REPORT: SORTING

# Information

**Course:** Cấu trúc dữ liệu và giải thuật-CSC10004\_22CLC02

**Teacher:**

Văn Chí Nam

Trần Thị Thảo Nhi

Bùi Huy Thông

Lê Thanh Tùng

**Project name:** Lab 3: Sorting

**Group ID:** 22127057\_22127064\_22127095\_22127131

**Group members’ information:**

|  |  |  |
| --- | --- | --- |
| Full Name | Student ID | Email |
| Đỗ Phan Tuấn Đạt | 22127057 | [dptdat22@clc.fitus.edu.vn](mailto:dptdat22@clc.fitus.edu.vn) |
| Phạm Thành Đạt | 22127064 | [ptdat22@clc.fitus.edu.vn](mailto:ptdat22@clc.fitus.edu.vn) |
| Đỗ Đình Hải | 22127095 | [ddhai22@clc.fitus.edu.vn](mailto:ddhai22@clc.fitus.edu.vn) |
| Trần Nguyễn Minh Hoàng | 22127131 | [tnmhoang22@clc.fitus.edu.vn](mailto:tnmhoang22@clc.fitus.edu.vn) |

# Introduction

## About the project

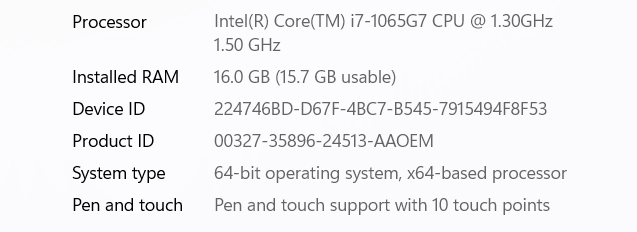
This project was carried out as part of the team’s education in the Data Structure and Algorithm course, specifically about sorting algorithms and their strengths/weaknesses.

The final product of the project consists of:

* Source code and executable file of the program written in C++, made by the team, for the purpose of caring out experiments.
* PDF file containing information about the project and team members, as well as research data and results from experiments.
* Excel file used for grading.

All referenced articles/websites will be credited at the end of this file.

## Specifications of device used for experimenting



\*Note: Device is set to run at best performance mode.

# Algorithm presentation

## Selection sort

### Idea

Separate the data set into 2 sections: sorted and unsorted. Initially, the sorted section will be empty, and the unsorted will contain the whole data set. Then, for each unsorted element, we choose the smallest, add it to the sorted section and remove it from the unsorted

### Step-by-step description

* First, for each element in the list, we will call this “selected element”, select the smallest in range from selected element to the end of the list.
* Second, swap the smallest element found in the above step with the selected element.
* Third, repeat the two steps above until the selected element is the penultimate item in the list.

A diagram of a swapping elements

Description automatically generated

A diagram of a swapping process

Description automatically generated

**A diagram of a number

Description automatically generated**

A diagram of a number

Description automatically generated

**A green rectangular box with numbers

Description automatically generated**

### Complexity evaluation

* Time complexity:
  + Worst case: O(n^2)
  + Average case: O(n^2)
  + Best case: O(n^2)
* Space complexity: O(1)

### Variations/Improvements

Instead of only choosing the smallest element, also select the largest element, then swap the smallest element with the selected element and swap the largest element with element at n – 1 – selected element’s position.

## Insertion sort

### Idea

Sort an array of size N in ascending order iterate over the array and compare the current element (key) to its predecessor, if the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

### Step-by-step description

Consider an array: {13, 12, 14, 6, 7}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 13 | 12 | 14 | 6 | 7 |

Step 1: First 2 elements of the array are compared in insertion sort

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 13 | 12 | 14 | 6 | 7 |

* 13 is greater than 12 so they are not in ascending order and 13 is not at it’s correct position. Swap 13 and 12.
* So 12 is stored in a sorted array

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 12 | 13 | 14 | 6 | 7 |

Step 2: Move to the next 2 elements and compare them

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 12 | 13 | 14 | 6 | 7 |

* Here 14 is greater than 13, both elements are in ascending order, so no swapping. 13 is also stored in a sorted array along with 11

Step 3:

* Move to the next two elements which are 14 and 6

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 12 | 13 | 14 | 6 | 7 |

* Both 14 and 6 are not at their correct place so swap them

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 12 | 13 | 6 | 14 | 7 |

* After swapping, elements 13 and 6 are not sorted, then swap again

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 12 | 6 | 13 | 14 | 7 |

* Here, 12 and 6 are not sorted, swap again

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 6 | 12 | 13 | 14 | 7 |

* Here, 6 is at correct position

Step 4:

* Move to the next two elements 14 and 7

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 6 | 12 | 13 | 14 | 7 |

* Swap between both

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 6 | 12 | 13 | 7 | 14 |

* 7 is smaller than 13, swap again

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 6 | 12 | 7 | 13 | 14 |

* 7 is smaller than 12, swap again

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 6 | 7 | 12 | 13 | 14 |

* Finally, the array is completely sorted.

### Complexity evaluation

* Time complexity:
  + Worst case: O(n^2)
  + Average case: O(n^2)
  + Best case: O(n)
* Space complexity: O(1)

### Variants/Improvements

* Binary insertion sort is a sorting algorithm which is similar to the [insertion sort](http://www.geeksforgeeks.org/insertion-sort/), but instead of using linear search to find the location where an element should be inserted, we use [binary search](https://www.geeksforgeeks.org/binary-search/). Thus, we reduce the comparative value of inserting a single element from O (N) to O (log N).
* Early Termination: During the insertion process, you can break out of the loop early if you find the correct position for the current element. This can prevent unnecessary comparisons when the array is almost sorted or has a lot of elements already in their correct positions.
* Move Instead of Swap: Instead of using swap operations to move elements in the sorted part of the array, you can use a single temporary variable to hold the element being inserted and shift elements in the sorted part by one position. This can be more efficient in terms of memory access and data movement.
* [Shell sort](http://en.wikipedia.org/wiki/Shellsort)is mainly a variation of [Insertion Sort](https://www.geeksforgeeks.org/insertion-sort/). In insertion sort, we move elements only one position ahead. When an element has to be moved far ahead, many movements are involved. The idea of ShellSort is to allow the exchange of far items. In Shell sort, we make the array h-sorted for a large value of h. We keep reducing the value of h until it becomes 1. An array is said to be h-sorted if all sublists of every h’th element are sorted.
* Adaptive Insertion Sort: Adaptive insertion sort is an enhancement that takes advantage of the partially sorted nature of the array. If the array is already partially sorted, the number of comparisons and shifts can be reduced.
* Combining Insertion Sort with Other Sorting Algorithms: Insertion sort can be used in combination with other sorting algorithms, such as merge sort or quicksort, for small sub-arrays. This hybrid approach can take advantage of the strengths of each sorting algorithm.

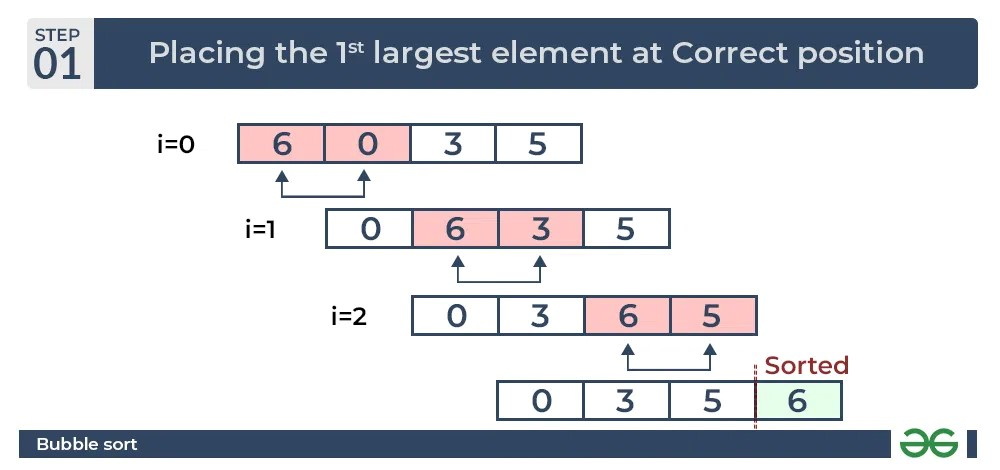
## Bubble sort

### Idea

Bubble sort is a basic algorithm for arranging a string of numbers or other elements in the correct order. The method works by examining each set of adjacent elements in the string, from left to right, switching their positions if they are out of order. The algorithm then repeats this process until it can run through the entire string and find no two elements that need to be swapped.

