VIETNAM NATIONAL UNIVERSITY
HO CHI MINH CITY

UNIVERSITY OF SCIENCE

Faculty of Information Technology

Data Structures and Algorithms

SORTING PROJECT REPORT

Nguyen Thien Bao - 22127032
Luong Quoc Dung - 22127078
Nguyen Bao Long - 22127243
Bui Minh Quan - 22127342

INSTRUCTORS
MR. Bui Huy Thong
MRS. Tran Thi Thao Nhi

July 28, 2023

EXPRESSION OF ATTITUDE

Firstly, our group would like to express our sincere gratitude to Mr. Bui Huy Thong and Mrs. Tran Thi Thao Nhi for your guidance and assistance with our Learning Project - The Sorting Project. Throughout the project, there were some features and functions that we encountered difficulties and doubts with, and it was thanks to their dedicated help, guidance and advice that our project was able to progress smoothly and effectively.

The tutor's advisor and support were indispensable throughout the process of completing the project. They provided us with essential information, knowledge, and practical experience in the field, as well as valuable feedback and guidance on how to improve our project. Their advice and suggestions also helped us to identify and solve any problems that arose during the project.

In addition, the tutor's support gave us the confidence and motivation to continue working on the project, even when we encountered difficulties or felt overwhelmed. They provided us with the necessary tools, resources, and guidance to ensure that we could complete the project to the best of our abilities.

Overall, the tutor's advisor and support played a crucial role in the success of our project. We are grateful for your assistance and look forward to applying the skills and knowledge we have gained in future projects.

1 INFORMATION PAGE

Name, ID, E-mail of group members:

ID	Name	Email
22127032	Nguyen Thien Bao	22127032@student.hcmus.edu.vn
22127078	Luong Quoc Dung	22127078@student.hcmus.edu.vn
22127243	Nguyen Bao Long	22127243@student.hcmus.edu.vn
22127342	Bui Minh Quan	22127342@student.hcmus.edu.vn

PROJECT LINK: `https://github.com/baolong110904/algorithm-analyzer`

SYSTEM INFORMATION

Computer Name	ASUS TUF Gaming F15 FX506LH-HN002T
Operating System	Windows 11 64-bit(10.0 Build 22621)
CPU	Intel® Core™ i5-10300H Processor
GPU	NVIDIA® GeForce® GTX 1650, 4GB GDDR6
RAM	GB DDR4-2933 SO-DIMM
Storage	512GB M.2 NVMe™ PCIe® 3.0 SSD
Display	15.6” FHD (1920 x 1080) 16:9, IPS-level panel, 144Hz
Wifi	Wifi 6 support

2 INTRODUCTION PAGE

Abstract

Sorting algorithms are fundamental techniques for organizing and manipulating data in computer science. They are widely used in various applications, such as searching, encryption, compression, and optimization. However, different sorting algorithms have different characteristics and performances, depending on the data order, size, and structure. Therefore, it is important to understand and compare the advantages and disadvantages of different sorting methods.

This essay describes a project that implements and compares 11 sorting algorithms using C/C++ programming language. The algorithms are Selection Sort, Insertion Sort, Bubble Sort, Shaker Sort, Shell Sort, Heap Sort, Merge Sort, Quick Sort, Counting Sort, Radix Sort, and Flash Sort. The project follows a specific scenario to test the algorithms on four data orders (Sorted, Nearly sorted, Reverse sorted, and Randomized) and six data sizes (10,000 to 500,000 elements). The project measures the running time and number of comparisons of each algorithm on each data order and size.

The essay presents the results of the experiments in tables and graphs and analyzes the performance of each algorithm. The essay also explains the ideas and complexities of each algorithm and discusses the advantages and disadvantages of different sorting methods. The essay concludes with some suggestions for future improvements and applications of sorting algorithms.

The essay aims to provide a comprehensive and practical overview of sorting algorithms and to demonstrate their usefulness and limitations in various contexts. The essay also aims to enhance the skills and knowledge of the students who participated in the project. The essay is organized as follows: Introduction, Algorithms, Experiments, Results and Analysis, and Conclusion.

Keywords: Sorting, graphs, experiments

Contents

2.1 Project Description

The project implements 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*), and tests them on different ascending data orders (*Nearly sorted, Reverse sorted, and Randomized*) and sizes(*10,000, 30,000, 50,000, 100,000, 300,000, and 500,000 elements*). The project measures and analyzes the running time and number of comparisons of each algorithm. The project aims to enhance the understanding and skills of the students who participated in the project and to provide a useful reference for choosing the appropriate sorting method for different situations. The project also follows the specific requirements and regulations of the Data Structures and Algorithms course.

2.1.1 Project Objectives

The performance metrics are the running time (in milliseconds) and the number of comparisons in each algorithm. The experiments will be conducted following a nested loop structure that iterates over each data order, each data size, and each sorting algorithm. For each iteration, an array will be created with the corresponding data order and size, and then sorted using the corresponding algorithm. The running time and the number of comparisons will be measured and recorded.

The expected outcomes of this project are:

- Graphs that show the relationship between the running time or the number of comparisons and the data order or the data size for each sorting algorithm.
- Comments that analyze the graphs and explain the observed trends and patterns.
- An overall comparison of the sorting algorithms based on their performance on all data orders and sizes.
- A discussion of the advantages and disadvantages of each sorting algorithm in terms of stability, complexity, memory usage, etc.

3 ALGORITHM PRESENTATION

3.1 Sorting algorithms with time complexity O(n^2)

Selection Sort

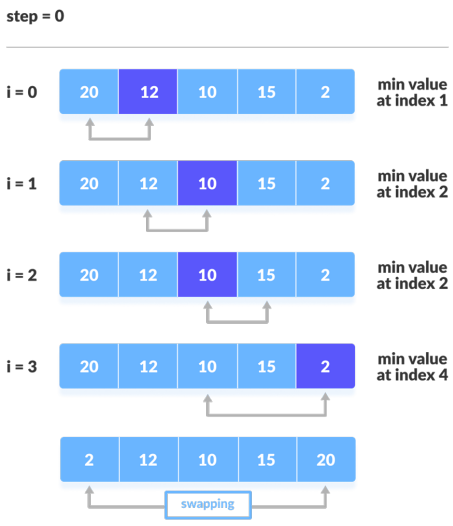
IDEA:

Selection sort is a simple and efficient sorting algorithm that *divides the input list into a sorted and an unsorted region. It repeatedly finds the minimum element from the unsorted region and moves it to the end of the sorted region until the whole list is sorted.*

- After each selection and swapping, the size of the sorted region grows by 1 and the size of the unsorted region shrinks by 1.
- A list of n elements requires $n - 1$ passes to completely rearrange the data.
- Each time we move one element from the unsorted sublist to the sorted sublist, we have completed a sort pass.

STEP-BY-STEP DESCRIPTIONS:

- Step 1. Set the increment variable i = 1
- Step 2. Find the smallest element in a[i. . n]
  - Swap the smallest element found above 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's the pseudocode for the selection sort algorithm

Algorithm 1 Selection Sort Algorithm

```
1: procedure SelectionSort(arr[0 ... n − 1])
2:     for i ← 0 to n − 2 do
3:         minIndex ← i
4:         for j ← i + 1 to n − 1 do
5:             if arr[j] < arr[minIndex] then
6:                 minIndex ← j
7:         Swap arr[i] with arr[minIndex]
```

COMPLEXITY EVALUATION:

- The number of comparisons in selection sort of n elements is given by:

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

The inner loop executes the size of the unsorted part minus 1, and in each iteration, there is one key comparison.

- The number of assignments in selection sort of n elements is $3(n-1)$.

The outer loop executes $n - 1$ times and invokes swapping once at each iteration.
Together, the number of key operations that a selection sort of n elements requires is:

$$\frac{n(n-1)}{2} + 3(n-1) = \frac{n^2}{2} + 5n - 3$$

- Thus, selection sort is $O(n^2)$ in all cases.

Time Complexity: $O(n^2)$ for both average and worst-case scenarios.
Space Complexity: $O(1)$ since it performs in-place swapping, as no extra space is required for the Selection sort algorithm.

VARIANTS/IMPROVEMENTS:

Here's the pseudocode for the *Min-Max selection* sort algorithm

Algorithm 2 Min-Max Selection Sort

```
1: procedure MinMaxSelectionSort(arr, n)
2:     for i ← 0 to n/2 do
3:         minIndex ← i
4:         maxIndex ← n − 1 − i
5:         for j ← i + 1 to n − i do
6:             if arr[j] < arr[minIndex] then
7:                 minIndex ← j
8:             if arr[j] > arr[maxIndex] then
9:                 maxIndex ← j
10:        swap arr[i] and arr[minIndex]
11:        if maxIndex = i then
12:            maxIndex ← minIndex
13:        swap arr[n − 1 − i] and arr[maxIndex]
```

- The algorithm works by finding the minimum and maximum elements in the unsorted part of the array and swapping them with the first and last elements of the unsorted part of the array.

  - The outer loop runs for (n/2) iterations, as it finds both the minimum and maximum elements simultaneously in a single pass.
  - The inner loop runs for (n/2), (n/2-1), ..., 1 iterations as the unsorted part of the array decreases in each iteration.

- Each iteration of the inner loop involves two comparisons (one for finding the minimum and one for finding the maximum) and two potential swaps.
- The time complexity of min-max selection sort is also O($n2$). Although it reduces the number of passes through the array, the number of key comparisons and swaps remains the same as the traditional selection sort, resulting in the same time complexity.

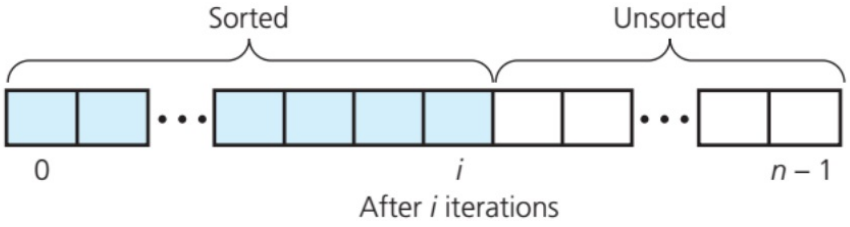
Insertion Sort

IDEA:

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

- Take the first element of the unsorted region and places it into its correct position in the sorted region
  - After each placement, the size of the sorted region grows by 1 and the size of the unsorted region shrinks by 1.
- A list of n elements requires n 1 passes to completely rearrange the data.

An insertion sort partitions the array into two regions



STEP-BY-STEP DESCRIPTIONS:

- Step 1. Set the increment variable $i = 2$
- Step 2. Find the correct position pos in $a[1 \dots i-1]$ to insert $a[i]$, i.e., where $a[\text{pos}-1] \leq a[i] \leq$a[pos]
  - Set x = a[i]
  - Move forward $a[\text{pos} \dots i-1]$ one element.
  - Set $a[\text{pos}] = x$
  - 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's the pseudocode for the Insertion sort algorithm

Algorithm 3 Insertion Sort Algorithm

1: $i \leftarrow 2$
2: **while** $i \leq n$ **do**
3: Find the correct position pos in $a[1..i-1]$ to insert $a[i]$
4: $x \leftarrow a[i]$
5: Move forward $a[pos..i-1]$ one element
6: $a[pos] \leftarrow x$
7: Increase i by 1

COMPLEXITY EVALUATION:

- The number of comparisons in Insertion sort of $1 + 2 + \ldots + n - 1 = \frac{n(n-1)}{2}$
  - The inner loop executes the size of the sorted part, and in each iteration, there is one key comparison.
- The number of assignments: $\frac{n(n-1)}{2} + 2(n-1)$
  - The inner loop moves data items at most the same number of times for comparisons. The outer loop moves data items twice per iteration.
  - Together, the number of key operations that an insertion sort of n elements requires in the worst case is $\frac{n(n-1)}{2} + \frac{n(n-1)}{2} + 2(n-1) = n^2 + n - 2$.

Time complexity: The time complexity of the insertion sort algorithm is O(n^2) in the worst and average cases, and O(n) in the best case.

Best case	Worst case	Average case
$O(n)$	$O(n^2)$	$O(n^2)$

In the best case, when the **input array is already sorted**, the inner loop will not perform any swaps, resulting in linear time complexity. However, in the worst and average cases, each element may need to be compared and shifted multiple times, leading to quadratic time complexity.

Space complexity: The space complexity of insertion sort is O(1) because it operates in-place, meaning it does not require any additional data structures to be allocated dynamically based on the input size. The sorting is done by moving elements within the original array, and no extra memory is used other than a few constant variables used for temporary storage during the sorting process. As a result, the space complexity remains constant, regardless of the input size.

VARIANTS/IMPROVEMENTS:

Shell Sort

IDEA:

- **Shell Sort** is one of the oldest sorting algorithms and it's an extension of the Insertion Sort.

This algorithm is fast and easy to implement, but it's hard to measure its performances.

- Unlike Insertion Sort, Shell Sort starts by comparing the elements distant from each other by a certain gap that gets progressively decreased. By starting with the most distant elements, it can optimize performances as it's not limited by just comparing two adjacent elements.

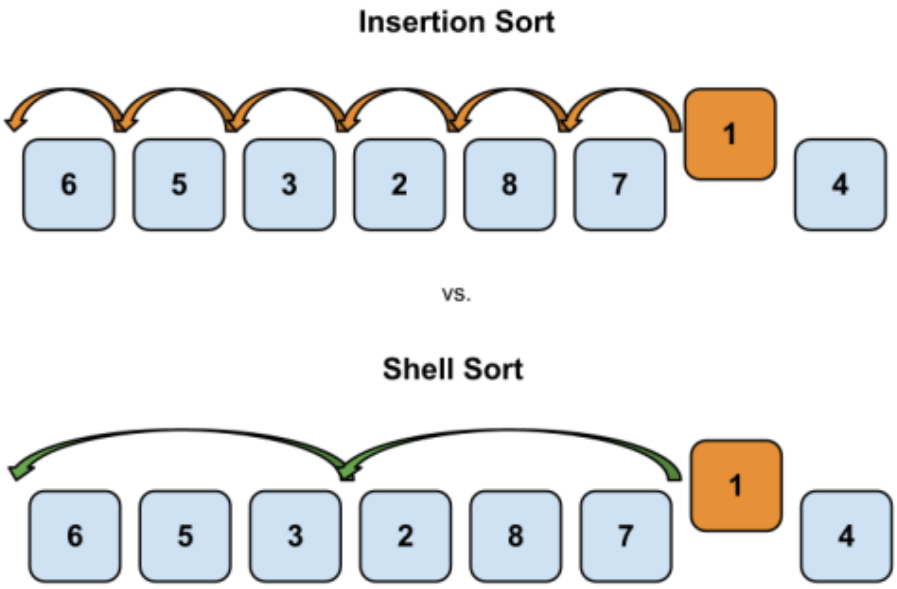


Figure 1: Differences between Insertion and Shell sort

STEP-BY-STEP DESCRIPTIONS:

Shell sort algorithm sorts an array of integers in ascending order.

- The outermost loop iterates over the gap sequence, starting with a gap size of n/2 and reducing it by half in each iteration until it reaches 1.
- The second loop iterates over the elements in the array, starting from the current gap size (interval) and incrementing by 1 in each iteration.
- The innermost loop performs an insertion sort on the sub-lists created by the current gap size.

Algorithm 4 Shell Sort

```
1: procedure ShellSort(arr, n)
2:     for interval ← n/2 to 1 do
3:         for i ← interval to n − 1 do
4:             value ← arr[i]
5:             j ← i
6:             while j ≥ interval and arr[j − interval] > value do
7:                 arr[j] ← arr[j − interval]
8:                 j ← j − interval
9:             arr[j] ← value
```

COMPLEXITY EVALUATION:

- **The time complexity** of Shell sort varies **depending on the gap sequence** used. The worst-case time complexity of Shell sort is O(n^2), while the best-case time complexity is O(n*log n). The average case time complexity depends on the gap sequence used and can be O(n*log n)
- **The space complexity** of Shell Sort is O(1). Determining the exact time complexity of Shell Sort is still an open problem.

Bubble Sort

IDEA:

- **Bubble Sort** is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.
- Bubble Sort is an iterative sorting algorithm that imitates the movement of bubbles in sparkling water. The bubbles represent the elements of the data structure.
- ***The bigger bubbles reach the top faster than smaller bubbles***, and this algorithm works in the same way. It *iterates through the data structure and for each cycle compares the current element with the next one, swapping them if they are in the wrong order.*

STEP-BY-STEP DESCRIPTIONS:

- Step 1. Set the increment variable *i = 1*
- Step 2. Swap any pair of adjacent elements in a[1 . . . 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 is reached.
  - After finishing the above loop, increase i by 1 and go to Step 3
- Step 3. Check whether all elements are 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's the pseudocode for the Bubble sort algorithm

Algorithm 5 Bubble Sort

```
1: procedure BubbleSort(a, n)
2:     i ← 1
3:     while i < n do
4:         j ← 1
5:         while j ≤ n − i do
6:             if a[j] > a[j + 1] then
7:                 swap a[j] with a[j + 1]
8:             j ← j + 1
9:         i ← i + 1
```

COMPLEXITY EVALUATION:

- **The number of comparisons:** n-1 + n-2 + $\cdots + 1 = \frac{n(n-1)}{2}$
  - The inner loop executes the size of the unsorted part minus 1, and in each iteration, there is one key comparison.
- **The number of exchanges:** the same as above
  - Each exchange requires **three assignments.**
- Together, the number of key operations that a bubble sort of n elements requires in the worst case is $2n^2 - 2n$.

The time complexity of Bubble sort is O(n^2) in the *worst and average* cases, where n is the number of elements in the array. This means that the algorithm takes quadratic time to complete, and its performance decreases as the size of the input increases. In the *best case*, when the input array is already sorted, the time complexity of Bubble sort is O(n), which means that the algorithm takes linear time to complete.

