks are used to measure time in various ways. [13]
- Here is a pseudocode:
  - `start_time = get_current_high_resolution_time()` (record the current time)
  - `runSortingAlgorithmWithTime(a, n, algoIndex)` (calling function)
  - `end_time = get_current_high_resolution_time()` (record the current time)
  - `duration = end_time - start_time` (calculate the duration in milliseconds)
  - `time = convert_to_milliseconds(duration)` (convert duration to milliseconds)

5 How to compile

To compile this project, follow the steps below:

1. Ensure you have a `C++` compiler installed on your system. Common options include `g++` (`std=c++11` or later) and `clang++`.
2. Open a terminal and navigate to the directory containing the project's source files.
3. Use the following command to compile the project:

   ```
   cd path/to/project
   g++ -o 07 *.cpp
   ```

   Here, `07` is the name of the executable file that will be generated.
4. Run the executable:

   ```
   ./07.exe
   ```

   - **Command 1:** Run a sorting algorithm on the user-provided data.
     - **Prototype:** `./07.exe.exe -a [Algorithm] [Input filename] [Output parameter(s)]`
     - **Example:** `./07.exe.exe -a radix-sort input.txt -both`
   - **Command 2:** Run a sorting algorithm on the data generated automatically with specified size and order.
     - **Prototype:** `./07.exe.exe -a [Algorithm] [Input size] [Input order] [Output parameter(s)]`
     - **Example:** `./07.exe.exe -a selection-sort 50 -rand -time`
   - **Command 3:** Run a sorting algorithm on ALL data arrangements of a specified size.
     - **Prototype:** `./07.exe.exe -a [Algorithm] [Input size] [Output parameter(s)]`
     - **Example:** `./07.exe.exe -a quick-sort 70000 -comp`
   - **Command 4:** Run two sorting algorithms on user-provided data.
     - **Prototype:** `./07.exe.exe -c [Algorithm 1] [Algorithm 2] [Input filename]`
     - **Example:** `./07.exe.exe -c heap-sort merge-sort input.txt`
   - **Command 5:** Run two sorting algorithms on the data generated automatically with specified size and order.
     - **Prototype:** `./07.exe.exe -c [Algorithm 1] [Algorithm 2] [Input size] [Input order]`
     - **Example:** `./07.exe.exe -c quick-sort merge-sort 100000 -nsorted`

This will compile and link the files `main.cpp`, `CommandLine.cpp`, `DataGenerator.cpp`, `SortWithCompare.cpp`, and `SortWithTime.cpp`, along with their corresponding header files.

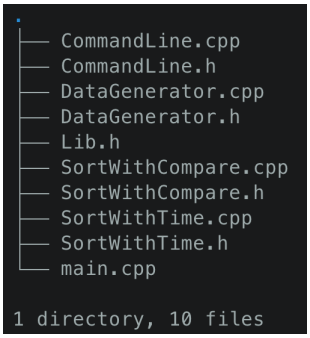


Figure 5.1: Folder Tree

6 References

[1] Selection Sort. website link, Jun 22 2024.

[2] Frank Carrano and Timothy Henry. *Data Abstraction & Problem Solving with C++: Walls and Mirrors, 7th Edition, 2016.* Jun 22 2024.

[3] DSA lectures Slides from Sorting Algorithms. Jun 20 2024.

[4] Bubble Sort. website link, Jun 22 2024.

[5] Frank M. Carrano and Timothy Henry. *Data Abstraction & Problem Solving with C++: Walls and Mirrors, Seventh Edition, 2017.* Pearson, Jun 20 2024.

[6] Quick Sort. website link, Jun 22 2024.

[7] Radix Sort Data Structures and Algorithms Tutorials GeeksforGeeks. website link, Jun 22 2024.

[8] #19.1. [C++] Thuật Toán Sắp Xếp Chèn | Sắp Xếp Nổi Bọt | Sắp Xếp Chọn | Sắp Xếp Đếm Phân Phối. youtube link, Jun 22 2024.

[9] Binary Insertion Sort. website link, Jun 22 2024.

[10] Tim Sort. website link, Jun 24 2024.

[11] Cocktail Sort. website link, Jun 24 22024.

[12] Flash Sort thuật toán sắp xếp với độ phức tạp O(n), Jun 22 2024.

[13] Chrono in C++ GeeksforGeeks. website link, Jun 26 22024.