### Step-by-step description

Consider Input: arr[] = {6, 3, 0, 5}



Explanation:

First loop: Repeat n-1 times (n = number of elements in the array, in this case, n = 4)

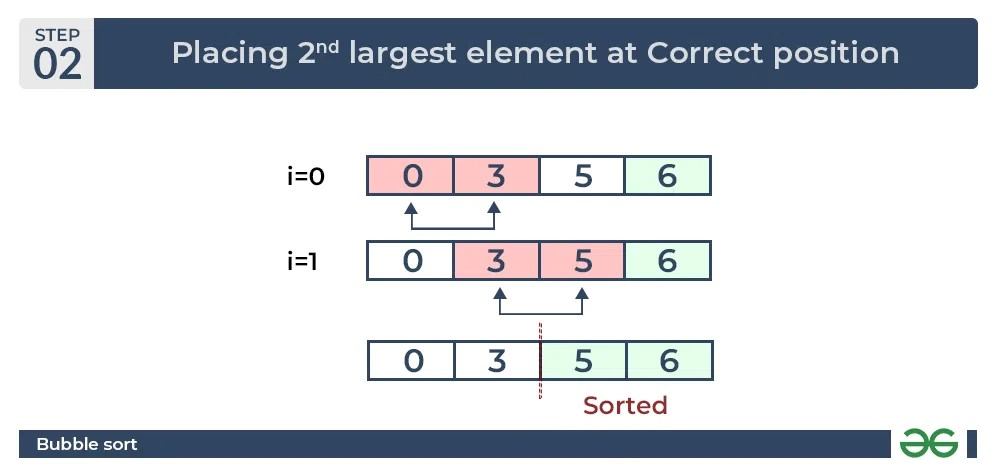
* Compare adjacent elements from left to right.
* If the left element is greater than the right element, swap them.
* After each iteration, the largest unsorted element "bubbles up" to the end of the array.

Step 1: Compare 6 > 3, swap 6 and 3: {3, 6, 0, 5}

Step 2: Compare 6 > 0, swap 6 and 0: {0, 3, 6, 5}

Step 3: Compare 6 > 5, swap 6 and 5: {0, 3, 5, 6}

The largest element, 6, is now in its correct sorted position at the end of the array.



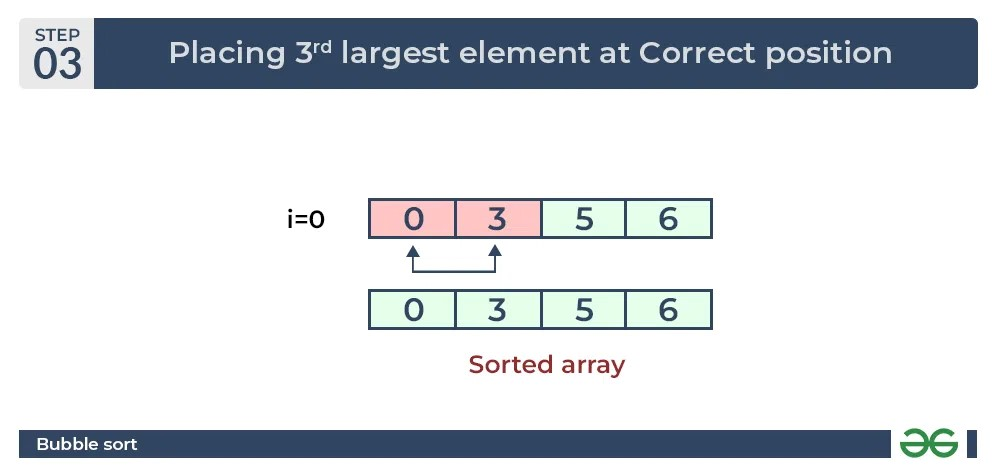
Second loop: Repeat n-2 times (as the largest element is already sorted, no need to consider it)

* Repeat the same process, but now the last element is in its correct sorted position.

Step 1: Compare 0 < 3, no swap, array remains unchanged: {0, 3, 5, 6}

Step 2: Compare 3 < 5, no swap, array remains unchanged: {0, 3, 5, 6}

The second-largest element, 5, is now in its correct sorted position.



Third loop: Repeat n-3 times (the last two elements are already sorted)

* Repeat the same process.

Step 1: Compare 0 < 3, no swap, array remains unchanged: {0, 3, 5, 6}

The remaining 2 elements, 0 and 3 are now in their correct sorted positions.

### Complexity evaluation

* Time complexity:
  + Worst case: O(n^2)
  + Average case: O(n^2)
  + Best case: O(n) – Due to a flag variable that is set to false before each pass and is set to true when swapping occurs (Improved version)
* Space complexity: O(1)

### Variants/Improvements

* **Odd-Even/Brick sort:** This algorithm is divided into two phases- Odd and Even Phase. The algorithm runs until the array elements are sorted and in each iteration two phases occurs- Odd and Even Phases. In the odd phase, we perform a bubble sort on odd indexed elements and in the even phase, we perform a bubble sort on even indexed elements.
* **Shaker/Cocktail sort:** Cocktail Sort, a variation of Bubble Sort, improves efficiency by traversing through the array bidirectionally. In each iteration, it performs two stages: the first stage moves from left to right, swapping adjacent elements to move the largest element to its correct position at the end of the array; the second stage moves from right to left, placing the second-largest element in its proper position. These stages alternate until the entire array is sorted.

## Shaker sort

### Idea

The Bubble sort algorithm always traverses elements from left and moves the largest element to its correct position in the first iteration and second-largest in the second iteration and so on, while Shaker Sort (variant of bubble sort) traverses through a given array in both directions alternatively. Shaker sort does not go through the unnecessary iteration making it efficient for large arrays.

Shaker sorts break down barriers that limit bubble sorts from being efficient enough on large arrays by not allowing them to go through unnecessary iterations on one specific region (or cluster) before moving onto another section of an array.

### Step-by-step description

Consider Input: arr[] = {8, 7, 4, 8, 1}

Left to right:

First loop: Repeat n-1 times (n = number of elements in the array, in this case, n = 5)

* Compare adjacent elements from left to right.
* If the left element is greater than the right element, swap them.
* After each iteration, the largest unsorted element "bubbles up" to the end of the array.

Step 1: Compare 8 > 7, swap 8 and 7: {7, 8, 4, 8, 1}

Step 2: Compare 8 > 4, swap 8 and 4: {7, 4, 8, 8, 1}

Step 3: Compare 8 = 8, no swap, array remains unchanged: {7, 4, 8, 8, 1}

Step 4: Compare 8 > 1, swap 8 and 1: {7, 4, 8, 1, 8}

The largest element, 8, is now in its correct sorted position at the end of the array.

Right to left:

Second loop: Repeat n-2 times (as the largest element is already sorted, no need to consider it)

* Compare adjacent elements from right to left.
* If the left element is greater than the right element, swap them.
* After each iteration, the smallest unsorted element "falls down" to the beginning of the array.

Step 1: Compare 1 < 8, swap 1 and 8: {7, 4, 1, 8, 8}

Step 2: Compare 1 < 4, swap 1 and 4: {7, 1, 4, 8, 8}

Step 3: Compare 1 < 7, swap 1 and 7: {1, 7, 4, 8, 8}

The smallest element, 1, is now in its correct sorted position at the beginning of the array.