The space complexity is O(1), which means that the algorithm uses a constant amount of additional memory, regardless of the size of the input. Bubble sort is an in-place sorting algorithm, which means that it does not require any additional memory to sort the array, apart from a small amount of memory used to store temporary variables.

VARIANTS/IMPROVEMENTS:

Shaker Sort

IDEA:

- **Shaker sort**, also known as *cocktail sort* or *bidirectional bubble sort*, is a variation of the bubble sort algorithm. Like the bubble sort algorithm, shaker sort sorts an array of elements by repeatedly swapping adjacent elements if they are in the wrong order.
  - Start from the beginning of the array and compare each adjacent pair of elements. If they are in the wrong order, swap them.
  - Continue iterating through the array in this manner until you reach the end of the array.

- Then, move in the opposite direction from the end of the array to the beginning, comparing each adjacent pair of elements and swapping them if necessary.
- Continue iterating through the array in this manner until you reach the beginning of the array.
- Repeat steps 1-4 until the array is fully sorted.

STEP-BY-STEP DESCRIPTIONS:

Before understanding how Shaker sort works, let's look at the figure below

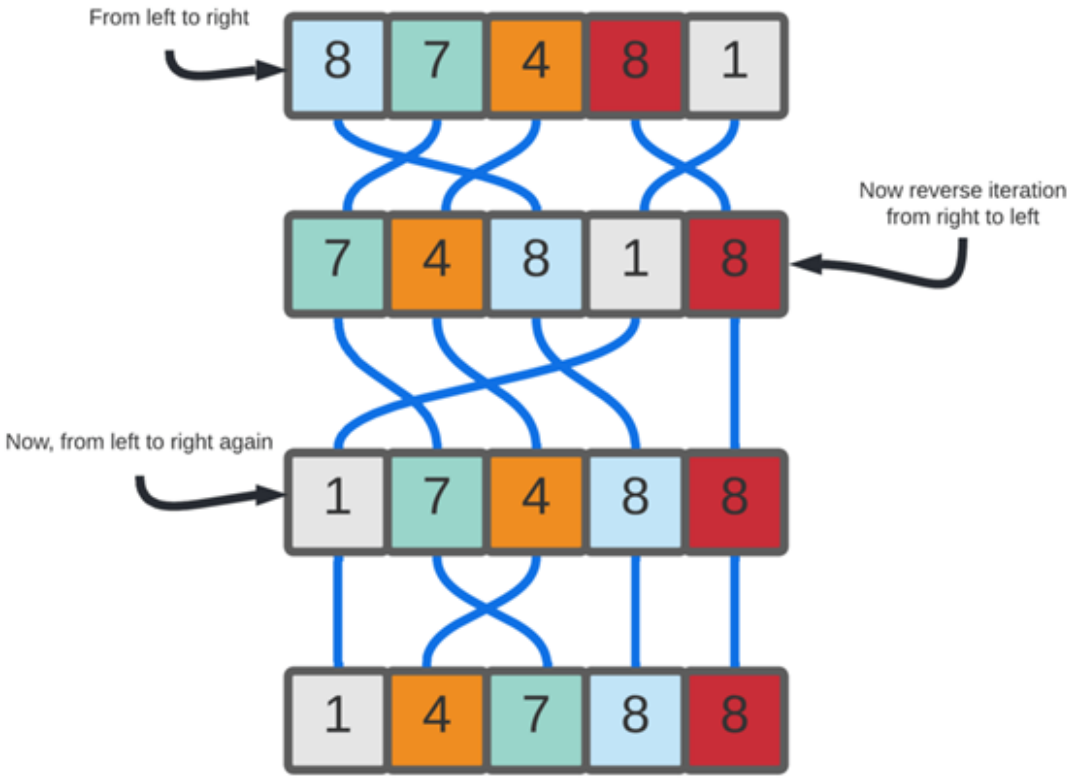


Figure 2: In the first phase, the heaviest element ascends to the end of the array. In the second phase, the lightest element descends to the beginning of the array

So here are step-by-step descriptions of Shaker sort:

The first phrase: algorithm traverses the array from right to left, comparing adjacent elements and swapping them if they are in the wrong order

1. Step 1: Let left = 1, right = n-1, k = n-1
2. Step 2: Let j = right
3. Step 3: If j ¿= left then continue, else go to step 6
4. Step 4: If arr[j - 1] ¿ arr[j] then swap them and let k = j
5. Step 5: Decrease j by 1 and go back to step 3

6. Step 6: Make left = k + 1

The second phase: algorithm traverses the array from left to right, and moves the largest element to the end of the array.

7. Step 7: Let j = left
8. Step 8: If j ¡= right then continue, else go to step 11
9. Step 9: If arr[j - 1] ¿ arr[j] then swap them and let k = j
10. Step 10: Increase j by 1 and go to step 8
11. Step 11: Make right = k - 1
12. Step 12: If left ¡= right then go back to step 2

Pseudo code for Shaker Sort described above:

Algorithm 6 Shaker Sort

```
1:  procedure ShakerSort(arr, n)
2:      Let left = 1, right = n-1, k = n-1
3:      Let j = right
4:      while j ¿= left do
5:          if arr[j - 1] ¿ arr[j] then
6:              swap(arr[j - 1], arr[j])
7:              k = j
8:          Decrease j by 1
9:      Make left = k + 1
10:     Let j = left
11:     while j ¡= right do
12:         if arr[j - 1] ¿ arr[j] then
13:             swap(arr[j - 1], arr[j])
14:             k = j
15:         Increase j by 1
16:     Make right = k - 1
17:     if left ¡= right then
18:         Go back to step 2
```

COMPLEXITY EVALUATION:

- **The time complexity** of the shaker sort algorithm is O(n^2) in the *worst case*, where n is the number of elements in the array. This is because, in the worst case, the algorithm needs to perform n/2 passes, and each pass takes O(n) time to compare and swap adjacent elements. The *best-case* time complexity of shaker sort is O(n), which occurs when the input array is already sorted.
- **The space complexity** of shaker sort is O(1), as it only requires a constant amount of additional memory to perform the sorting, regardless of the size of the input array.

3.2 Sorting algorithms with time complexity O(nlog2n)

Heap Sort

IDEA:

- **Heap sort** is a *comparison-based sorting technique based on Binary Heap* data structure. It is similar to the selection sort where we *first find the minimum element and place the minimum element at the beginning*. Repeat the same process for the remaining elements.
- First convert the array into heap data structure using heapify, then one by one delete the root node of the Max-heap and replace it with the last node in the heap and then heapify the root of the heap. Repeat this process until size of heap is greater than 1.
- Build a heap from the given input array.
- Repeat the following steps until the heap contains only one element:
  - Swap the root element of the heap (which is the largest element) with the last element of the heap.
  - Remove the last element of the heap (which is now in the correct position).
  - Heapify the remaining elements of the heap.
- The sorted array is obtained by reversing the order of the elements in the input array.

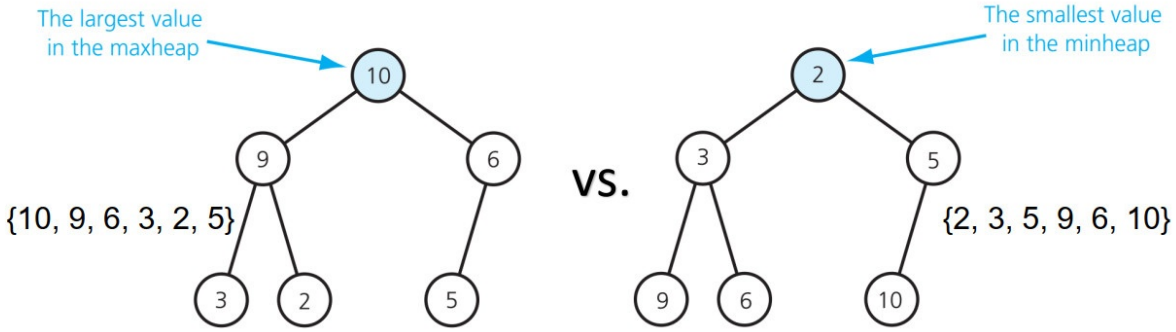
STEP-BY-STEP DESCRIPTIONS:

1. **Phase 1. Heap construction. Construct a heap for the array.**

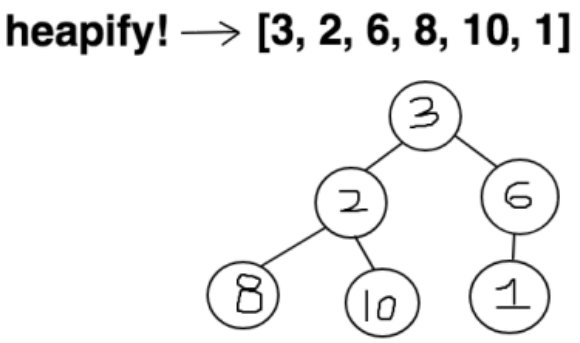
A max heap is a sequence of n elements, $h_1, h_2, \ldots, h_n$ such that $h_i \geq h_{2i}$ and $h_i \geq h_{2i+1}$ for all $i = 1, 2, \ldots, \frac{n}{2}$.

- The sequence of elements $h_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, h_n$ is a natural heap.
- The element h_1 of a heap is the largest value.

We also have min heap with opposite characteristics.
Here are how Maxheap and Minheap work!



- A new heap is obtained by letting *the element on top of the subheap "sift down" along the path of the larger comparands*, which at the same time move up.
  - The "sift down" process stops when the element on top of the subheap is larger than or equal to both its comparands.
- The heap is extended to the left where in each step a new element is included and properly positioned by a sift.



Here's the pseudocode of Heapify:

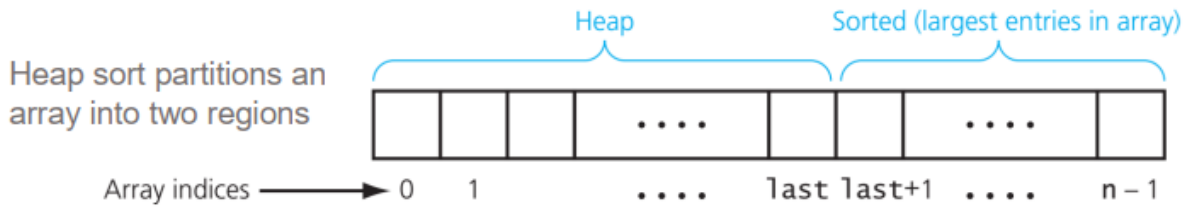
Algorithm 7 Heapify Algorithm

```
1:  procedure HEAPIFY(A as array, n as int, i as int)
2:      max ← i
3:      leftchild ← 2i + 1
4:      rightchild ← 2i + 2
5:      if leftchild ≤ n and A[i] < A[leftchild] then
6:          max ← leftchild
7:      else
8:          max ← i
9:      if rightchild ≤ n and A[max] > A[rightchild] then
10:         max ← rightchild
11:     if max ≠ i then
12:         Swap A[i] with A[max]
13:         HEAPIFY(A, n, max)
```

2. **Phase 2. Maximum deletion. Apply maximum key deletion $n-1$ times to the remaining heap**

- Swap the first element and the last element of the heap
- Decrease heapSize by 1, heapSize $= n - 1$
- While heapSize > 1
  - Rebuild the heap at the first position, $a[0..heapSize]$
  - Swap the first element and the last element of the heap
  - Decrease heapSize by 1, heapSize = heapSize -1

Here's the pseudocode of Heap sort:



Algorithm 8 Heapsort Algorithm

```
1: procedure HEAPSORT(A as array)
2:     n = length(A)
3:     for i = n/2 downto 1 do
4:         HEAPIFY(A, n, i)
5:     for i = n downto 2 do
6:         exchange A[1] with A[i]
7:         A.heapsize = A.heapsize − 1
8:         HEAPIFY(A, i, 0)
```

COMPLEXITY EVALUATION:

The heap sort algorithm includes two stages:

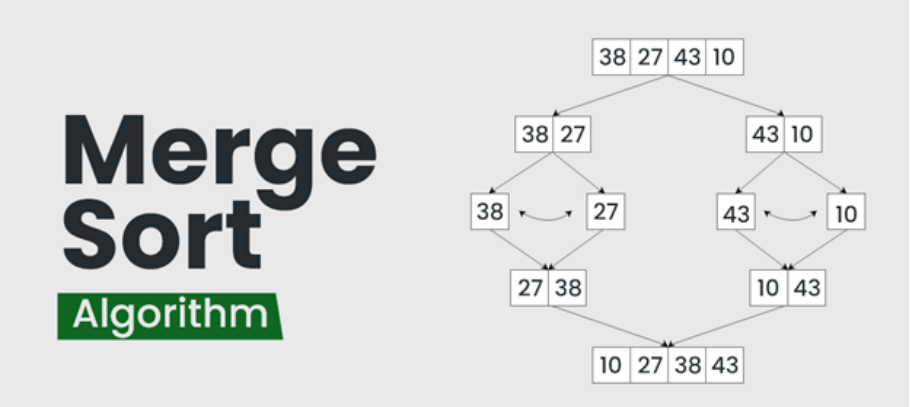
- Heap construction takes $O(n \log_2 n)$ time. Because there are $\frac{n}{2}$ sifts, each of which runs in $O(\log_2 n)$ time.
- Sorting takes $O(n \log_2 n)$ time. It executes $n - 1$ steps where each step runs in $O(\log_2 n)$ time.

Thus, heap sort is $O(n \log_2 n)$ in all cases. It is not recommended for small numbers of elements.

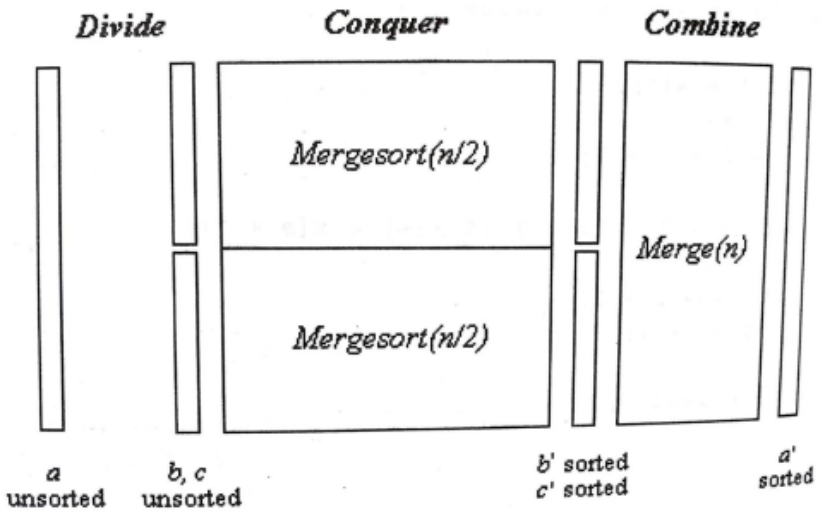
- **The time complexity** of the Heap Sort algorithm is O(n*log n) in all cases (best, average, and worst). This is because the heap construction takes O(n log n) time, as there are n/2 sifts, each of which runs in O(log n) time. The sorting stage also takes O(n*log n) time, as it executes n-1 steps where each step runs in O(log n) time.
- **The space complexity** of Heap Sort is O(1), as it is an in-place algorithm. However, it is typically 2-3 times slower than a well-implemented QuickSort.

Merge Sort

IDEA:



Merge sort is a recursive algorithm that continuously splits the array in half until it cannot be further divided i.e., the array has only one element left (an array with one element is always sorted). Then the sorted subarrays are merged into one sorted array.



STEP-BY-STEP DESCRIPTIONS:

The algorithm can be broken down into three steps: divide, conquer, and combine.

1. **Divide:** The first step is to divide the list into two halves. This is done by finding the midpoint of the list and dividing it into two sublists.
2. **Conquer:** The second step is to recursively sort each of the two sublists by repeating the divide and conquer steps. This continues until the sublists are of length 1 or 0, at which point they are considered sorted.
3. **Combine:** The final step is to merge the two sorted sublists back together into a single sorted list. This is done by comparing the first elements of each sublist and taking the smaller of the two and adding it to the final sorted list. This process is repeated until one of the sublists is empty, at which point the remaining elements of the other sublist are added to the final sorted list.