Left to right:

Third loop: Repeat n-3 times (the last and first elements are already sorted)

* Repeat the same process.

Step 1: Compare 7 > 4, swap 7 and 4: {1, 4, 7, 8, 8}

Step 2: Compare 7 < 8, no swap, array remains unchanged: {1, 4, 7, 8, 8}

The second-largest element, 8, is now in its correct sorted position.

Right to left:

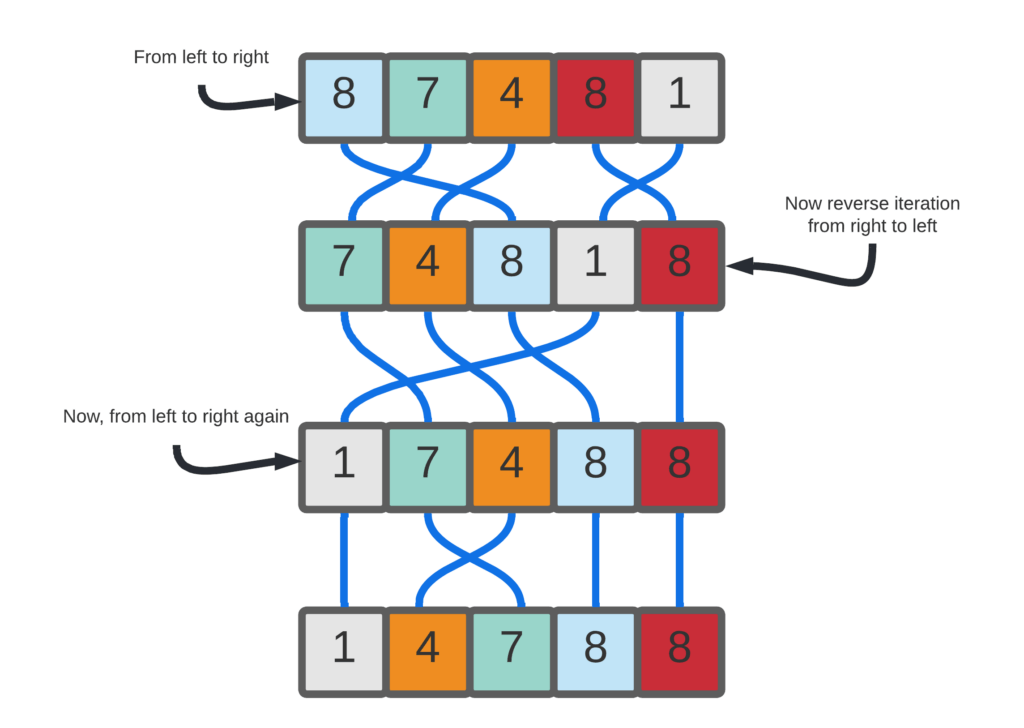
Fourth loop: Repeat n-4 times (the first and last two elements are already sorted)

* Repeat the same process.

Step 2: Compare 7 > 4, no swap, array remains unchanged: {1, 4, 7, 8, 8}

Step 3: Compare 4 > 1, no swap, array remains unchanged: {1, 4, 7, 8, 8}

Due to no swapping operations taking place, the array is now completely sorted.



### Complexity evaluation

* Time complexity:
  + Worst case: O(n^2)
  + Average case: O(n^2)
  + Best case: O(n)
* Space complexity: O(1)

## Shell sort

### Idea

Shell sort is an improvement of insertion sort. It first sort elements that have specific gaps, determined by the used increment rule used. It then reduces the gap recursively until the gap is one. The inner sorting method is insertion sort, customized by the interval or gap. If the gap becomes one, the inner sorting becomes the original insertion sort.

### Step-by-step description

* The interval first equals the size of the array divided by 2, then the interval will reduce itself by 2 at every iteration, until it becomes 1.
* We then sort the array/list using the insertion sort with the interval.
* Consider array {12, 34, 54, 2, 3}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 12 | 34 | 54 | 2 | 3 |

* First, we have the interval is 5/2 = 2. So, we sort {12, 54, 3} and {34,2} concurrently using insertion sort.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | 2 | 12 | 34 | 54 |

* Second, we will have the interval as 5/4 = 1, so we sort {3, 2, 12, 34, 54} using insertion sort.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 3 | 12 | 34 | 54 |

* Because we have reached interval= 1, we terminate the loop and end the sorting.

### Complexity evaluation

* Time complexity (Original):
  + Worst case: O(n^2).
  + Average case: O(nlogn), depend on the chosen sequence.
  + Best case: O(nlogn).
* Space complexity: O(1).

### Variations/Improvements

Because the interval/gap used in shell sort is not fixed, the algorithm can improve based on the increment/decrement sequence we use. Some common sequences:

* Knuth’s increments: 1, 4, 13, …, (3^k – 1)/ 2
* Sedgewick's increments: 1, 8, 23, 77, 281, 1073, 4193, 16577...4j+1+ 3·2j+ 1
* Hibbard's increments: 1, 3, 7, 15, 31, 63, 127, 255, 511…
* Papernov & Stasevich increment: 1, 3, 5, 9, 17, 33, 65, ...
* Pratt: 1, 2, 3, 4, 6, 9, 8, 12, 18, 27, 27, 16, 24, 36, 54, 81....

## Heap sort

### Idea

The array will first be converted into a Max-heap using the heapify operation. Next, the root node of the Max-heap will be sequentially removed and replaced with the last node in the heap. After each replacement, the root of the heap will be heapified to maintain the Max-heap property. This process will be repeated until the size of the heap becomes greater than 1.

### Step-by-step description

Consider the array: {5, 11, 4, 6, 2}

Step 1: Build complete binary tree from the array

A diagram of a line with numbers and circles

Description automatically generated

Step 2: Transform into max heap: to transform into max heap, the parent node should always be greater than the children nodes

* 5 is smaller than 11 so swap them

A black line with red numbers and a black line

Description automatically generated

* 5 is smaller than 6 so swap them to build max heap

A diagram of a diagram

Description automatically generated

* Now we have max heap

Step 3: Remove the root element (11) from the max heap. To achieve this, swap (11) node with the last node and remove it from the heap. Heapify the tree again to get another max heap

A black line with red numbers and a circle

Description automatically generated

Step 4: Repeat the above step

A black line with red numbers and a circle

Description automatically generated

Step 5: Remove the root (5) and perform heapify again

A black and white image of a long black stick

Description automatically generated

Step 6: Remove the root (4) and heapify again.

A black circle with red numbers

Description automatically generated

Step 7: Delete the last root and the sorted array will be:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 4 | 5 | 6 | 11 |

### Complexity evaluation

* Time complexity:
  + Worst case: O(n\*logn)
  + Average case: O(n\*logn)
  + Best case: O(n\*logn)
* Space complexity: O(1)

### Variants/Improvements

* Min Heap Sort: Beside the Heap Sort builds a max heap to sort in ascending oder, we can build a min heap to sort elements descending order.
* Iterative Heap Sort: the standard Heap Sort using recursion to heapify. However, an iterative version can be implemented using loops instead of recursion, which might be more memory-efficient and suitable for language or environments with limited stack space.
* Bottom-up Heap Sort: This variation starts by creating an array-based heap without using the standard heapify operation. Instead, it starts from the first non-leaf node and sifts down each element to build the heap. Afterward, it performs the sorting by repeatedly extracting the root and heapifying the remaining elements.
* D-ary Heap Sort: Basic Heap Sort uses Binary Heap, in which each node has 2 children. In D-ary Heap Sort, the heap is generalized to have at most D children for each node. This reduce the height of the heap, potentially improving performance.
* Parallel Heap Sort: in parallel computing environments, we can divide the heap into smaller heaps and sort them concurrently. After that, the sorted smaller heaps are merged to get the final sorted result, potentially reducing the sorting time.
* Adaptive Heap Sort: aims to optimize the Heap Sort performance for partially sorted or nearly sorted array. If the input is partially sorted and adjusts the heapify operation accordingly to reduce unnecessary work.
* In-place Heap Sort: In standard Heap Sort, it requires extra space for the heap data structure. However, with some modifications, it can be implemented in an in-place version that sorting without using additional memory for the heap.

## Merge sort

### Idea

Split the data set recursively into subsets, until there are all subsets with only one element. Then, every pair of adjacent subsets, merge them following this rule:

* If the current item in set 1 is larger than the current item in set 2, push the item in set 2 into a common array/list. If not, push the item in set 1.
* If either set is empty when the other is not, push the rest of the other set into the common list.

This also means that if the element number is not even, we are still able to merge them. By merging all available pairs of adjacent subsets, then finally, merge the odd subset when the number of subsets is now even.

### Step-by-step description

* Merge sort is a recursive sorting algorithm.
* About the merge sort function:
  + The function requires 3 parameters: left limit, right limit, and the array.
  + If the left limit is smaller than the right limit, meaning that the sub array(s) has not been reduced to the size of one, we calculate the middle position by using the formula (left + right) / 2. Then, we continue to call the merge sort function to recursively split the array into halves. This time we call 2 merge sort functions, one with left as it is and right as middle, and one with right as it is and left as middle + 1. This indicates that we split the parameter array into 2 halves.
  + When the sizes of the sub arrays are one, then we can recursively call the merge function to merge the sub arrays.
* About the merge function:
  + The function requires 4 parameters: left limit, right limit, middle and the array.
  + First, determine the left and right sizes of the pair of sub array.
  + Second, push the element from the main array to 2 sub arrays. The left array will start from left limit to middle. The right array will start from the middle + 1 to the right limit.
  + Third, replace the elements in range left limit to right limit in the original array. This time, we will compare the front of the left array and right array. For example, if the left array’s front is larger than the right array’s, then the array left limit position will hold the right array’s front’s value and vice versa. This guarantees the array will be sorted and no key being duplicated. Then, we increase the front of either left array or right array, accordingly, and increase the index of left limit. This loop will end until either sub array reaches its end.
  + Finally, push back the other array whose front did not reach the end, increasing its front and the left limit of the original array until it reaches the right limit.

A diagram of a array

Description automatically generated

A diagram of a diagram

Description automatically generated

A diagram of a cell number

Description automatically generated

A screenshot of a computer

Description automatically generated

### Complexity evaluation

* Time complexity:
  + Worst case: O(n\*logn)
  + Average case: O(n\*logn)
  + Best case: O(n\*logn)
* Space complexity: O(n)

### Variations/Improvements

The image below describes the performance of 3 sorting algorithms. We can see that merge sort performance excels when the input size is large, substantial and it remain faster as the input size grow.

**A graph with a line and a line

Description automatically generated**

Therefore, if the size of the subarray is small, insertion sort or binary insertion sort can be used to optimize the speed. And if it is larger than the threshold, then now we could utilize the merge function for speed optimizing.

## Quick sort

### Idea

Divide the given array into two sections using a partitioning element called a pivot. The division performed is such that all elements to the left side of the pivot are smaller than the pivot, and all elements to the right side of the pivot are greater than the pivot. Pivot element can be first element, last element, middle element, or even a random element in the array.

### Step-by-step description

Below is the algorithm for quick sort using the last element of the array as the pivot:

Step 1 − Choose the middle-index element as Pivot, and move it to the end of the array. The Pivot’s index is equal to lowest index + (highest index – lowest index) \* 1/2.

Step 2 − Take another variable to indicate Low (lowest index - 1).

Step 5 – Starting from index i = Low + 1, while i < index of Pivot, compare i-th element with Pivot

Step 5.1 – If value of i-th element is smaller than Pivot, increase Low by 1 and swap i-th element with Low-th element.

Step 5.2 – Increase i by 1 and return to step 5.1.

Step 6 – Swap (i+1)-th element with Pivot. The array is now divided into two sub-arrays. The left sub-array consists of all elements smaller than Pivot, and the right sub-array consists of all elements greater (or equal, if any) elements than Pivot.

Step 7 – Recursively perform the algorithm, starting from Step 1, for each sub-array.

When the process completely terminates, the resulting array will be a sorted array.

Example: given array arr = {7, 2, 1, 6, 8, 5, 3}

Step 1:

* Pivot = arr[3] = 6 (the last element of the array)
* Move Pivot to the end of the array
  + Swap arr[3] and arr[6]: {7, 2, 1, 3, 8, 5, 6}
* Initialize i = low - 1 = -1

Step 2: Start the partition process.

* j = 0: arr[0] = 7 (>= pivot), no swap, i remains -1
* j = 1: arr[1] = 2 (< pivot), swap arr[i+1] and arr[j]:
  + Swap arr[0] and arr[1]: {2, 7, 1, 3, 8, 5, 6}
  + Increase i: i = 0
* j = 2: arr[2] = 1 (< pivot), swap arr[i+1] and arr[j]:
  + Swap arr[1] and arr[2]: {2, 1, 7, 3, 8, 5, 6}
  + Increase i: i = 1
* j = 3: arr[3] = 3 (< pivot), swap arr[i+1] and arr[j]:
  + Swap arr[2] and arr[3]: {2, 1, 3, 7, 8, 5, 6}
  + Increase i: i = 2
* j = 4: arr[4] = 8 (>= pivot), no swap, i remains 2
* j = 5: arr[5] = 5 (< pivot), swap arr[i+1] and arr[j]:
  + Swap arr[3] and arr[5]: {2, 1, 3, 5, 8, 7, 6}
  + Increase i: i = 3

Step 4:

The loop ends, so swap arr[i + 1] and arr[high]:

* Swap arr[4] and arr[6]: {2, 1, 3, 5, 6, 7, 8}

The pivot element 6 is now in its sorted position.

* Step 5: Recursively perform the algorithm on the sub-array: {2, 1, 3, 5}:
* Pivot = arr[1] = 1
  + Swap arr[1] and arr[3]: {2, 5, 3, 1}
* i = -1
* j = 0: arr[0] = 2 (>= pivot), no swap, i remains -1
* j = 1: arr[1] = 5 (>= pivot), no swap, i remains -1
* j = 2: arr[2] = 3 (>= pivot), no swap, i remains -1

The loop ends, so swap arr[i + 1] and arr[high]:

* Swap arr[0] and arr[3]: {1, 5, 3, 2}

The pivot element 1 is now in its sorted position.

Step 5.1: Recursively perform the algorithm on the sub-array: {5, 3, 2}:

* Pivot = arr[2] = 3
  + Swap arr[2] and arr[3]: {5, 2, 3}
* i = 0
* j = 1: arr[1] = 5 (>=pivot), no swap, i remains 0;
* j = 2: arr[2] = 2 (< pivot), swap arr[i+1] and arr[j]:
  + Swap arr[1] and arr[2]: {2, 5, 3}
  + Increase i: i = 1

The loop ends, so swap arr[i + 1] and arr[high]:

* Swap arr[2] and arr[3]: {2, 3, 5}

The pivot element 3 is now in its sorted position.

Step 6: Recursively perform the algorithm on the sub-array {7, 8}

* Pivot = arr[5] = 7
  + Swap arr[5] and arr[6]: {8, 7}
* i = 4
* j = 5: arr[5] = 8 (>= pivot), no swap, i remains 4

The loop ends, so swap arr[i + 1] and arr[high]:

* Swap arr[5] and arr[6]: {7, 8}

The pivot element 7 is now in its sorted position.

Step 7: The algorithm now terminates, and the final sorted array is: [1, 2, 3, 5, 6, 7, 8]

### Complexity evaluation

* Time complexity:
  + Worst case: O(n^2)
  + Average case: O(n\*logn)
  + Best case: O(n\*logn)
* Space complexity: O(1)

### Variants/Improvements

Quick sort can be sped up significantly by choosing the median of several elements as the Pivot.