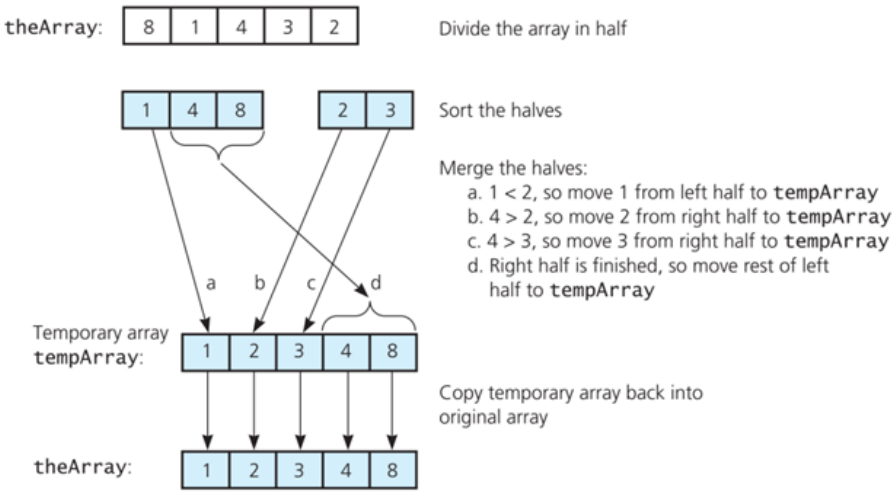


Figure 3: Here's how the array is sorted by divide, conquer and combine

The pseudocode of Merge process:

Algorithm 9 Merge Sort Algorithm

```
1:  procedure MERGE(A, p, q, r)
2:      n_1 ← q − p + 1
3:      n_2 ← r − q
4:      let L[1 ... n_1 + 1] and R[1 ... n_2 + 1] be new arrays
5:      for i ← 1 to n_1 do
6:          L[i] ← A[p + i − 1]
7:      for j ← 1 to n_2 do
8:          R[j] ← A[q + j]
9:      L[n_1 + 1] ← ∞
10:     R[n_2 + 1] ← ∞
11:     i ← 1
12:     j ← 1
13:     for k ← p to r do
14:         if L[i] <= R[j] then
15:             A[k] = L[i]
16:             i = i + 1
17:         else
18:             A[k] = R[j]
19:             j = j + 1
```

And this is when we recursively call the mergeSort in the initiative stages:

COMPLEXITY EVALUATION:

Algorithm 10 Merge Sort Algorithm

```
1: procedure MergeSort(A, p, r)
2:     if p < r then
3:         q ← ⌊(p + r)/2⌋
4:         MergeSort(A, p, q)
5:         MergeSort(A, q + 1, r)
6:         Merge(A, p, q, r)
```

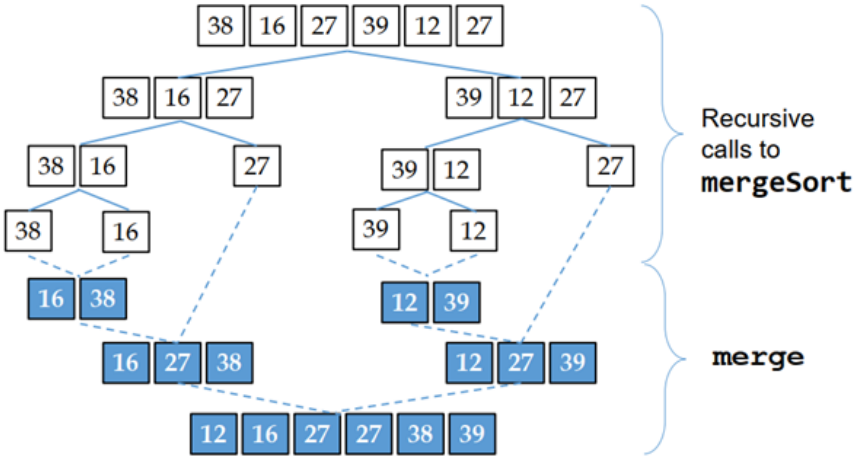


Figure 4: 2 separate processes when recursive calls to mergeSort and merge

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, and 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: $k = \log_2 n$ (for $n = 2^k$) or $k = 1 + \log_2 n$ (for $n < 2^k$).

This table demonstrates the followed-up levels and operations of each iteration:

Level	MergeSort Called	Merge Called	Operations
0	1	1	$3n-1$
1	2	2	$3n-2$
2	4	4	$3n-4$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
m	2^m	2^m	$3n-2^m$

- Each level of the recursion requires $O(n)$ operations, and there are either $\log_2 n$ or $1+\log_2 n$ levels.
- Thus, merge sort is $O(n \log_2 n)$ in all cases.

1. **The time complexity** of the Merge Sort algorithm is O(n*logn) in all cases (best, average, and worst). This is because the algorithm uses a divide-and-conquer approach, where it continuously

splits the array in half until it cannot be further divided, and then merges the sorted subarrays back together. Each level of recursion requires O(n) operations, and there are either log n or 1 + log n levels.

2. **The space complexity** of Merge Sort is O(n). This is because the algorithm requires an additional temporary array to merge the sorted subarrays back together.

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.
- The key process in quickSort is a partition[]. The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot.
- Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.

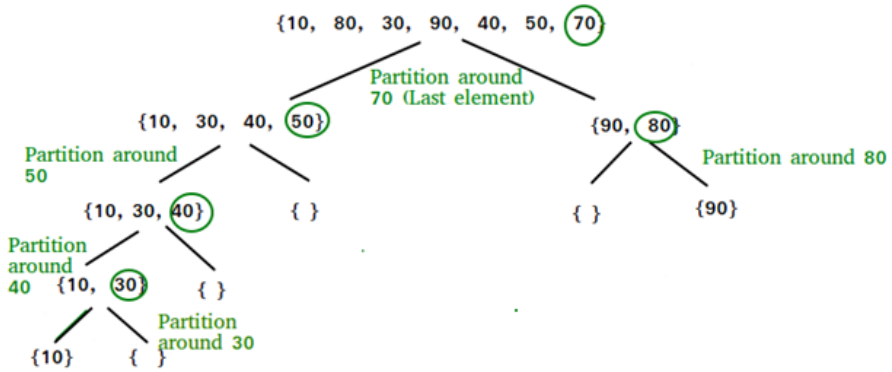


Figure 5: How Quicksort works with the last pivot chosen

STEP-BY-STEP DESCRIPTIONS:

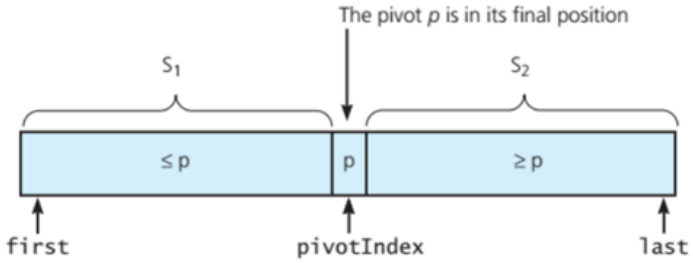
CHOICE OF PIVOT:

There are many different choices for picking pivots:

- Always pick the first element as a pivot.
- Always pick the last element as a pivot
- Pick a random element as a pivot.
- Pick the middle as the pivot.(implemented below)

PARTITION ALGORITHM:

Partition the initial array segment into two regions as follows



Recursively partition on smaller segments, i.e. S1 and S2, until the array contains only one element.

Here are step-by-step process of the quick sort algorithm:

Consider an initial array, $a[\text{first} \ldots \text{last}]$

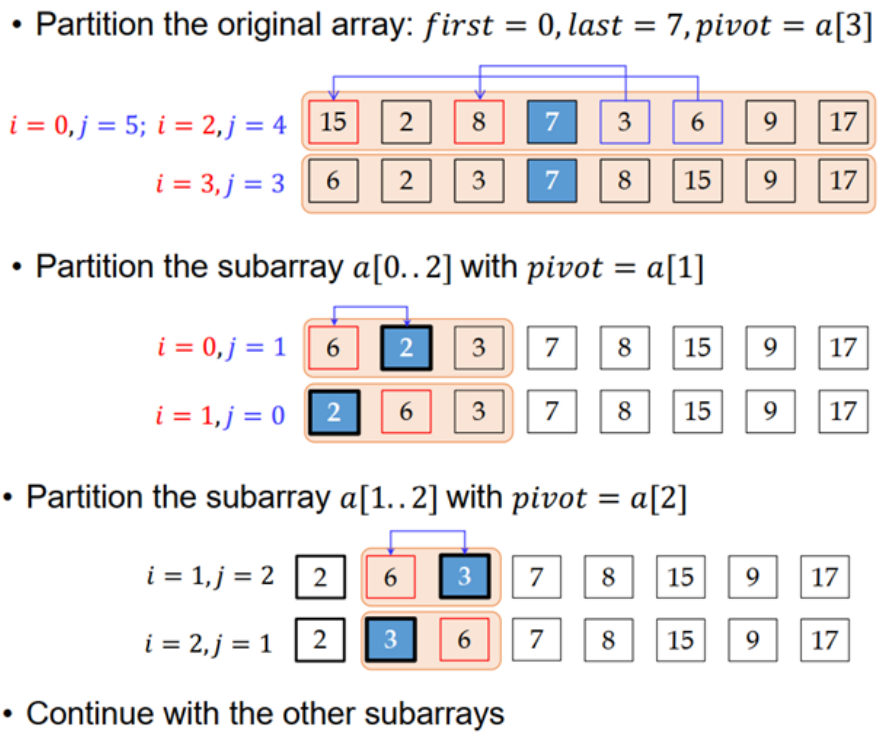
1. **Step 1.** Pick the pivot $p = a[k]$, where $k = (\text{first} + \text{last})/2$
2. **Step 2**. Identify pairs of elements that are not in their correct positions and swap them
   - Set the increment variables, $i = \text{first}$ and $j = \text{last} - 1$
   - While $a[i] < p$ do increase i by 1.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

And this is the pseudocode for those aforementioned steps:

Algorithm 11 QuickSort Algorithm

```
1:  procedure QuickSort(A, first, last)
2:      if first < last then
3:          k ← (first + last)/2
4:          p ← A[k]
5:          i ← first
6:          j ← last − 1
7:          while i ≤ j do
8:              while A[i] < p do
9:                  i ← i + 1
10:             while A[j] > p do
11:                 j ← j − 1
12:             if i ≤ j then
13:                 Swap A[i] with A[j]
14:                 i ← i + 1
15:                 j ← j − 1
16:         QuickSort(A, first, j)
17:         QuickSort(A, i, last)
```

ILLUSTRATION OF QUICKSORT:



VARIANTS/IMPROVEMENTS:

There are multiple of variants of Quick sort, and here is the Median-of-three Quicksort.

IDEA:

The Median-of-three QuickSort is a variant of the QuickSort algorithm that chooses the pivot element as the median of three elements: the first, middle, and last elements of the array, The idea behind this approach is to choose a pivot that is likely to represent the middle of the values to be sorted, which can help mitigate some common pathologies of sorted or reverse-sorted inputs.

The chosen pivot is swapped with the last element a[last] to get it out of the way during partition. Various strategies exist for making the choice of pivot. The algorithm still maintains two searches:

- A forward search starting at the first entry looks for the first entry that is greater than or equal to the pivot, and
- A backward search starting at the next-to-last entry looks for the first entry that is less than or equal to the pivot.

Place the pivot between the two subarrays, $S1$ and $S2$, by swapping a[indexFromLeft] and a[last].

STEP-BY-STEP DESCRIPTIONS:

Here is a step-by-step process for how the Median-of-three QuickSort works:

- Pick the pivot: Choose the first, middle, and last elements of the array and find their median. This value will be used as the pivot.

Figure 6: Take as pivot the median of three entries: the first entry, the middle entry and the last entry

Figure 7: Sort only those entries and use the middle value as the pivot

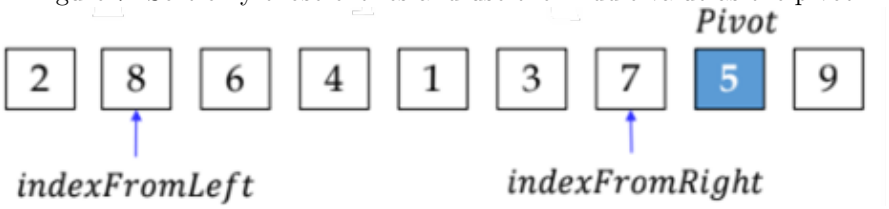


Figure 8: Position the pivot before partitioning

And this is how we figure out 3 elements: fist entry, middle entry, last entry in pseudocode:

Algorithm 12 sortFirstMiddleLast Function

```
1: function SORTFIRSTMIDDLELAST(arr, first, last)
2:     mid ← first + (last − first)/2
3:     if arr[first] > arr[mid] then
4:         Swap arr[first] with arr[mid]
5:     if arr[mid] > arr[last] then
6:         Swap arr[mid] with arr[last]
7:     if arr[first] > arr[mid] then
8:         Swap arr[first] with arr[mid]
9:     return mid
```

- Partition: Partition the array around the pivot element using the same process as in regular QuickSort. This involves identifying pairs of elements that are not in their correct positions and swapping them:

  • **Swap the pivot:** The chosen pivot is swapped with the last element of the array to get it out of the way during partitioning.

  • **Identify pairs of elements:** The algorithm maintains two searches: a forward search starting at the first entry that looks for the first entry that is greater than or equal to the pivot, and a backward search starting at the next-to-last entry that looks for the first entry that is less than or equal to the pivot.

  • **Swap elements:** If a pair of elements is identified where one element is greater than or equal to the pivot and the other element is less than or equal to the pivot, these elements are swapped.

  • **Place the pivot:** Once all pairs of elements have been identified and swapped, the pivot element is placed between the two subarrays by swapping it with the first element that is greater than or equal to it.

- Recursion: Recursively apply the Median-of-three QuickSort algorithm to the two subarrays on either side of the pivot until the entire array is sorted.

Algorithm 13 Median-of-three QuickSort Partition

```
1:  function PARTITION(A, first, last)
2:      mid ← first + (last − first)/2
3:      pivot ← median of A[first], A[mid], and A[last]
4:      Swap pivot with A[last]
5:      i ← first
6:      j ← last − 1
7:      while i ≤ j do
8:          while A[i] < pivot do
9:              i ← i + 1
10:         while A[j] > pivot do
11:             j ← j − 1
12:         if i ≤ j then
13:             Swap A[i] with A[j]
14:             i ← i + 1
15:             j ← j − 1
16:     Swap A[i] with A[last]
17:     return i
```

COMPLEXITY EVALUATION:

Best case	Worst case	Average case
$O(n \log_2 n)$	$O(n^2)$	$O(n \log_2 n)$

The time complexity of the QuickSort algorithm depends on the choice of the pivot element, which is used to partition the array 1.

- Best case: The best case time complexity of QuickSort is O(n*logn), which occurs when the pivot is always chosen as the median or closest to the median element.
- Average case: The average time complexity of QuickSort is also O(n*logn).
- Worst case: The worst case time complexity of QuickSort is O(n^2), which occurs when the pivot is always chosen as the smallest or largest element.

The space complexity of QuickSort is O(log n) for the in-place version, which uses a stack to keep track of the subarrays that still need to be sorted. This is because the algorithm uses a divide-and-conquer approach, where it continuously splits the array in half until it cannot be further divided, and then merges the sorted subarrays back together.

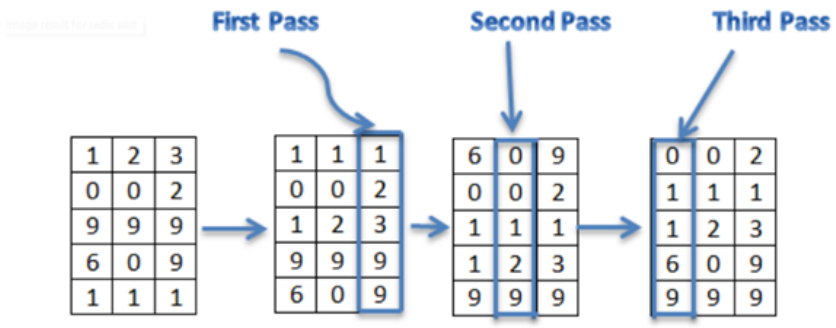
QUICK SORT VS. MERGE SORT

- Quick sort can be faster in practice and does not require the additional memory that merge sort needs for merging
- The efficiency of a merge sort is somewhere between the possibilities for a quick sort.

3.3 Sorting algorithms with time complexity O(n)

Radix Sort

IDEA:



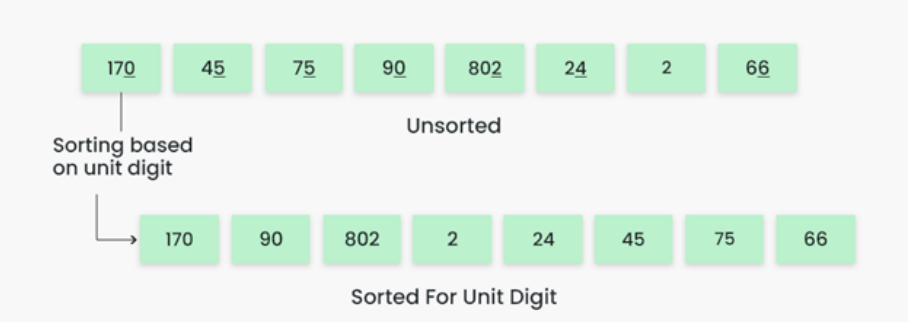
Radix Sort is a linear sorting algorithm that sorts elements by processing them digit by digit. It is an efficient sorting algorithm for integers or strings with fixed-size keys.

- Rather than comparing elements directly, Radix Sort **distributes the elements into buckets based on each digit's value**. By *repeatedly sorting the elements by their significant digits*, from the *least significant to the most significant*, Radix Sort achieves the final sorted order.
- The **key idea** behind Radix Sort is to exploit the concept of place value. It assumes that sorting numbers **digit by digit** will eventually result in a fully sorted list. Radix Sort can be performed using different variations, such as *Least Significant Digit (LSD) Radix Sort or Most Significant Digit (MSD)* Radix Sort.

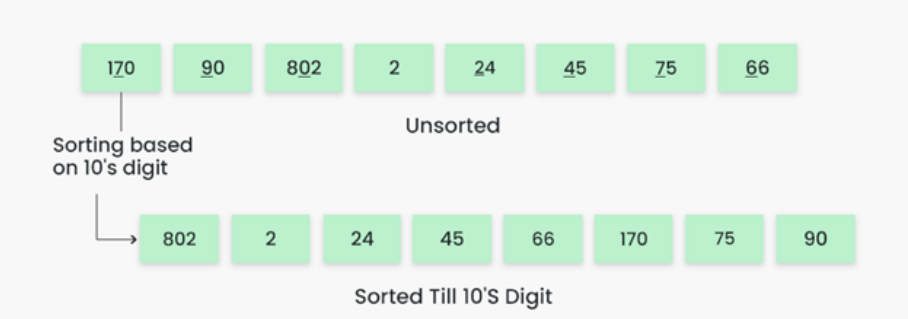
STEP-BY-STEP DESCRIPTIONS:

1. **Step 1.** Take the array. Check whether the number of digits in every array element is the same. If it is not the same, make it the same by using 0 before MSD.