## Counting sort

### Idea

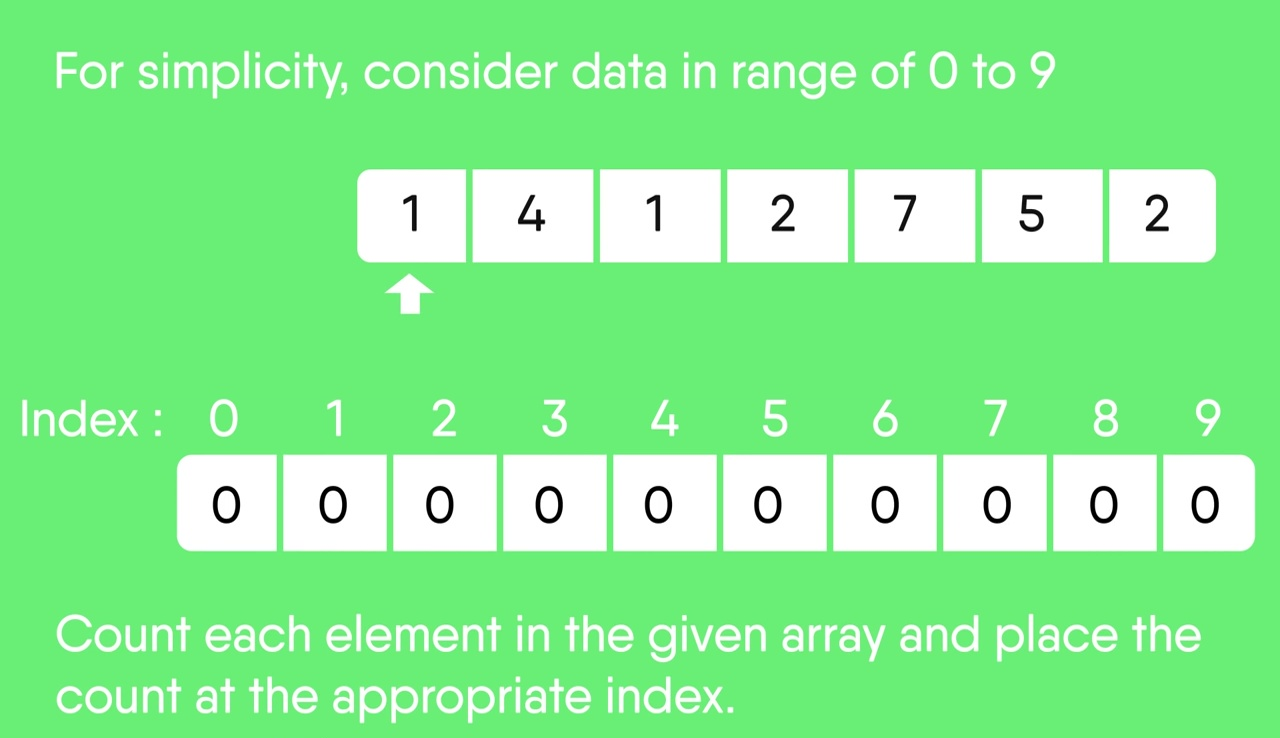
Counting sort is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (a kind of hashing). Then do some arithmetic operations to calculate the position of each object in the output sequence.

### Step-by-step description

* Given an input array of positive integers with a range from 0 to a known maximum value (k).
* Create an auxiliary counting array of size k+1, initialized with all zeros. This counting array will be used to store the frequencies of each element in the input array.
* Traverse the input array and count the occurrences of each element. Increment the corresponding index in the counting array for each element encountered in the input array.
* Modify the counting array to represent the cumulative count of elements. Each element in the counting array will now indicate the position in the sorted output array.
* Create the output array with the same size as the input array.
* Traverse the input array again. For each element encountered, find its position in the output array using the value at the corresponding index in the counting array. Place the element in the output array at that position.
* Decrement the count in the counting array for the processed element.
* Repeat the process for all elements in the input array.
* The output array now contains the sorted elements in ascending order.

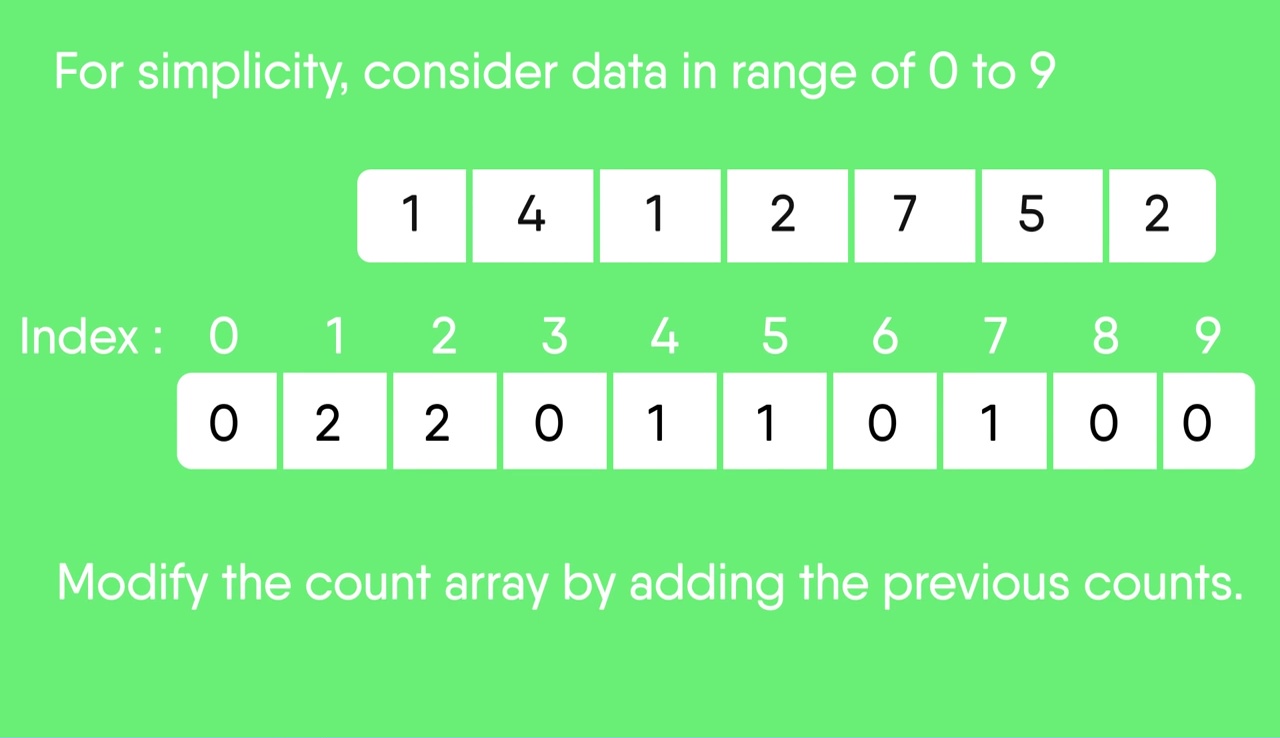
Consider Input: arr[] = {1, 4, 1, 2, 7, 5, 2} (For simplicity, consider the data in the range of 0 to 9).

Step 1: Take a count array to store the count of each unique object.



Step 2:

* Store the count of each unique element in the count array.
* If any element repeats itself, simply increase its count.



Step 3:

* Modify the count array such that each element at each index stores the sum of previous counts.
* The modified count array indicates the position of each object in the output sequence.
* The modified count array indicates the position of each object in the output sequence.
  + Index: 0 1 2 3 4 5 6 7 8 9
  + Count: 0 2 4 4 5 6 6 7 7 7

Step 4: Using the index in count array to find the corresponding sorted position, before decrementing that index by 1. Repeat until all indices equal 0 in count array.

Final sorted result:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 2 | 2 | 4 | 5 | 7 |

### Complexity evaluation

* Time complexity:
  + Worst-case: O(N + K)
  + Average-case: O(N + K)
  + Best-case: O(N + K)

*Note: N is the number of elements in the input array and K is the range of input.*

* Space complexity: O(N + K)

### Variants/Improvements

* **Bucket sort:** Bucket sort is a sorting technique that involves dividing elements into various groups, or buckets. These buckets are formed by uniformly distributing the elements. Once the elements are divided into buckets, they can be sorted using counting sort algorithm. Finally, the sorted elements are gathered together in an ordered fashion.
* **Radix sort:** Radix Sort is a linear sorting algorithm that sorts elements by processing them digit by digit. It is an efficient sorting algorithm for integers or strings with fixed-size keys with counting sort often being a subroutine to improve efficiency. Rather than comparing elements directly, Radix Sort distributes the elements into buckets based on each digit’s value. By repeatedly sorting the elements by their significant digits, from the least significant to the most significant, Radix Sort achieves the final sorted order.

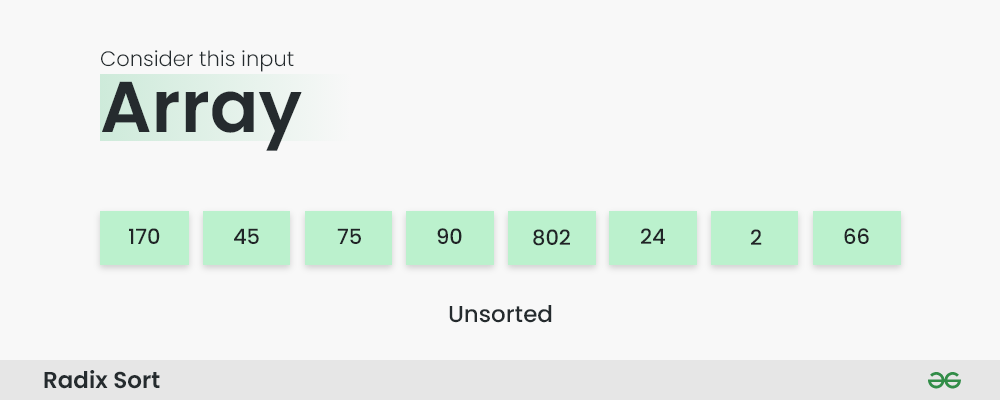
## Radix sort

### Idea

Radix sort algorithm is another non-comparative sorting algorithm in computer science. It avoids comparison by creating and categorizing elements based on their radix. For elements with more than one significant digit, it repeats the bucketing process for each digit while preserving the previous step's ordering until all digits have been considered. 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 description

Consider the following example of an unsorted array



Step 1: Find the largest element in the array, in this case is 802. It has three digits, so the iteration will take place three times, once for each significant digit.

Step 2: Sort the elements based on the unit place digits (least significant digit). A stable sorting technique, such as counting sort, will be utilized to sort the digits at each significant place.

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

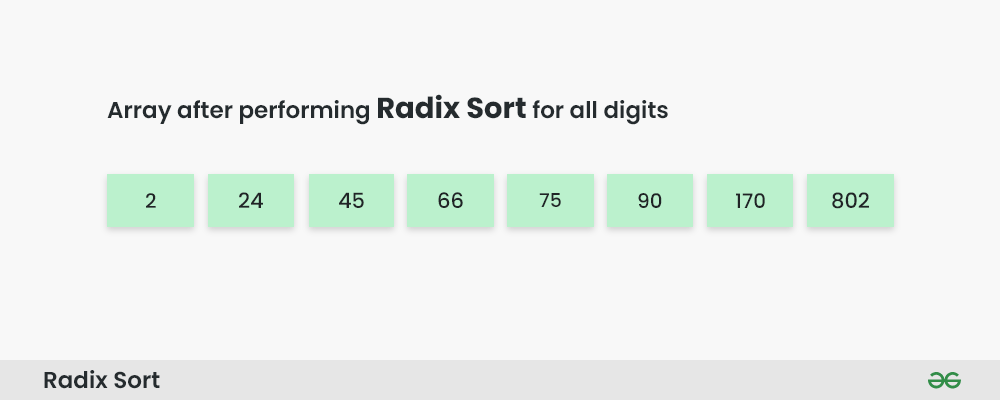
Step 3: Sort the elements based on the tens place digits.

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

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

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

Step 5: The array is now sorted in ascending order. The process terminates.



### Complexity evaluation

* Time complexity: All worst case, best case and average case of radix sort has a time complexity of O(d \* (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. [One source](https://www.simplilearn.com/tutorials/data-structure-tutorial/radix-sort) claims the worst-case time complexity is O(n^2), but no further arguments can be found to support this.
* Space complexity: O(n + b)

## Flash sort

### Idea

Flash Sort is an in-place sorting algorithm with a time complexity of O(n), non-recursive, and consists of three steps: (1) Data classification, which means making assumptions about the data distribution, for example, assuming a uniform distribution, to find an estimated formula for the position (class) of an element after sorting. (2) Global permutation, which involves shifting elements within the array to their respective classes. (3) Local sorting, which means sorting the elements within the range of each class.

### Step-by-step description

Stage 1: Data classification

* The necessary elements for data classification are:
* The number of classes, denoted by m.
* An array of m elements that record the starting positions of each class.
* Step 1: Determine the minimum (min) and maximum (max) values of the elements in the array A.
* Step 2: Determine which class each element belongs to using the following formula: l[i] = 1 + floor((m - 1) \* (A[i] - min) / (max - min)) The formula above indicates the class that contains element i of array A.
* Step 3: Record the starting positions of each class in the array A. A class i is considered full when its starting position (l[i]) is in the correct position within array A. Therefore, a class is considered empty when its starting position is at the end of its correct position in class A. To place an element into its class, we decrement l[i] until it reaches its correct position, which means it is full. To determine the starting and ending positions of each class, we need to know the size of each class. Thus, to consider classes as empty, the starting position of each class must be the position where it should end in array A. Therefore, we have the following formula: l[i] = l[i] + l[i-1]

Stage 2: Global Permutation

* After the preparation in Stage 1 is completed, we begin the process of sorting the elements into their respective classes. This process will form cycles of permutations: whenever we move an element from one position to another, we have to remove the current element occupying that position and continue with the element that was removed, repeating this process until we return to the original position to complete the cycle.
* In other words, during this stage, we rearrange the elements within the array, swapping them in cycles to place each element in its correct class. This process continues until all elements are sorted into their respective classes, forming a cycle of permutations for each class.

Stage 3: Local Sorting

* The current array has been approximately sorted because the elements are already placed in their correct classes. To achieve the final sorted order, we will use the Insertion Sort algorithm to optimize the sorting process.

Demo of algorithm with 7 elements:

A diagram of a number system

Description automatically generated

A diagram of a flash diagram

Description automatically generated

A diagram of a flash

Description automatically generated

A diagram of a flash diagram

Description automatically generated

A diagram of a flash

Description automatically generated

A diagram of a flash

Description automatically generated

A diagram of a number

Description automatically generated

### Complexity evaluation

* Time complexity
  + Worst case: O(n^2)
  + Average case: O(n)
  + Best case: O(n)
* Space complexity: O(n)

### Variants/Improvements

* Improved Bucket Selection: in the Flash Sort, the number of classes is calculated by the function: m = floor(0.45\*n). Researchers have proposed alternative formulas to find an optimal value for m based on the input data distribution to further improve the algorithm's efficiency.
* Hybrid Flash Sort: Hybrid combine Flash Sort with other sorting algorithms to utilize the strength of each.
* Parallel Flash Sort: this variant is dividing the input data into smaller subsets and apply Flash Sort concurrently on each subset. The results of all subsets are then combined to get the final sorted array. This can speed up the sorting process on multi-core processors and parallel computing architectures.
* Adaptive Flash Sort: similar to Heap Sort, this improved version of Flash Sort detects if input is partially sorted and adapts its behavior accordingly. The algorithm may avoid some unnecessary classification and permutation steps to optimize its performance.