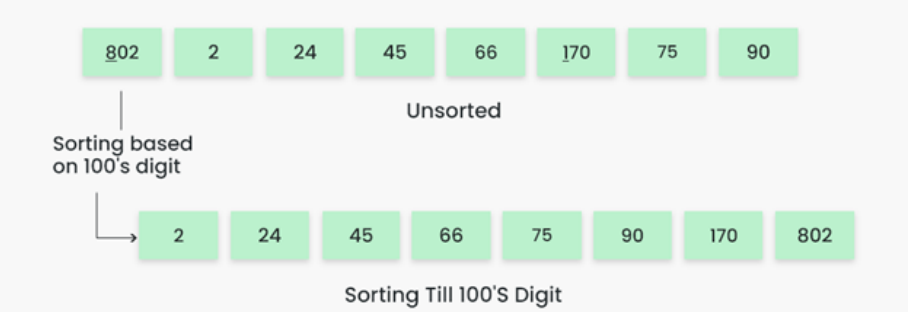
2. **Step 2.** Find how many buckets are needed. Now, if you are given a decimal number, the digit will fall in the range of 0 to 9, so take 10 buckets. If you are given a string, then characters will fall in the range a-z (26 alphabets), so consider 0 – 25 buckets.



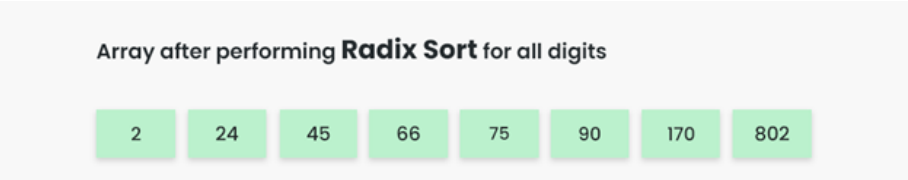
3. **Step 3.** Begin with the LSD (leftmost bit/character) and place the number based on the LSD in the appropriate bucket number. (Do not sort within the buckets). Just concatenate from the buckets and append the numbers in an empty array.



4. **Step 4.** Once it is done with one's place (LSD), follow step 3 again for ten's place, the hundred's place, and so on until the MSD is reached.



5. **Step 5.** The last output will be the resultant sorted array.



This is the pseudocode for radix sort, which sorts an array of integers by sorting them based on their digits from least significant to most significant. The function digit(a, i) returns the i-th digit of the integer a.

Algorithm 14 Radix Sort

```
1:  function RADIX-SORT(A)
2:      d ← max_{a∈A} log_10 a
3:      for i ← 1 to d do
4:          B ← empty array of size |A|
5:          C ← empty array of size 10
6:          for j ← 1 to |A| do
7:              C[digit(A[j], i)] ← C[digit(A[j], i)] + 1
8:          for j ← 2 to 10 do
9:              C[j] ← C[j] + C[j − 1]
10:         for j ← |A| to 1 do
11:             B[C[digit(A[j], i)]] ← A[j]
12:             C[digit(A[j], i)] ← C[digit(A[j], i)] − 1
13:         A = B
```

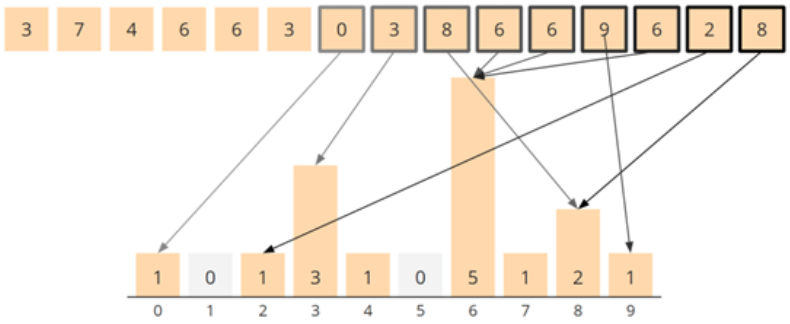
COMPLEXITY EVALUATION:

- It takes $2 \times n \times d$ moves to sort n strings, each of which has d characters.
  - n moves for forming groups and n moves for combining them into one group. The algorithm performs these 2 × n moves d times.
- No comparisons are necessary.
- Thus, radix sort is O n .
- It is not appropriate as a general-purpose sorting algorithm.
- Substantial demand of memory if using arrays use chain instead.
- **The time complexity** of radix sort is given by the formula **'T(n) = O(d*(n+b))'**, where 'd' is the number of digits in the given list, 'n' is the number of elements in the list, and 'b' is the base or bucket size used, which is normally base 10 for decimal representation.
- 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.

Counting Sort

IDEA:

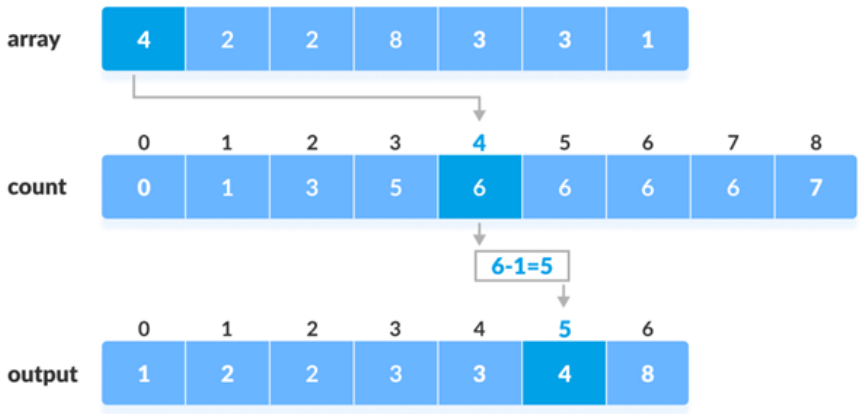
Counting sort is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (a kind of hashing). Then do some arithmetic operations to calculate the position of each object in the output sequence.



- For each element, its final position in the sorted array is indicated by the total number of smaller elements.
- The sorted order is determined based on counting.
  - This works in a situation that elements to be sorted belong to a known small set of values: $\forall i \in [1, u] : a_i \in N \wedge a_i \in [l, u]$
  - For simplicity, let $l = 0$

STEP-BY-STEP DESCRIPTIONS: Here's a step-by-step description of how counting sort works:

1. **Step 1:** Take a count array to store the count of each unique object.
2. **Step 2:** Store the count of each unique element in the count array. If any element repeats itself, simply increase its count.
3. **Step 3:** Modify the count array such that each element at each index stores the sum of previous counts.
4. **Step 4:** The modified count array indicates the position of each object in the output sequence.
5. **Step 5:** Find the index of each element of the original array in the count array. This gives the cumulative count.
6. **Step 6:** Output each object from the input sequence followed by increasing its count by 1.



This is the pseudocode for counting sort, which sorts an array of integers by counting the number of objects having distinct key values. The function takes as input an array A and the range of possible key values k:

Algorithm 15 Counting Sort

```
1:  function COUNTING-SORT(A, k)
2:      let C[0..k] be a new array
3:      for i ← 0 to k do
4:          C[i] ← 0
5:      for j ← 1 to A.length do
6:          C[A[j]] ← C[A[j]] + 1
7:      // C[i] now contains the number of elements equal to i
8:      for i ← 1 to k do
9:          C[i] ← C[i] + C[i − 1]
10:     // C[i] now contains the number of elements less than or equal to i
11:     let B[1..A.length] be a new array
12:     for j ← A.length down to 1 do
13:         B[C[A[j]]] ← A[j]
14:         C[A[j]] ← C[A[j]] − 1
```

COMPLEXITY EVALUATION:

- Better efficiency yet high space complexity
- Two consecutive passes are made through the input array.
- In practice, this algorithm should be used when $u \leq n$
- It would be disaster if $u \gg n$
- Counting radix sort: an alternative implementation of radix sort that avoids using bins
- **The time complexity** of counting sort is **'O(n+k)'**, where 'n' is the number of elements to be sorted and 'k' is the range of possible key values. This is because counting sort counts the occurrence of each element in the input range, which takes 'k' time, and then finds the correct index value of each element in the sorted output array, which takes 'n' time.
- **The space complexity** is also **'O(n+k)'**. This is because counting sort requires an auxiliary array 'C' of size 'k' to store the count of each unique object, and an output array 'B' of size 'n' to store the sorted elements.

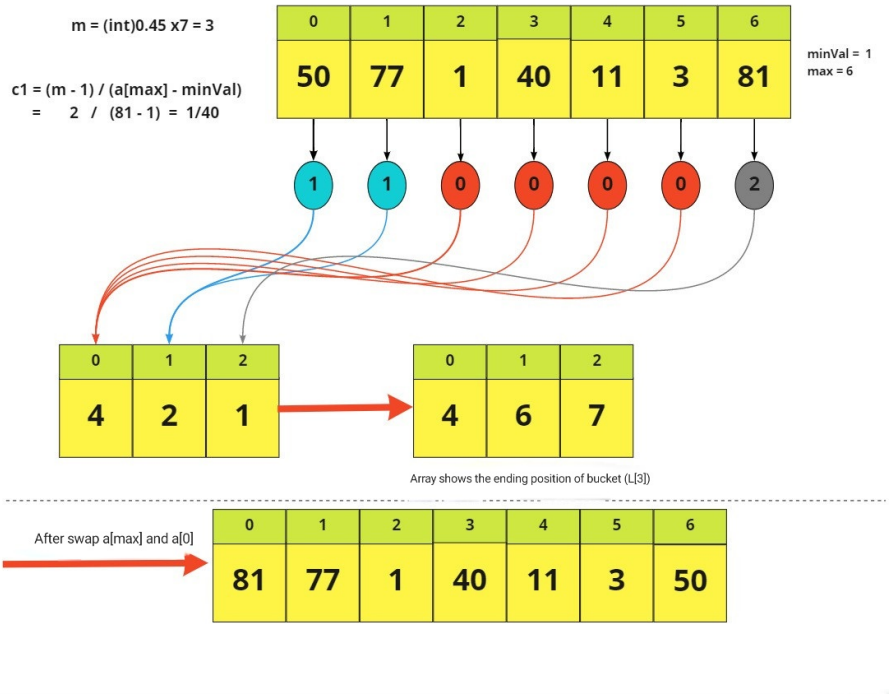
Flash Sort

IDEA:

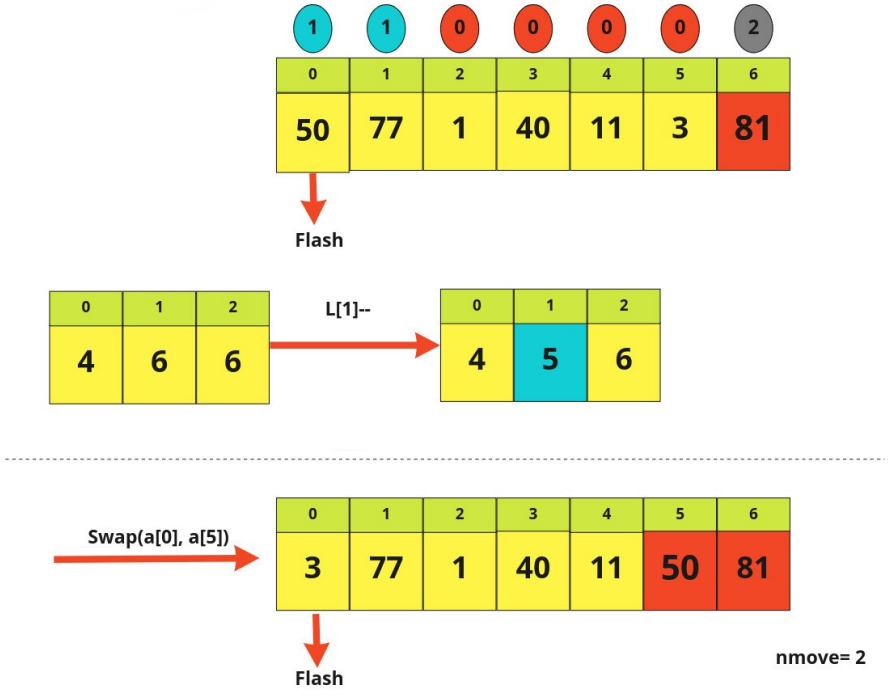
- **Flash sort** is a distribution sorting algorithm that shows *linear computational complexity O(n)* for uniformly distributed data sets and requires relatively little additional memory.
- Flash sort is an efficient in-place implementation of histogram sort, **itself a type of bucket sort.** It *assigns each of the n input elements to one of m buckets*, efficiently *rearranges the input* to place the buckets in the correct order, then sorts each bucket

STEP-BY-STEP DESCRIPTIONS:

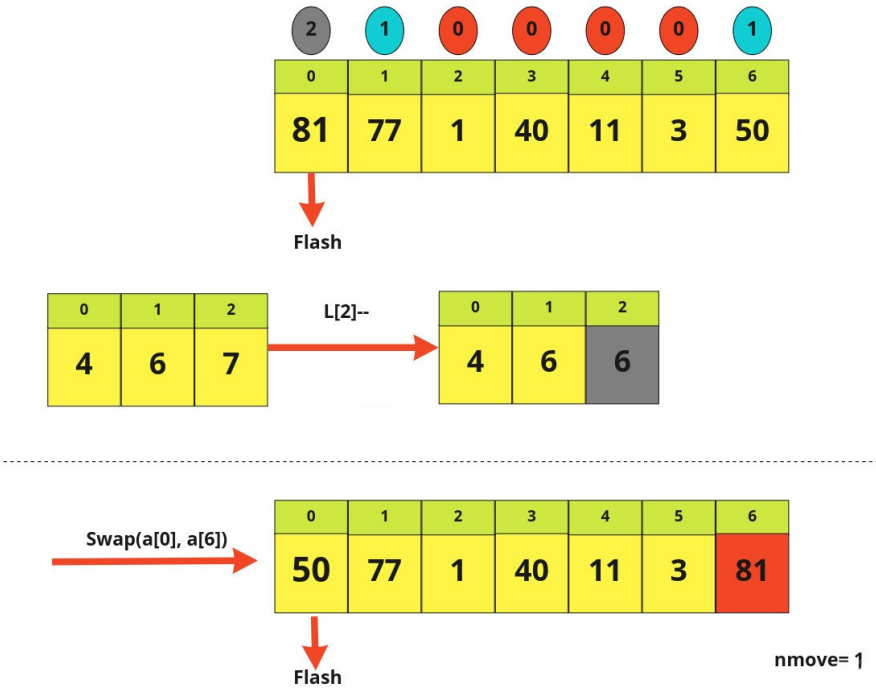
- Step 1: Find the minimum value of the elements in the array (minVal) and the position of the largest element in the array (max).



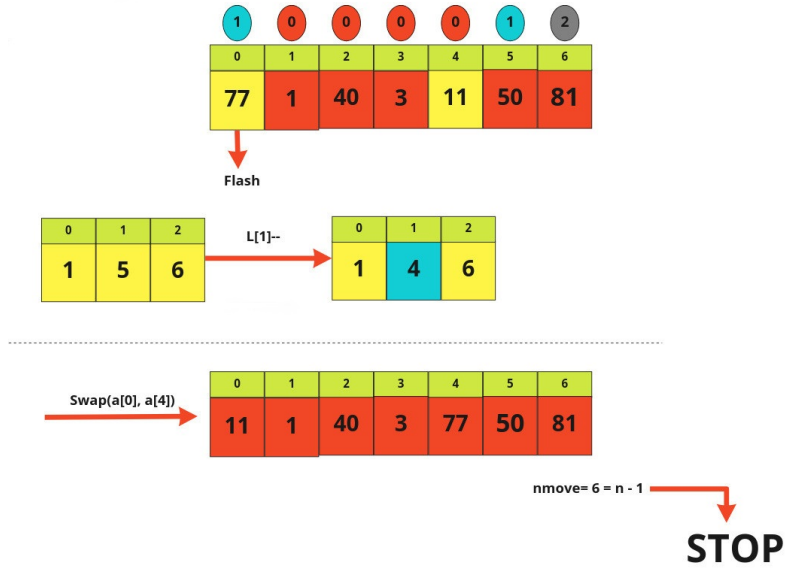
- Step 2: Initialize a vector L with m elements (corresponding to m classes, in this source code, the number of classes is chosen to be 0.45n).



- Step 3: Count the number of elements in each class according to the rule, element a[i] will belong to class k = int((m - 1) * (a[i]-minVal) / (a[max] - minVal)).



- Step 4: Calculate the end position of class j according to the formula L[j] = L[j] + L[j - 1] (j increases from 1 to m - 1).



COMPLEXITY EVALUATION:

Flash sort is a distribution sorting algorithm that shows linear computational complexity **'O(n)'** for uniformly distributed data sets and requires relatively little additional memory. The algorithm is based on in situ permutation and requires, in addition to the array holding the elements to be sorted, only a vector of a length equal to the number of different keys as an auxiliary memory.

Case	Time Complexity
Best	$O(n)$
Average	$O(n + r)$
Worst	$O(n^2)$

4 EXPERIMENTAL RESULT AND COMMENTS

We had organized the experimental results in the following figures, graphs, and tables below which are presented the resulting statistics (i.e., running times and number of comparisons) of each sorting algorithm. With a specific data arrangement, we hope that these reports could bring about the pictorial and visualized comparison between each sorting algorithm to conclude the most and least efficient sorts. For a better visualisation, please click on this link to enjoy a better experience!
CLICK HERE FOR A BETTER VISUALIZATION!

4.1 Randomized data

Data order: -rand												
Data size	10,000		30,000		50,000		100,000		300,000		500,000	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision
Selection Sort	49	100020001	456	900060001	1175	2500100001	4585	10000200001	90307	90000600001	244276	250001000001
Insertion Sort	35	50333111	321	450199000	843	1248987264	3271	4990744971	48631	44997841235	137220	124955310301
Bubble Sort	173	100009999	1632	900029999	5221	2500049999	22819	10000099999	306681	90000299999	863434	250000499999
Shaker Sort	174	67019271	1574	600438971	4474	1665955604	18372	6656602324	233423	60033572180	669758	166687832394
Shell Sort	1	640344	5	2346753	9	4325053	21	10197796	78	33672471	140	64089907
Heap Sort	1	637799	6	2150694	10	3771528	22	8045080	99	26486697	182	45969164
Merge Sort	120	573554	476	1907389	913	3332682	1154	7065764	2598	23082278	4373	39882853
Quick Sort	1	271856	3	902983	5	1533701	11	3173422	42	10599484	67	17727386
Counting Sort	1	60002	1	180002	1	282770	1	532770	5	1532770	8	2532770
Radix Sort	1	100153	1	360188	4	600188	6	1200188	14	3600188	25	6000188
Flash Sort	1	98458	1	293837	1	446961	2	863795	27	2577201	53	4229883

Figure 9: Running Times and numbers of comparisons of Random data arrangement

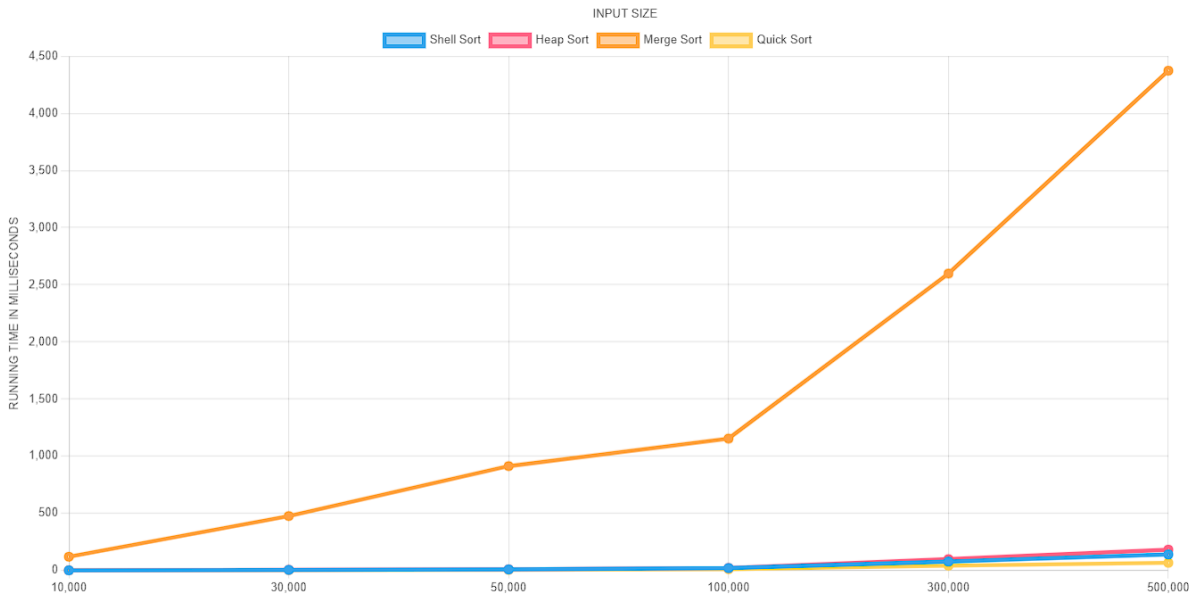


Figure 11: Randomized time for $O(n \log_2 n)$

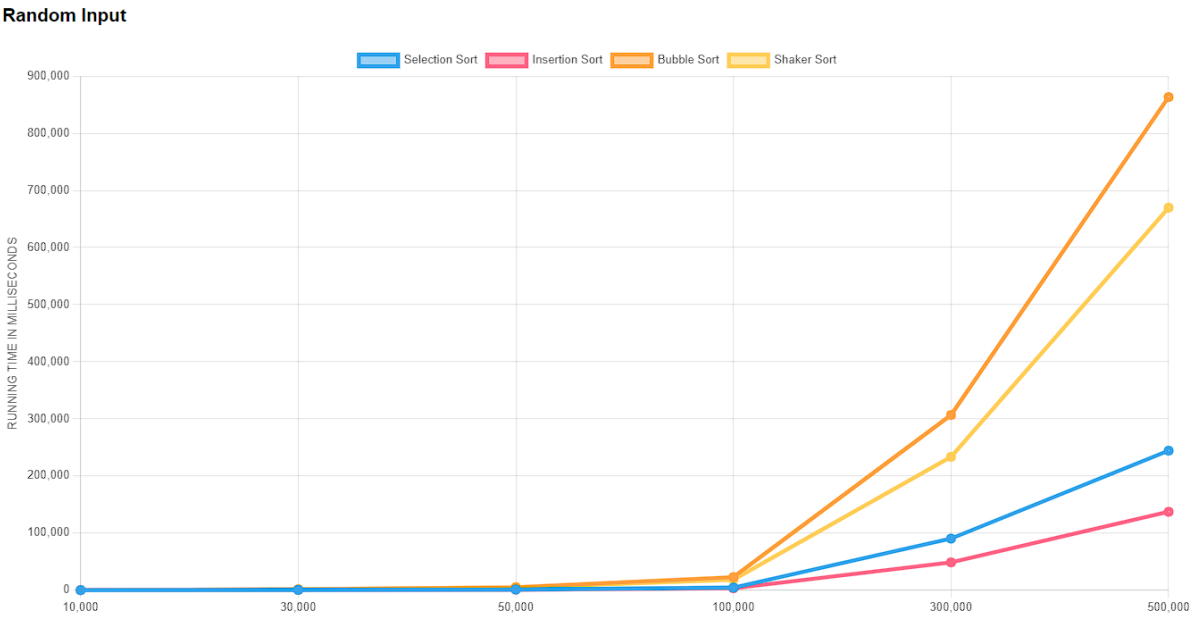


Figure 10: Randomized time for $O(n^2)$

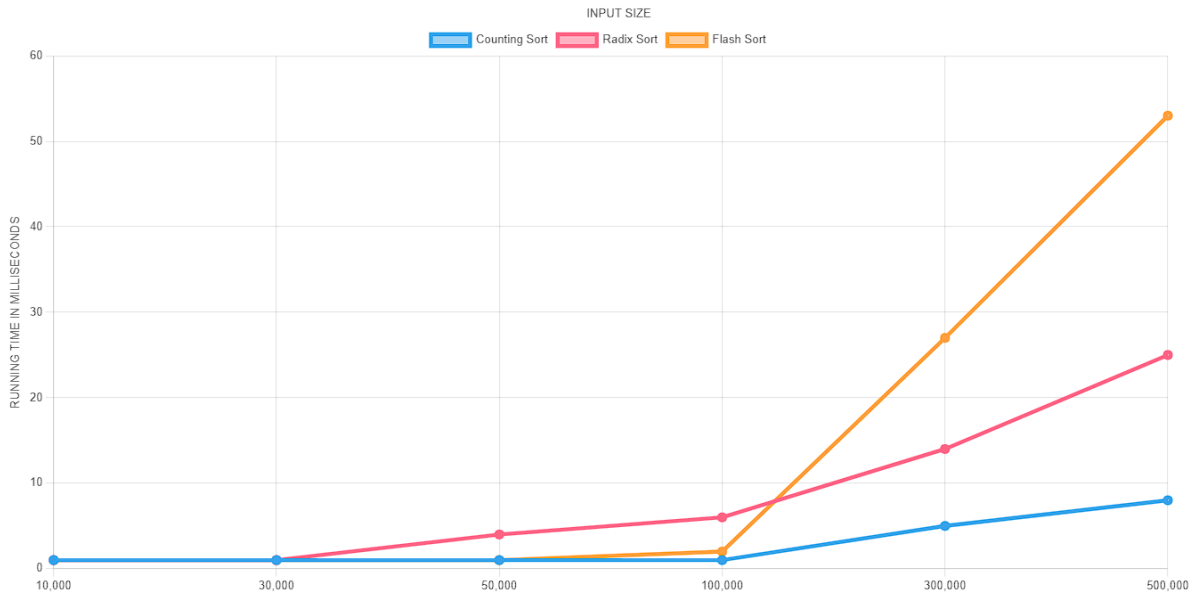


Figure 12: Randomized time for $O(n)$

- For smaller data sizes, such as 1*0,000 and 30,000*, **Selection Sort, Insertion Sort, and Bubble Sort** have relatively longer running times compared to other sorts such as **Shell Sort, Heap Sort, Quick Sort, Counting Sort, Radix Sort, and Flash Sort**. As the data size *increases to 50,000 and beyond,* the difference in running time between these sorts becomes more pronounced.

- Based on the table and graphs, the **fastest algorithms** in terms of RUNNING TIME are **Shell Sort, Heap Sort, Quick Sort, Counting Sort, Radix Sort and Flash Sort**. They all have running times *less than 0.2 seconds* for all data sizes. While the **Counting Sort, Radix Sort, Flash Sort and Quick Sort** are the ones that have the quickest time compiled in 500.000 data sized. The **slowest algorithms** are **Selection Sort, Insertion Sort, Bubble Sort and Shaker Sort**. They have running times *more than 10 seconds for data sizes of 300,000 and 500,000.*

- It can be concluded that for the given data order and sizes, **Shell Sort, Heap Sort, Quick Sort, Counting Sort, Radix Sort and Flash Sort** are the *most efficient sorting algorithms* in terms of running time.

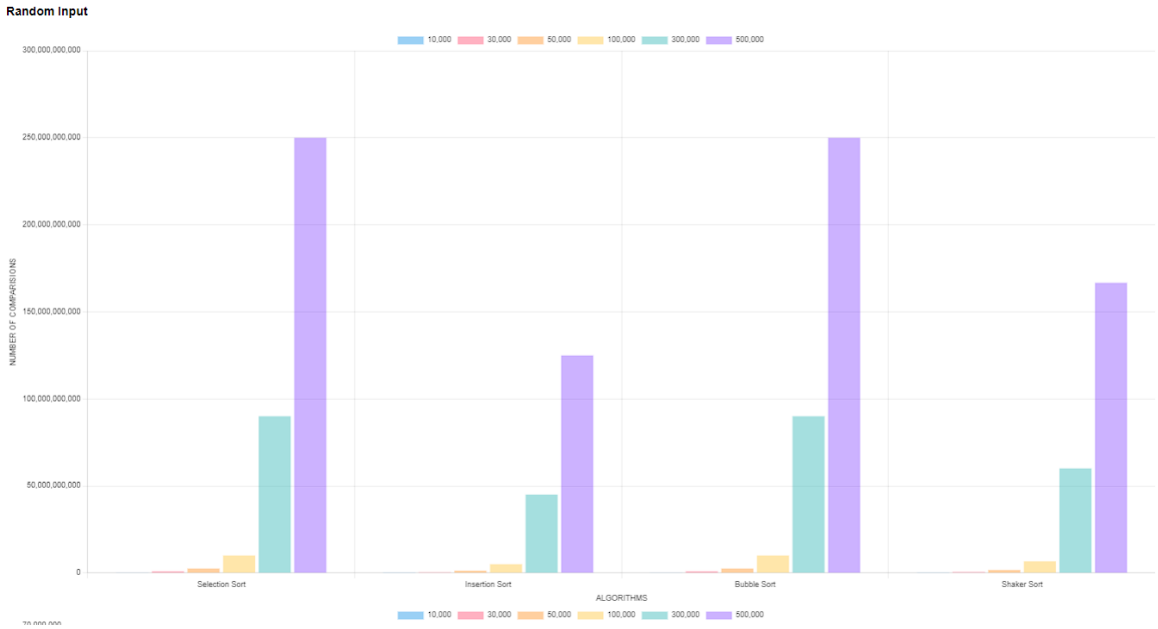


Figure 13: Randomized number of comparison for $O(n^2)$

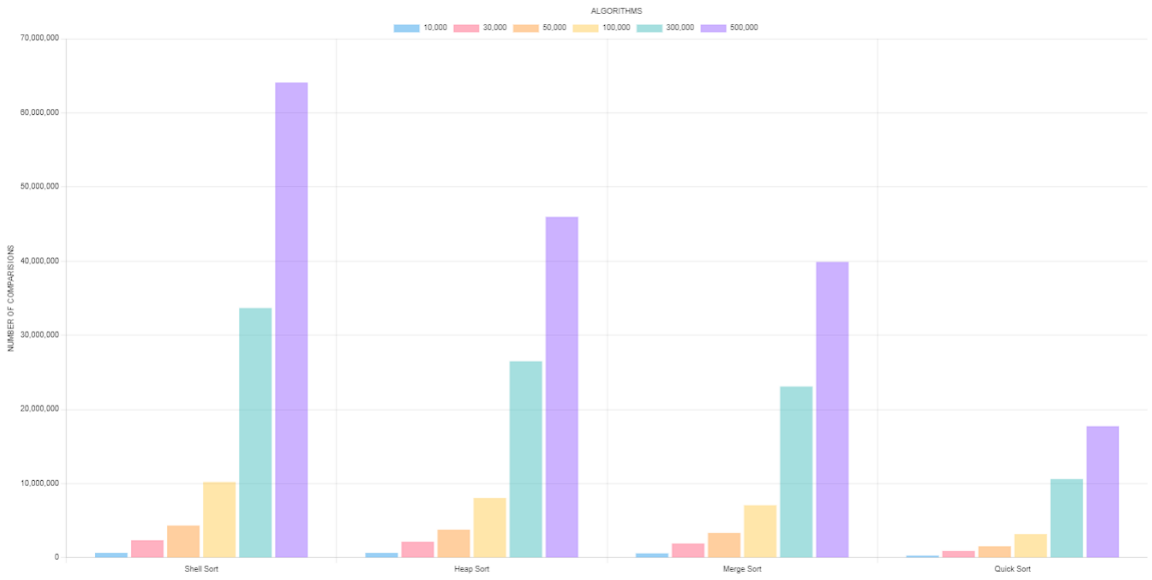


Figure 14: Randomized number of comparison for $O(n \log_2 n)$

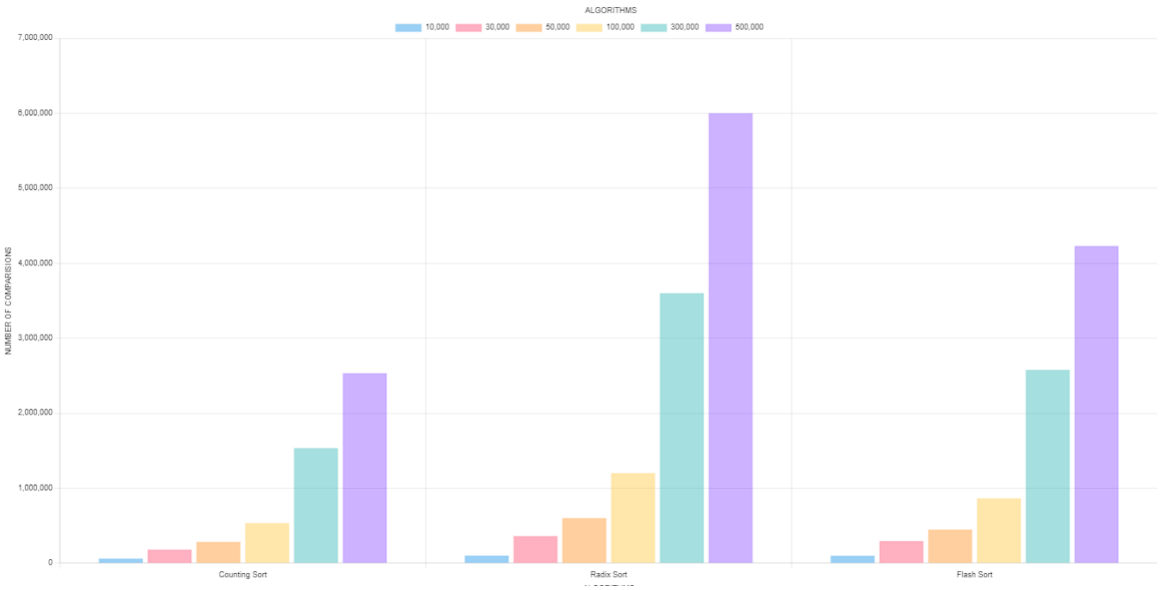


Figure 15: Randomized number of comparison for $O(n)$

- The algorithms with the **least NUMBER OF COMPARISONS** are **Counting Sort, Radix Sort and Flash Sort**. They have *less than 10 million comparisons* for all data sizes. The algorithms with the **most number of comparisons are Selection Sort, Insertion Sort and Bubble Sort.** They have *more than 100 billion comparisons* for data sizes of 300,000 and 500,000.

- The running time and number of comparisons of the algorithms seem to depend on the data order as well as the data size. For example, *Insertion Sort* performs **better on nearly sorted data than on randomized data,** while *Quick Sort* performs **worse on sorted data than on randomized data**.

- Overall, **the most efficient algorithms** on ***randomized data*** *are Shell Sort, Heap Sort, Quick Sort, Counting Sort, Radix Sort and Flash Sort.* They have low running time and number of comparisons regardless of the data size. The **least efficient algorithms** are *Selection Sort, Insertion Sort, Bubble Sort and Shaker Sort.* They have high running time and number of comparisons that increase dramatically with the data size.

- The **stable algorithms** are *Selection Sort, Insertion Sort, Bubble Sort, Shaker Sort and Merge Sort.* They preserve the relative order of equal elements in the input. The **unstable algorithms** are *Shell Sort, Heap Sort, Quick Sort, Counting Sort, Radix Sort and Flash Sort.* They may change the relative order of equal elements in the input.