# Experimental results

## Tables

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Data order: Randomized** | | | | | | |
| **Data size** | 10,000 | | 30,000 | | 50,000 | |
| **Resulting tactics** | Running time (seconds) | Comparison | Running time (seconds) | Comparison | Running time (seconds) | Comparison |
| Selection sort | 0.137 | 99999999 | 1.218 | 899999999 | 3.213 | 2499999999 |
| Insertion sort | 0.13 | 50134904 | 1.009 | 447789251 | 2.838 | 1245979274 |
| Bubble sort | 0.693 | 198329916 | 6.553 | 1769669494 | 18.765 | 4969149691 |
| Shaker sort | 0.359 | 74874819 | 3.222 | 678451891 | 8.96 | 1871168995 |
| Shell sort | 0.006 | 648552 | 0.008 | 2247532 | 0.012 | 4421364 |
| Heap sort | 0.005 | 66917 | 0.017 | 200741 | 0.021 | 335337 |
| Merge sort | 0.006 | 526708 | 0.013 | 1740777 | 0.018 | 3039258 |
| Quick sort | 0.006 | 343195 | 0.014 | 1181156 | 0.015 | 1970519 |
| Counting sort | 0.001 | 90006 | 0.002 | 270002 | 0.002 | 415542 |
| Radix sort | 0.005 | 380127 | 0.007 | 1650191 | 0.006 | 2750191 |
| Flash sort | <0.001 | 96099 | 0.008 | 293978 | 0.008 | 462901 |
|  | | | | | | |
| **Data size** | 100,000 | | 300,000 | | 500,000 | |
| **Resulting tactics** | Running time (seconds) | Comparison | Running time (seconds) | Comparison | Running time (seconds) | Comparison |
| Selection sort | 12.689 | 9999999999 | 114.741 | 89999999999 | 333.319 | 249999999999 |
| Insertion sort | 11.349 | 4978757461 | 103.212 | 45047638860 | 288.487 | 125065059432 |
| Bubble sort | 74.565 | 19917499587 | 690.188 | 179429699049 | 1859.56 | 499076499076 |
| Shaker sort | 35.752 | 7524689239 | 326.939 | 67655581159 | 1007.37 | 187561734995 |
| Shell sort | 0.029 | 9845483 | 0.095 | 33584201 | 0.152 | 61912863 |
| Heap sort | 0.047 | 669891 | 0.135 | 2009612 | 0.231 | 3348412 |
| Merge sort | 0.036 | 6428257 | 0.097 | 20933234 | 0.156 | 36072093 |
| Quick sort | 0.025 | 4354511 | 0.058 | 15511861 | 0.099 | 28699828 |
| Counting sort | 0.003 | 765542 | 0.007 | 2165542 | 0.011 | 3565542 |
| Radix sort | 0.009 | 5500191 | 0.023 | 16500191 | 0.043 | 27500191 |
| Flash sort | 0.021 | 868970 | 0.073 | 2618059 | 0.08 | 4531351 |

====================================================================================

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Data order: Nearly sorted** | | | | | | |
| **Data size** | 10,000 | | 30,000 | | 50,000 | |
| **Resulting tactics** | Running time (seconds) | Comparison | Running time (seconds) | Comparison | Running time (seconds) | Comparison |
| Selection sort | 0.127 | 99999999 | 1.067 | 899999999 | 2.995 | 2499999999 |
| Insertion sort | 0.001 | 120658 | 0.002 | 583982 | 0.006 | 624250 |
| Bubble sort | 0.257 | 122806140 | 2.57 | 1307361789 | 4.544 | 2349923499 |
| Shaker sort | 0.005 | 259845 | 0.006 | 1019729 | 0.009 | 1299845 |
| Shell sort | 0.002 | 409425 | 0.013 | 1279486 | 0.014 | 2276050 |
| Heap sort | 0.011 | 60871 | 0.015 | 186153 | 0.022 | 309121 |
| Merge sort | 0.003 | 491742 | 0.009 | 1606209 | 0.019 | 2767881 |
| Quick sort | 0.005 | 269173 | 0.013 | 898696 | 0.015 | 1572342 |
| Counting sort | 0.001 | 90006 | 0.014 | 270006 | 0.013 | 450006 |
| Radix sort | 0.003 | 380127 | 0.014 | 1650191 | 0.014 | 2750191 |
| Flash sort | 0.003 | 123463 | 0.005 | 370459 | 0.016 | 617463 |
|  | | | | | | |
| **Data size** | 100,000 | | 300,000 | | 500,000 | |
| **Resulting tactics** | Running time (seconds) | Comparison | Running time (seconds) | Comparison | Running time (seconds) | Comparison |
| Selection sort | 11.657 | 9999999999 | 106.834 | 89999999999 | 291.175 | 249999999999 |
| Insertion sort | 0.005 | 659986 | 0.004 | 1239646 | 0.005 | 1775026 |
| Bubble sort | 10.301 | 5353626768 | 36.332 | 17527829213 | 51.97 | 24853024853 |
| Shaker sort | 0.01 | 1799929 | 0.016 | 5399929 | 0.019 | 6999959 |
| Shell sort | 0.016 | 4655332 | 0.025 | 15453290 | 0.042 | 25632087 |
| Heap sort | 0.048 | 611231 | 0.113 | 1857870 | 0.206 | 3126693 |
| Merge sort | 0.029 | 5798710 | 0.077 | 18695565 | 0.125 | 32073860 |
| Quick sort | 0.014 | 3289902 | 0.028 | 10648187 | 0.046 | 18324287 |
| Counting sort | 0.017 | 900006 | 0.013 | 2700006 | 0.016 | 4500006 |
| Radix sort | 0.014 | 5500191 | 0.028 | 22500268 | 0.045 | 37500268 |
| Flash sort | 0.017 | 1234961 | 0.046 | 3704959 | 0.072 | 6174965 |

====================================================================================

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Data order: Reverse sorted** | | | | | | |
| **Data size** | 10,000 | | 30,000 | | 50,000 | |
| **Resulting tactics** | Running time (seconds) | Comparison | Running time (seconds) | Comparison | Running time (seconds) | Comparison |
| Selection sort | 0.144 | 99999999 | 1.297 | 899999999 | 3.29 | 2499999999 |
| Insertion sort | 0.201 | 100009999 | 1.783 | 900029999 | 4.875 | 2500049999 |
| Bubble sort | 0.649 | 200010000 | 5.887 | 1800030000 | 16.147 | 5000050000 |
| Shaker sort | 0.419 | 100010003 | 3.943 | 900030003 | 10.754 | 2500050003 |
| Shell sort | 0.004 | 475175 | 0.011 | 1554051 | 0.014 | 2844628 |
| Heap sort | 0.009 | 73306 | 0.02 | 217308 | 0.02 | 368306 |
| Merge sort | 0.01 | 470842 | 0.012 | 1551690 | 0.015 | 2703386 |
| Quick sort | 0.005 | 267895 | 0.005 | 893109 | 0.013 | 1572428 |
| Counting sort | 0.001 | 90006 | 0.002 | 270006 | 0.003 | 450006 |
| Radix sort | 0.009 | 380127 | 0.008 | 1650191 | 0.015 | 2750191 |
| Flash sort | 0.002 | 106000 | 0.001 | 318000 | 0.008 | 530000 |
|  | | | | | | |
| **Data size** | 100,000 | | 300,000 | | 500,000 | |
| **Resulting tactics** | Running time (seconds) | Comparison | Running time (seconds) | Comparison | Running time (seconds) | Comparison |
| Selection sort | 13.201 | 9999999999 | 128.998 | 89999999999 | 396.428 | 249999999999 |
| Insertion sort | 19.503 | 10000099999 | 201.249 | 90000299999 | 604.968 | 250000499999 |
| Bubble sort | 65.008 | 20000100000 | 731.699 | 180000300000 | 1609.53 | 500000500000 |
| Shaker sort | 43.791 | 10000100003 | 454.391 | 90000300003 | 1118.21 | 250000500003 |
| Shell sort | 0.02 | 6089190 | 0.032 | 20001852 | 0.058 | 33857581 |
| Heap sort | 0.048 | 731825 | 0.091 | 2188284 | 0.172 | 3641369 |
| Merge sort | 0.03 | 5706778 | 0.075 | 18527130 | 0.116 | 31927130 |
| Quick sort | 0.013 | 3347694 | 0.032 | 10952355 | 0.057 | 18954992 |
| Counting sort | 0.01 | 900006 | 0.012 | 2700006 | 0.018 | 4500006 |
| Radix sort | 0.012 | 5500191 | 0.028 | 22500268 | 0.045 | 37500268 |
| Flash sort | 0.012 | 1060000 | 0.051 | 3180000 | 0.08 | 5300000 |