4.2 Nearly sorted data

Data order: -nsorted												
Data size	10,000		30,000		50,000		100,000		300,000		500,000	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision
Selection Sort	142	100020001	1055	900060001	3046	2500100001	11259	10000200001	114413	90000600001	308738	250001000001
Insertion Sort	1	170758	1	447814	1	513430	1	781602	1	1381738	4	2023354
Bubble Sort	110	100009999	881	900029999	2434	2500049999	10528	10000099999	87807	90000299999	279723	250000499999
Shaker Sort	1	138230	1	537830	1	630684	2	803550	3	1739533	4	2371402
Shell Sort	1	413995	2	1326379	4	2211798	6	4682277	21	15469239	34	25640939
Heap Sort	2	669916	5	2236677	10	3925135	24	8364618	67	27413296	117	47405023
Merge Sort	100	491256	257	1617064	550	2758433	903	5730820	2730	18467640	5174	31634915
Quick Sort	1	149091	1	485570	2	881127	4	1862212	10	5889352	18	10048626
Counting Sort	1	60002	1	180002	1	300002	1	600002	4	300000	6	500000
Radix Sort	1	100153	1	360188	3	600188	5	1200188	18	4200223	29	7000223
Flash Sort	1	123461	1	370463	1	617455	2	1234961	7	3704961	12	6174961

Figure 16: Running Times and numbers of comparisons of Nearly sorted data arrangement

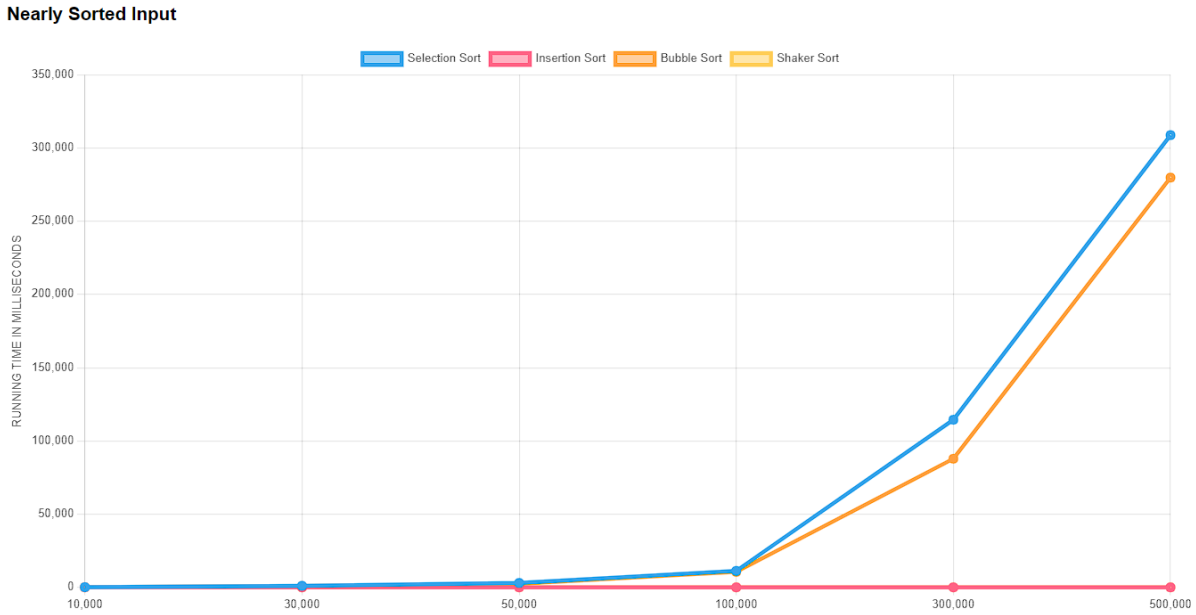


Figure 17: Nearly sorted time for $O(n^2)$

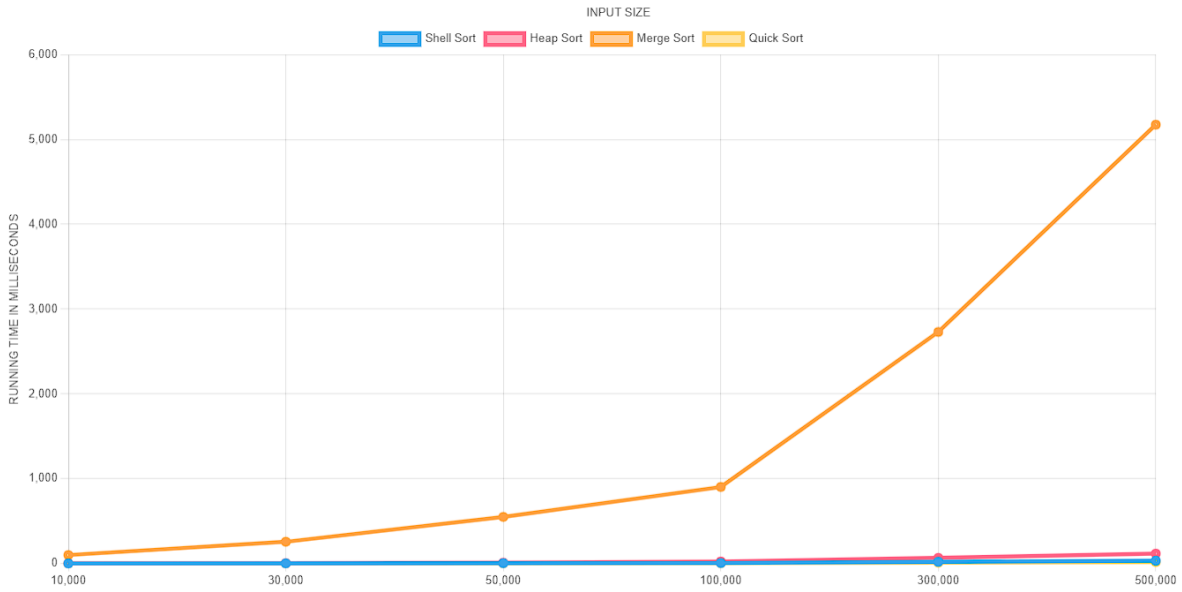


Figure 18: Nearly sorted time for $O(n \log_2 n)$

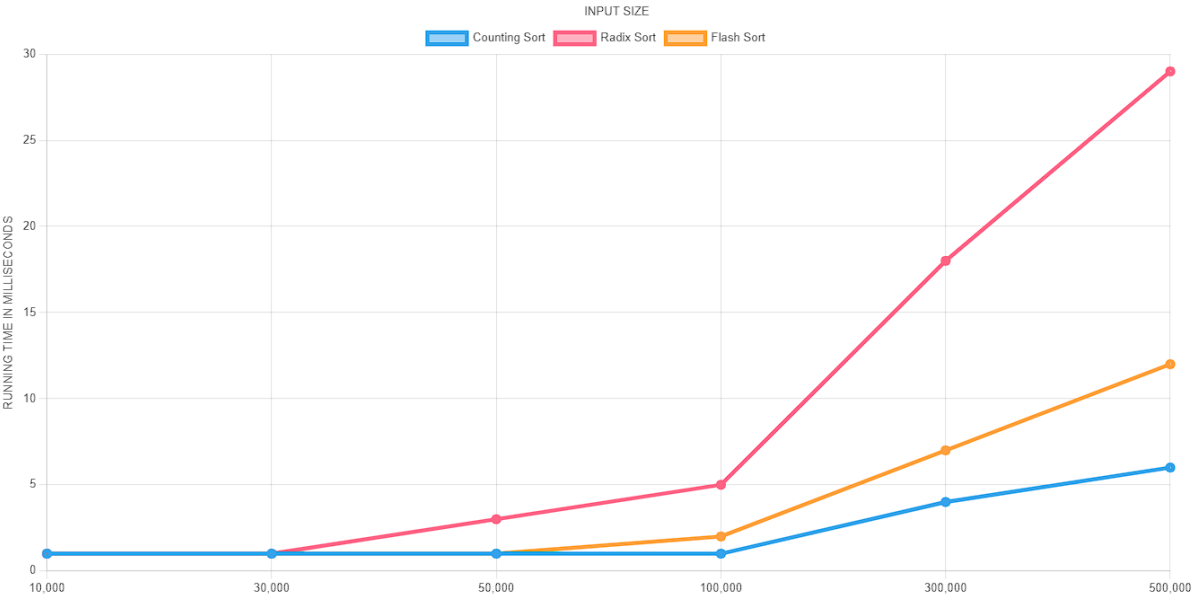


Figure 19: Nearly sorted time for $O(n)$

- For smaller data sizes, such as *10,000 and 30,000*, **Insertion Sort and Shaker Sort** have *very short running times*, while **Selection Sort and Bubble Sort** have relatively *longer running times*. This is because Insertion Sort and Shaker Sort are adaptive algorithms that perform well on nearly sorted data, while Selection Sort and Bubble Sort are not adaptive and perform poorly regardless of the data order.
- As the data size *increases to 50,000 and beyond*, the **difference in running time between the adaptive and non-adaptive algorithms becomes more pronounced**. Insertion Sort and Shaker Sort still have very short running times, while Selection Sort and Bubble Sort have much longer running times that increase dramatically with the data size.
- The other algorithms (**Shell Sort, Heap Sort, Merge Sort, Quick Sort, Counting Sort,**

Radix Sort and Flash Sort) have *relatively short running times for all data sizes.* Their running time increases slowly with the data size and is not affected much by the data order.

- To conclude, the **fastest algorithms** in terms of *RUNNING TIME* are ***Insertion Sort, Shaker Sort, Shell Sort, Heap Sort, Quick Sort, Counting Sort and Radix Sort.*** They all have running times less than 0.03 seconds for all data sizes. The **slower algorithms** are ***Selection Sort, Bubble Sort and Merge Sort***. They have running times more than 2 seconds for data sizes of 300,000 and 500,000.

Figure 20: Nearly sorted number of comparison for $O(n^2)$

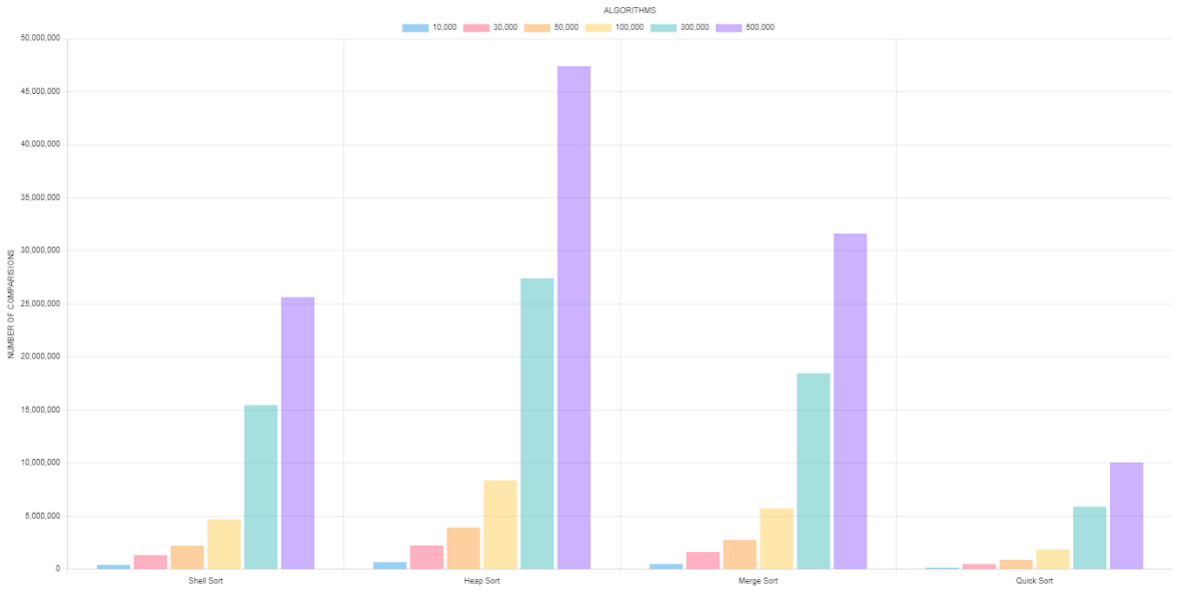


Figure 21: Nearly sorted number of comparison for $O(n \log_2 n)$

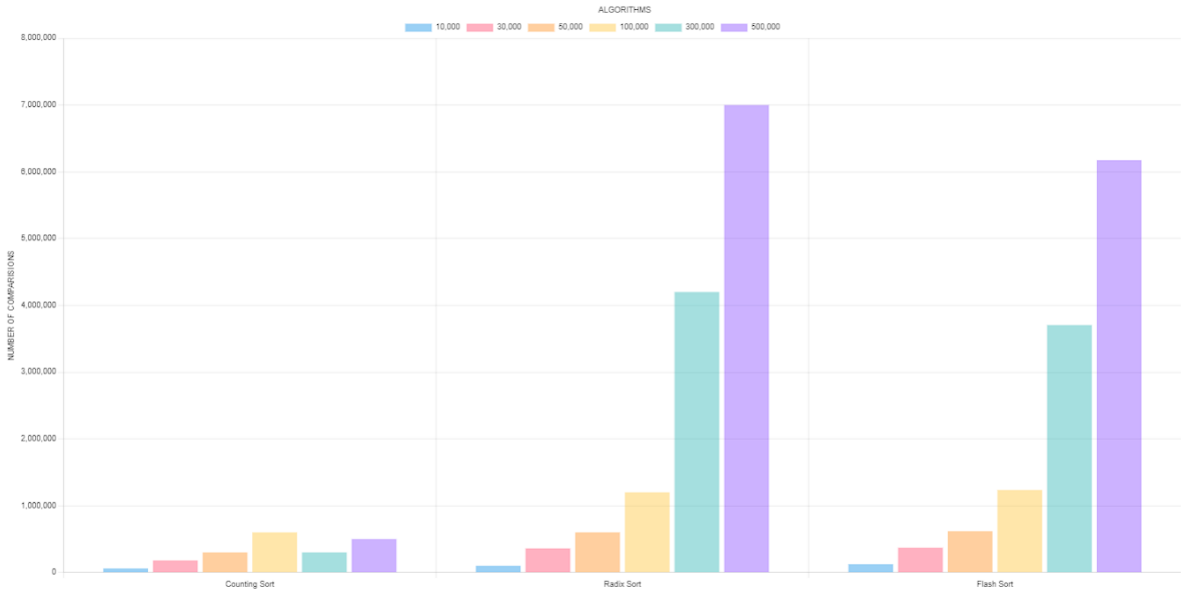


Figure 22: Nearly sorted number of comparison for $O(n)$

- Regarded to the table and charts, the **fastest algorithms** in terms of *RUNNING TIME* are ***Insertion Sort, Shaker Sort, Shell Sort, Heap Sort, Quick Sort, Counting Sort and Radix Sort.*** They all have running times less than 0.03 seconds for all data sizes. The **slowest algorithms** are ***Selection Sort, Bubble Sort and Merge Sort***. They have running times more than 2 seconds for data sizes of 300,000 and 500,000.

- The algorithms with the **least NUMBER OF COMPARISONS** are ***Insertion Sort, Shaker Sort, Counting Sort, Radix Sort and Flash Sort.*** They have less than 10 million comparisons for all data sizes. The algorithms with the **most number of comparisons** are ***Selection Sort, Bubble Sort and Merge Sort.*** They have more than 30 million comparisons for data sizes of 300,000 and 500,000.

- As mentioned before, the running time and number of comparisons of the algorithms seem to depend on the data order as well as the data size. For example, **Insertion Sort and Shaker Sort** perform *better on nearly sorted data than on randomized data,* while **Heap Sort and Quick Sort** perform *worse on nearly sorted data than on randomized data.*

- Overall, the **most efficient algorithms** on **nearly sorted data** are *Insertion Sort, Shaker Sort, Shell Sort, Quick Sort, Counting Sort and Radix Sort.* They have low running time and number of comparisons regardless of the data size. The **least efficient algorithms** are *Selection Sort, Bubble Sort and Merge Sort.* They have high running time and number of comparisons that increase dramatically with the data size.

- The **stable algorithms** are *Selection Sort, Insertion Sort, Bubble Sort, Shaker Sort and Merge Sort.* They preserve the relative order of equal elements in the input. The **unstable algorithms** are *Shell Sort, Heap Sort, Quick Sort, Counting Sort, Radix Sort and Flash Sort.* They may change the relative order of equal elements in the input.

4.3 Sorted data

Data order: -sorted												
Data size	10,000		30,000		50,000		100,000		300,000		500,000	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision
Selection Sort	148	100020001	1229	900060001	3279	2500100001	11149	10000200001	12673	90000600001	246244	250001000001
Insertion Sort	1	29998	1	89998	1	149998	1	299998	1	899998	4	1499998
Bubble Sort	103	100009999	873	900029999	2432	2500049999	9736	10000099999	88364	90000299999	272123	250000499999
Shaker Sort	1	20001	1	60001	1	100001	1	200001	1	600001	1	1000001
Shell Sort	1	360042	2	1170050	3	2100049	6	4500051	20	15300061	34	25500058
Heap Sort	2	670329	6	2236648	10	3925351	24	8365080	67	27413230	116	47404886
Merge Sort	101	465243	259	1529915	570	2672827	900	5645659	2711	18345947	5081	31517851
Quick Sort	1	149055	1	485546	2	881083	4	1862156	11	5889300	18	10048590
Counting Sort	1	60002	1	180002	1	300002	1	600002	4	1800002	6	3000002
Radix Sort	1	100153	1	360188	3	600188	5	1200188	19	4200223	29	7000223
Flash Sort	1	123491	1	370491	1	617491	2	1234991	7	3704991	12	6174991

Figure 23: Running Times and numbers of comparisons of Sorted data arrangement

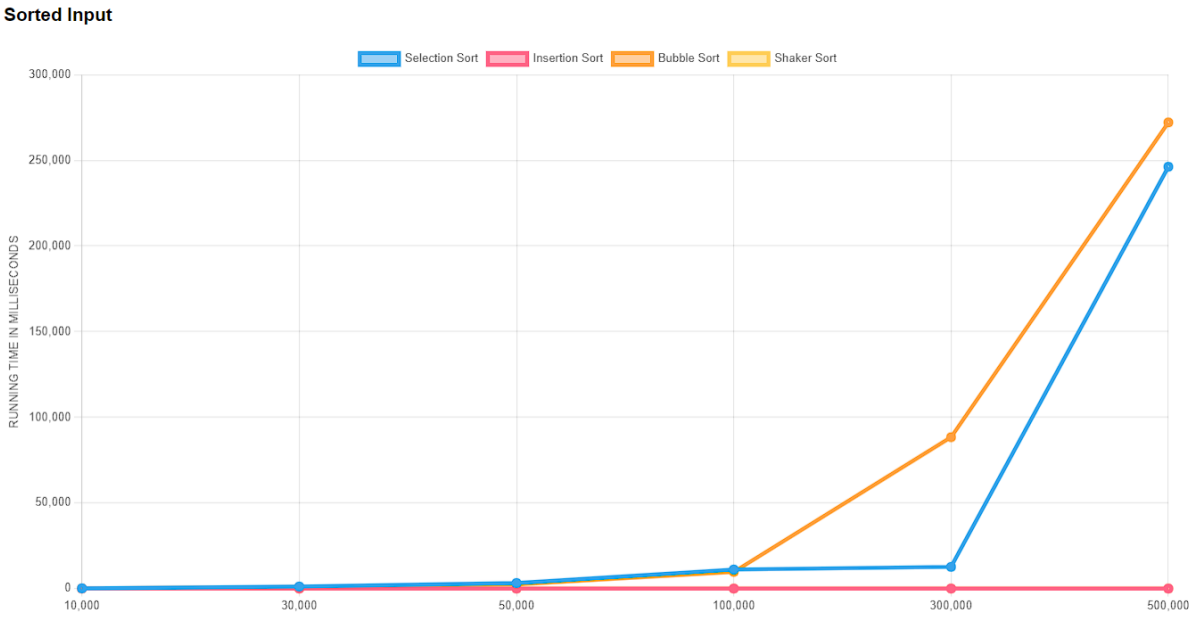


Figure 24: Sorted data time for $O(n^2)$

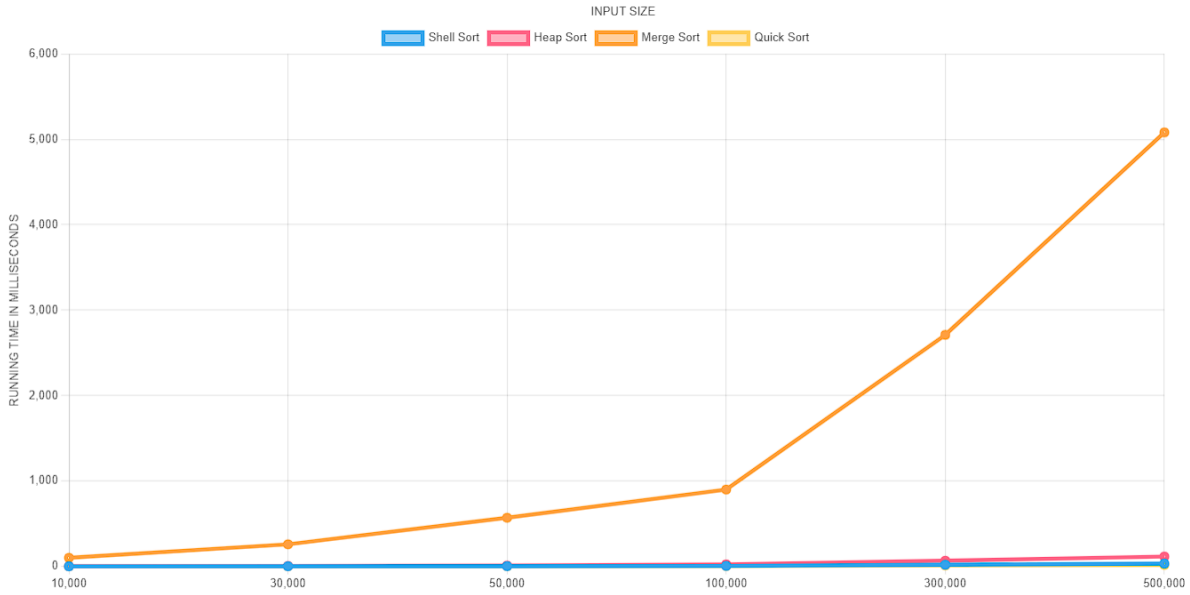


Figure 25: Sorted data time for $O(n \log_2 n)$

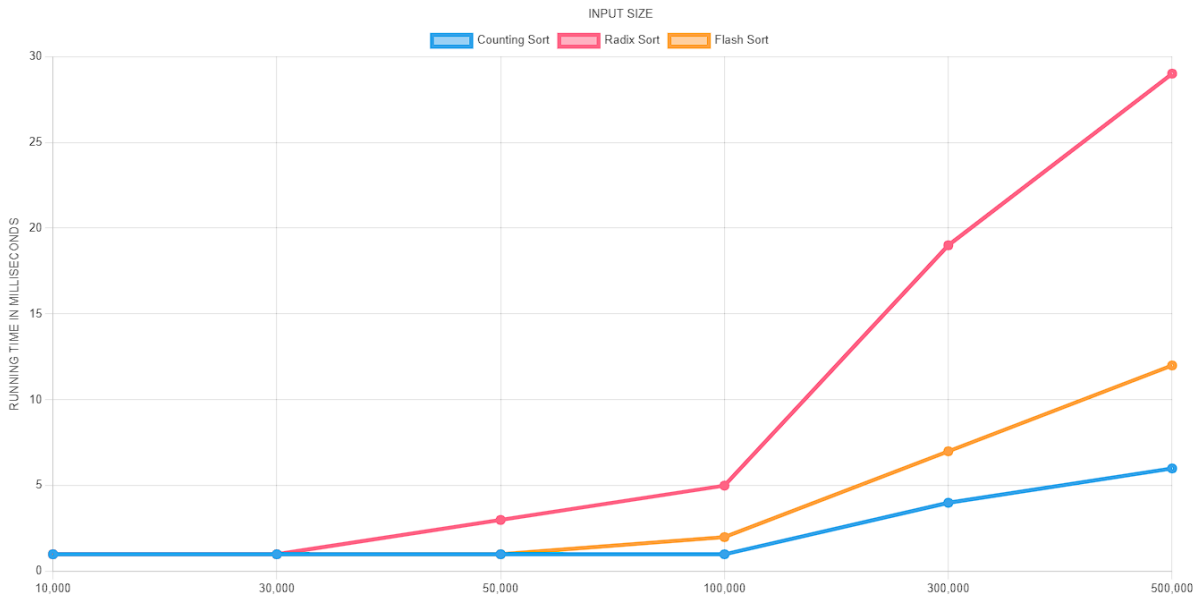


Figure 26: Sorted data time for $O(n)$

- As quite the same as Nearly sorted data, for s*maller data sizes, such as 10,000 and 30,000*, **Insertion Sort and Shaker Sort** have **very short running times,** while **Selection Sort and Bubble Sort** have relatively **longer running times**. This is because Insertion Sort and Shaker Sort are adaptive algorithms that perform well on sorted data, while Selection Sort and Bubble Sort are not adaptive and perform poorly regardless of the data order.
- As the data size increases to 50,000 and beyond, the difference in running time between the adaptive and non-adaptive algorithms becomes more pronounced. Insertion Sort and Shaker Sort still have very short running times, while Selection Sort and Bubble Sort have much longer running times that increase dramatically with the data size.
- The other algorithms (Shell Sort, Heap Sort, Merge Sort, Quick Sort, Counting Sort, Radix Sort

and Flash Sort) have relatively short running times for all data sizes. Their running time increases slowly with the data size and is not affected much by the data order.

- The **fastest algorithms** in terms of RUNNING TIME are ***Insertion Sort, Shaker Sort, Shell Sort, Heap Sort, Quick Sort, Counting Sort, Radix Sort and Flash Sort***. They all have running times less than 0.03 seconds for all data sizes. The **slowest algorithms** are ***Selection Sort, Bubble Sort and Merge Sort***. They have running times more than 2 seconds for data sizes of 300,000 and 500,000.

Figure 27: Sorted data number of comparison for $O(n^2)$

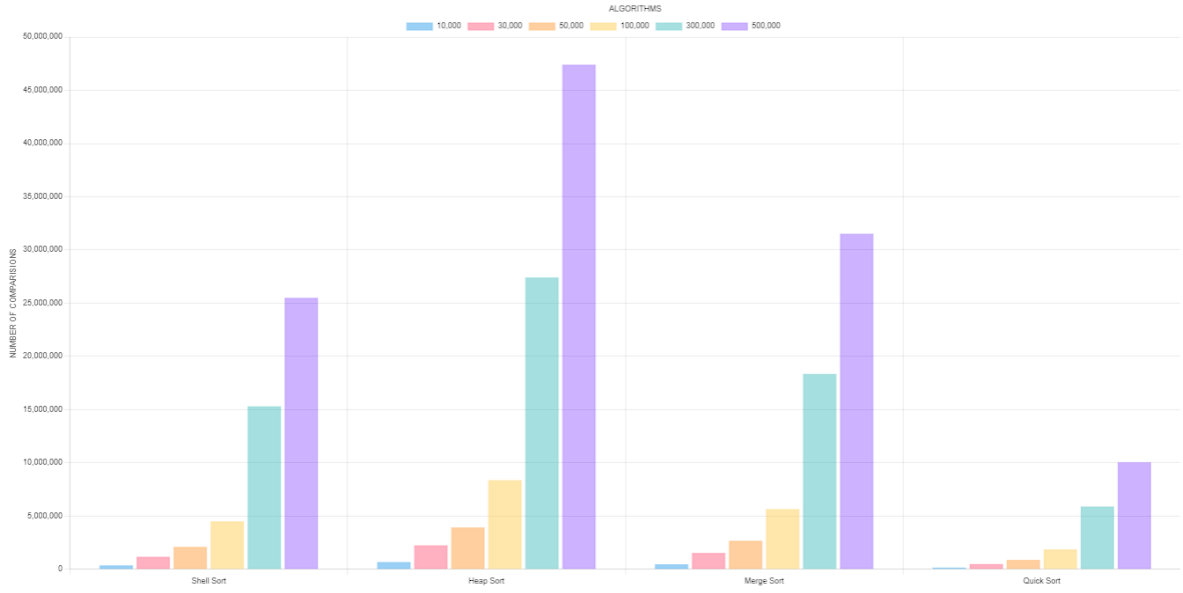


Figure 28: Sorted data number of comparison for $O(n \log_2 n)$

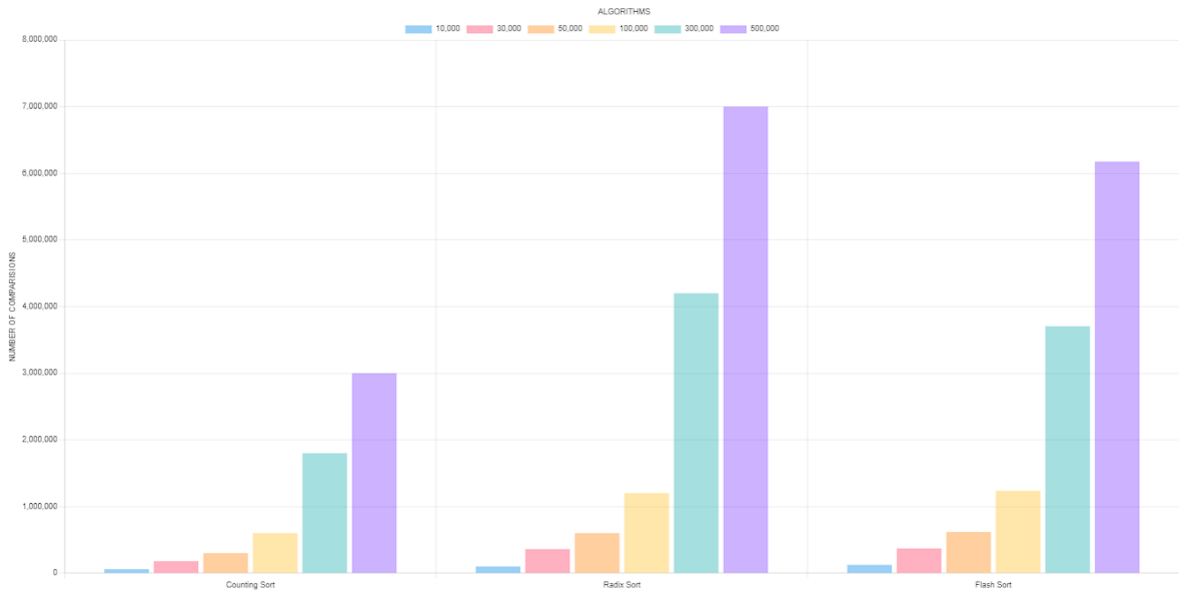


Figure 29: Sorted data number of comparison for $O(n)$

- The algorithms with the **least NUMBER OF COMPARISONS** are***Insertion Sort, Shaker Sort, Counting Sort and Radix Sort***. They have less than 10 million comparisons for all data sizes. The algorithms with the **most number of comparisons** are ***Selection Sort, Bubble Sort and Merge Sort***. They have more than 30 million comparisons for data sizes of 300,000 and 500,000.

- In this case, as quite the same as Nearly sorted data, **Insertion Sort and Shaker Sort** perform *better on sorted data than on randomized data*, while **Heap Sort and Quick Sort** perform *worse on sorted data than on randomized data.*

- This is because **Insertion Sort and Shaker Sort** takes advantage of the partially sorted nature of the input data. When the data is already sorted, each element is already in its correct position, so the algorithm only needs to perform a constant number of comparisons for each element.

- Overall, the **most efficient algorithms on sorted data** are ***Insertion Sort, Shaker Sort, Shell Sort, Quick Sort, Counting Sort and Radix Sort.*** They have low running time and number of comparisons regardless of the data size. The **least efficient algorithms** are ***Selection Sort, Bubble Sort and Merge Sort.*** They have high running time and number of comparisons that increase dramatically with the data size.

- The **stable algorithms** are *Selection Sort, Insertion Sort, Bubble Sort, Shaker Sort and Merge Sort*. They preserve the relative order of equal elements in the input. The **unstable algorithms** are *Shell Sort, Heap Sort, Quick Sort, Counting Sort, Radix Sort and Flash Sort*. They may change the relative order of equal elements in the input.

4.4 Reverse sorted data

Data order: -rev												
Data size	10,000		30,000		50,000		100,000		300,000		500,000	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision
Selection Sort	145	100020001	1058	900060001	3174	2500100001	10648	10000200001	123807	90000600001	248218	250001000001
Insertion Sort	110	100009999	989	900029999	2685	2500049999	11463	10000099999	97091	90000299999	285622	250000499999
Bubble Sort	371	100009999	2900	900029999	8075	2500049999	32341	10000099999	294657	90000299999	826428	250000499999
Shaker Sort	339	100005000	3300	900015000	8286	2500025000	34555	10000050000	300813	90000150000	873368	250000250000
Shell Sort	1	475175	2	1554051	4	2844628	8	6089190	25	20001852	43	33857581
Heap Sort	1	606771	7	606771	10	3612724	20	7718943	64	25569379	112	44483348
Merge Sort	100	466442	277	1543466	542	2683946	910	5667898	2677	18408314	5152	31836410
Quick Sort	1	159070	1	515556	2	931094	4	1962168	11	6189320	18	10548604
Counting Sort	1	60002	1	180002	1	300002	1	600002	4	1800002	6	3000002
Radix Sort	1	100153	1	360188	3	600188	5	1200188	17	4200223	29	7000223
Flash Sort	1	106000	1	318000	1	530000	2	1060000	7	3180000	12	5300000

Figure 30: Running Times and numbers of comparisons of Reversed data arrangement

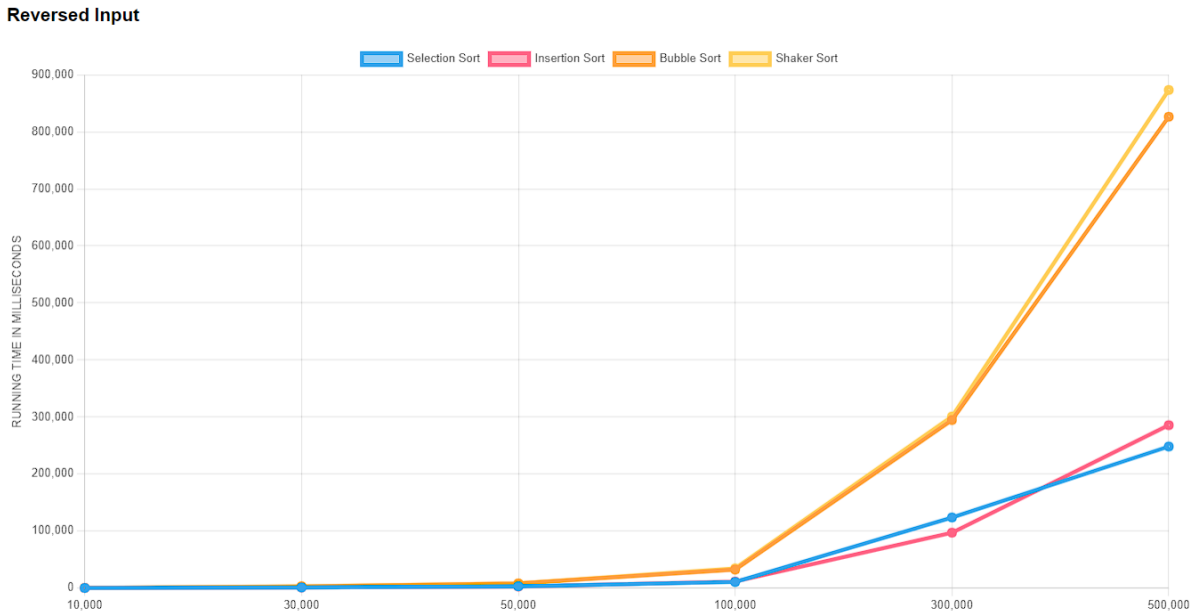


Figure 31: Reversed data time for $O(n^2)$

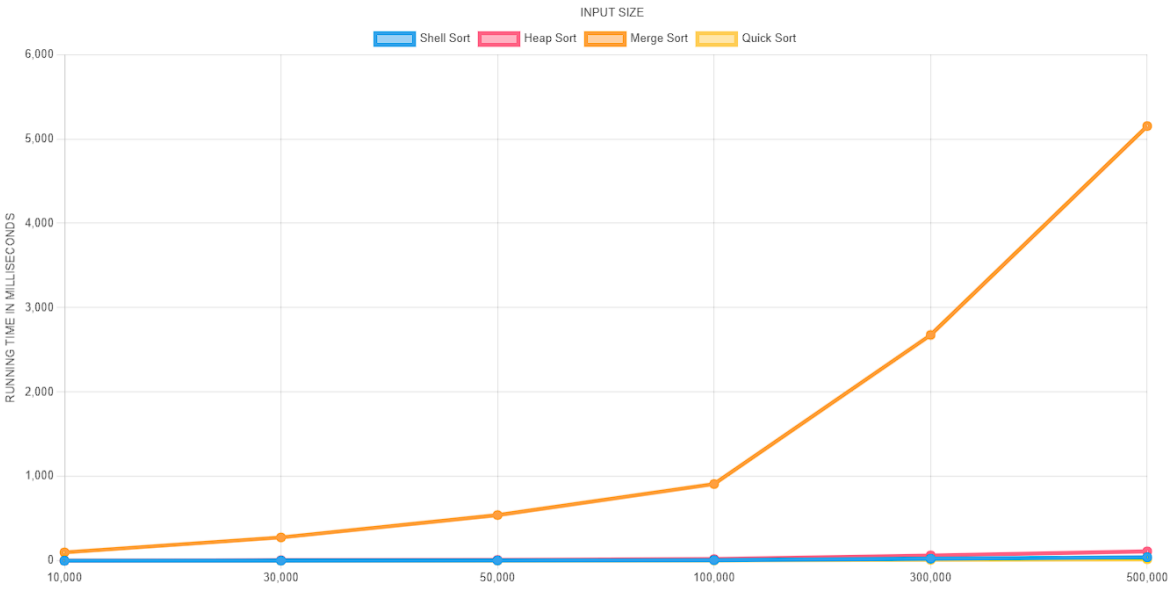


Figure 32: Reversed data time for $O(n \log_2 n)$

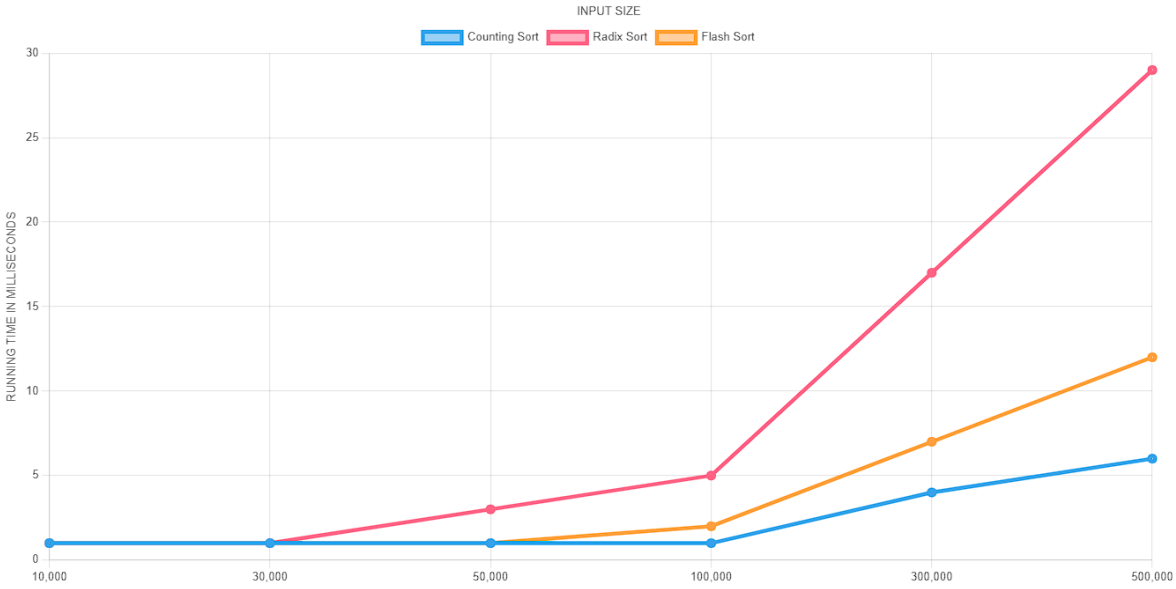


Figure 33: Reversed data time for $O(n)$

- When the data is *reverse sorted*, **Insertion Sort, Shaker Sort, Bubble Sort** has to shift every element to the right when inserting a new element, which **is the worst-case** scenario for this algorithm. That is the reason why these sorting algorithms are **the least efficient.**
- For reversed input data, **Shell Sort, Heap Sort, Quick Sort, Counting Sort, Radix Sort, and Flash Sort** are **the fastest algorithms** in terms of running time. These algorithms are based on dividing and conquering the data set into smaller subproblems and combining the solutions. They have the *lowest running time and number of comparisons for all data sizes.* These algorithms are efficient for uniformly distributed data sets or data sets with a known range of values.

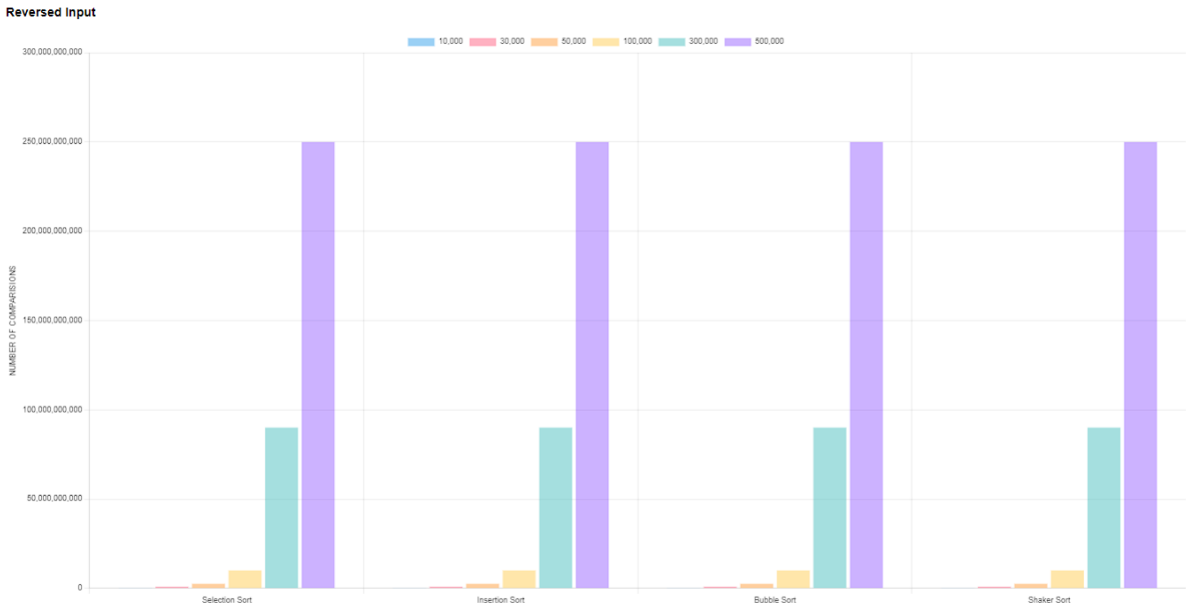


Figure 34: Reversed data number of comparison for $O(n^2)$

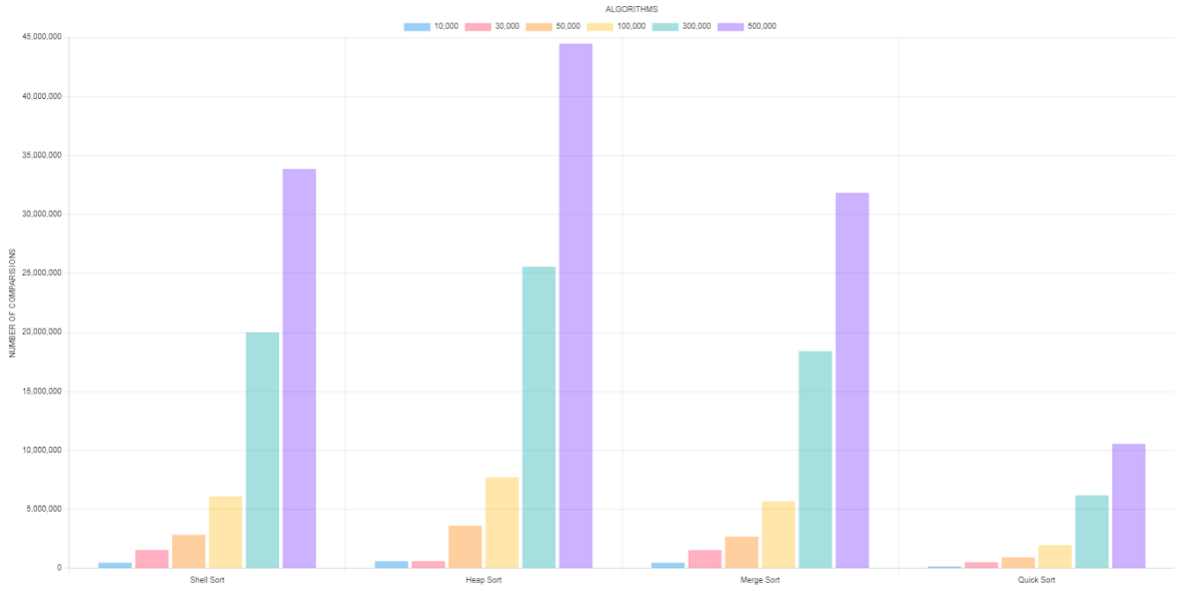


Figure 35: Reversed data number of comparison for $O(n \log_2 n)$

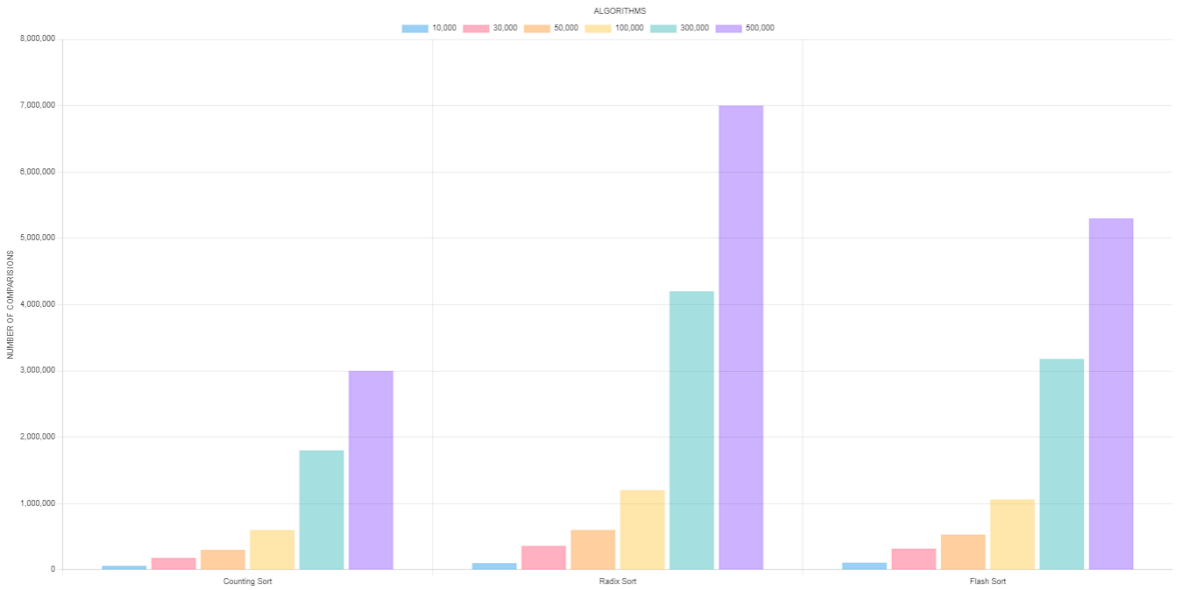


Figure 36: Reversed data number of comparison for $O(n)$

- The slowest algorithms overall are selection sort, insertion sort, bubble sort, and shaker sort. They have the highest running time and number of comparisons for all data sizes. These algorithms are inefficient for large or reverse data sets because they compare and swap elements one by one.
- The heap sort, merge sort, and quick sort algorithms are in the middle range of performance. They have moderate running time and number of comparisons for all data sizes. These algorithms are based on dividing and conquering the data set into smaller subproblems and combining the solutions.
- Overall, it appears that Shell Sort, Heap Sort, Quick Sort, Counting Sort, Radix Sort, and Flash Sort are the fastest algorithms for all data orders and sizes. These algorithms can be grouped as unstable algorithms. On the other hand, Selection Sort, Insertion Sort, Bubble Sort, and Shaker Sort are the slowest algorithms for all data orders and sizes. These algorithms can be grouped as stable algorithms. Merge sort is a stable algorithm that performs relatively well in terms of running time and number of comparisons for all data orders and sizes.

5 PROJECT ORGANIZATION AND PROGRAMMING NOTES

We organized our source codes into three main folders: algorithms, chart and tools. Each folder contains different files that serve different purposes for the sorting project.
The **algorithms folder** contains the header file "algorithm.h", which declares the prototypes of 11 sorting algorithms and a general function sort.cpp that calls these algorithms. It also contains 11 .cpp files, each implementing one sorting algorithm using C++. The algorithms are: selection-sort, insertion-sort, bubble-sort, shaker-sort, shell-sort, heap-sort, merge-sort, quick-sort, counting-sort, radix-sort and flash-sort. We used the standard C++ library ¡iostream¿ for input/output operations, ¡fstream¿ for file handling, ¡time.h¿ for measuring running time.

The **chart folder** contains the files for creating a web-based user interface for displaying the results of the sorting algorithms in graphical form. We intend to create charts from the JSON data that store the results of the sorting algorithms. The codes use the fetch API to get the data.json file and then parse it into a JavaScript object. The codes then loop through the different fields (time or comp), orders (rand, nsorted, sorted or rev) and algos (nn, nlogn or n) to create different charts for each combination. The codes use the Chart.js library to create line charts for the running time and bar charts for the number of comparisons. We also specify the labels, titles and scales for each chart. We append the charts to the HTML document using the getElementById method. It includes chart.jsm, which is a JavaScript module that uses the Chart.js library to create interactive charts from JSON data; data.json, which is a JSON file that stores the data for the charts; and style.css, which is a cascading style sheet file that defines the style and layout of the web page. The web page is index.html, which links to these files and displays the charts in a browser.

The **tools folder** contains two .cpp files: command.cpp (cmd.cpp) and dataGenerator.cpp. The command.cpp file parses the command-line arguments from the user and executes the appropriate mode (algorithm or comparison) and output parameters (time or comp or both) according to the specifications given in the project description. It also writes the input or output data to the corresponding files as required. The dataGenerator.cpp file generates input data of different sizes and orders (random, nearly sorted, sorted or reverse) using C++ random number generators and sorting functions.

We also included a main.cpp file in the root directory of the project, which simply calls the cmd.cpp (command file) file with the command-line arguments passed by the user. This is the execution file that the user needs to run to start the program.

Our team wrote some notes and comments in each source code file to explain the logic and functionality of each function or block of code. We also wrote a README.md file in Markdown format that gives an overview of the project, its objectives, requirements, structure and instructions on how to compile and run it. Some screenshots and tables of the console output and web interface are included to demonstrate how the program works.

6 CONCLUSION

The project consisted of three main parts: algorithm implementation, experiment execution and result visualization. The algorithm implementation involved writing header and source files for each sorting algorithm and a general function that calls them. The experiment execution involved writing a command-line interface that parses the user arguments and executes the appropriate mode and output parameters. It also involved writing a data generator that creates input arrays of different orders and sizes using C++ random number generators and sorting functions. The result visualization involved writing a web-based user interface that displays the running time and number of comparisons of each algorithm on each data order and size in interactive charts using JavaScript and Chart.js library.

The project demonstrated the strengths and weaknesses of each sorting algorithm on different data scenarios. It showed that some algorithms performed better on certain data orders or sizes than others. It also showed the trade-off between running time and number of comparisons of each algorithm. The project was a valuable learning experience that enhanced the understanding of sorting algorithms and their performance analysis. It also improved the skills of C/C++ programming, command-line interface design, data generation, web development and chart creation.

We would like to thank the tutor/professor for reading this article and giving us feedback on my work. We appreciate the opportunity to learn from this project and improve my knowledge and skills.

A LIST OF REFERENCES

- Algorithms Theory:

  1 Pr. Nguyen Thanh Phuong (2020) "Lecture notes of CS163 – Data structures" University of Science - Vietnam National University HCMC.

  2 Pr. Nguyen Ngoc Thao (2023) "Lecture notes of CSC10004 – Data structures" University of Science - Vietnam National University HCMC.

  3 Pr. Van Chi Nam (2019) "Lecture notes of CSC14004 – Data structures and algorithms" University of Science - Vietnam National University HCMC.

  4 Frank M. Carrano, Robert Veroff, Paul Helman (2014) "Data Abstraction and Problem Solving with C: Walls and Mirrors" Sixth Edition, Addion-Wesley. Chapter 11.

  5 Geeks for geeks:

    * `https://www.geeksforgeeks.org/cocktail-sort/`

    * `https://en.wikipedia.org/wiki/Cocktail_shaker_sort`

    * `https://www.geeksforgeeks.org/radix-sort-vs-bucket-sort/?ref=gcse`

  6 Others:

  – `sortvisualizer.com`

  – Sorts:

    * `https://www.programiz.com/dsa/counting-sort`

    * `https://github.com/HaiDuc0147/sortingAlgorithm`