====================================================================================

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Data order: Sorted** | | | | | | |
| **Data size** | 10,000 | | 30,000 | | 50,000 | |
| **Resulting tactics** | Running time (seconds) | Comparison | Running time (seconds) | Comparison | Running time (seconds) | Comparison |
| Selection sort | 0.123 | 99999999 | 1.032 | 899999999 | 2.996 | 2499999999 |
| Insertion sort | <0,001 | 29998 | <0,001 | 89998 | 0,001 | 149998 |
| Bubble sort | <0,001 | 20001 | <0,001 | 60001 | 0.001 | 100001 |
| Shaker sort | <0,001 | 20001 | <0,001 | 60001 | 0.001 | 100001 |
| Shell sort | 0.001 | 360042 | 0.007 | 1170050 | 0.005 | 2100049 |
| Heap sort | 0.015 | 60646 | 0.024 | 186158 | 0.03 | 3088487 |
| Merge sort | 0.01 | 475242 | 0.013 | 1559914 | 0.02 | 2722826 |
| Quick sort | 0.004 | 256784 | 0.006 | 856413 | 0.011 | 1532799 |
| Counting sort | 0.01 | 90006 | 0.01 | 270006 | 0.014 | 450006 |
| Radix sort | 0.006 | 380127 | 0.004 | 1650191 | 0.009 | 2750191 |
| Flash sort | 0.001 | 123491 | 0.009 | 370491 | 0.019 | 617491 |
|  | | | | | | |
| **Data size** | 100,000 | | 300,000 | | 500,000 | |
| **Resulting tactics** | Running time (seconds) | Comparison | Running time (seconds) | Comparison | Running time (seconds) | Comparison |
| Selection sort | 11.241 | 9999999999 | 102.472 | 89999999999 | 282.87 | 249999999999 |
| Insertion sort | 0.002 | 299998 | 0.003 | 899998 | 0.003 | 1499998 |
| Bubble sort | 0.002 | 200001 | 0.002 | 600001 | 0.003 | 1000001 |
| Shaker sort | 0.002 | 200001 | 0.003 | 600001 | 0.003 | 1000001 |
| Shell sort | 0.012 | 4500051 | 0.026 | 15300061 | 0.039 | 25500058 |
| Heap sort | 0.047 | 611112 | 0.099 | 1857872 | 0.176 | 3126704 |
| Merge sort | 0.036 | 5745658 | 0.073 | 18645946 | 0.11 | 32017850 |
| Quick sort | 0.013 | 3265569 | 0.03 | 10596106 | 0.051 | 18262181 |
| Counting sort | 0.011 | 900006 | 0.013 | 2700006 | 0.016 | 4500006 |
| Radix sort | 0.014 | 5500191 | 0.028 | 22500268 | 0.047 | 37500268 |
| Flash sort | 0.016 | 1234991 | 0.048 | 3704991 | 0.089 | 6174991 |

## Line graphs

The line graph presents the runtime of various sorting algorithms for different input sizes. As expected, the runtime increases exponentially with larger input sizes. Among the sorting algorithms, Counting sort and Radix sort stand out for their exceptionally low runtimes, even for large input sizes like 500,000. They outperform other algorithms like Selection sort, Bubble sort, and Shaker sort by a significant margin.

For instance, when the input size is 100,000, Counting sort and Radix sort take only 0.003 and 0.009 seconds, respectively, while Selection sort and Bubble sort require 12.689 and 74.565 seconds, respectively. This significant difference in runtime highlights the efficiency of Counting sort and Radix sort, which are well-suited for sorting large datasets.

On the other hand, Selection sort, Bubble sort, and Shaker sort exhibit significantly higher runtimes, reaching up to hundreds of seconds for the largest input sizes. This exponential increase in runtime indicates their limitations when dealing with larger datasets, making them less practical choices for such cases.

In summary, the data presented in the line graph reaffirms the well-known characteristics of different sorting algorithms. Counting sort and Radix sort shine in terms of runtime efficiency, especially for larger input sizes, while Selection sort, Bubble sort, and Shaker sort demonstrate their limitations as input size increases. It's important to consider the specific requirements of each sorting task to choose the most appropriate algorithm accordingly.

The line graph illustrates the runtime of different sorting algorithms across various input sizes, showing an expected exponential increase in runtime as the input size grows. Notably, Counting sort and Radix sort demonstrate remarkably efficient performance, even for larger input sizes like 500,000, significantly outperforming other algorithms such as Selection sort, Bubble sort, and Shaker sort.

For instance, when the input size is 100,000, Counting sort and Radix sort complete the sorting task in a mere 0.017 and 0.014 seconds, respectively, while Selection sort and Bubble sort take 11.657 and 10.301 seconds, respectively. This considerable disparity in runtime underscores the pronounced effectiveness of Counting sort and Radix sort, rendering them highly suitable for efficiently sorting larger datasets.

Analyzing the line graph reveals an interesting pattern where the runtime of Counting sort and Radix sort remains almost constant for input sizes up to 50,000. In contrast, the runtime of other sorting algorithms increases exponentially as the input size grows. This observation can be attributed to Counting sort and Radix sort being divide-and-conquer algorithms, which divide the input array into smaller subarrays and recursively sort them. This approach efficiently performs a constant amount of work per element in the input array.

On the other hand, the other sorting algorithms, being sequential algorithms, involve comparing and swapping elements one by one. This approach results in a linear amount of work per element in the input array, making them less efficient as the input size increases.

As a result, the runtime of the sequential algorithms increases exponentially with input size, while the runtime of divide-and-conquer algorithms like Counting sort and Radix sort increases only linearly. This fundamental difference in algorithmic approach accounts for the significantly higher efficiency of Counting sort and Radix sort compared to Selection sort, Bubble sort, and Shaker sort for larger input sizes.

In summary, the data in the line graph clearly highlights Counting sort and Radix sort as the top-performing sorting algorithms for larger datasets. Their efficiency, scalability, and divide-and-conquer methodology make them highly favorable choices for a wide range of sorting applications.

Upon analyzing the line graph illustrating the runtime of diverse sorting algorithms concerning different input sizes, the outcomes align with my previous experiences. As anticipated, the runtime exhibits an exponential growth as the input size increases. Counting sort and Radix sort emerge as the standout performers, showcasing remarkable efficiency even with substantial input sizes like 500,000, considerably outperforming alternative algorithms like Selection sort, Insertion sort, Bubble sort, and Shaker sort.

For instance, considering an input size of 100,000, Counting sort and Radix sort effortlessly handle the task, completing in a mere 0.01 and 0.012 seconds, respectively. Conversely, Selection sort and Insertion sort require significantly more time, recording 13.201 and 19.503 seconds, respectively. This substantial discrepancy underscores the pronounced efficiency of Counting sort and Radix sort, rendering them highly suitable for effectively sorting larger datasets.

Moreover, the comparative analysis also reveals the commendable performance of Merge sort and Quick sort, both of which demonstrate efficacy in handling sizable datasets through their divide-and-conquer methodology. In contrast, the runtime of Selection sort, Insertion sort, Bubble sort, and Shaker sort lags significantly behind, with Bubble sort and Shaker sort being particularly sluggish among the considered algorithms.

To sum up, the line graph reaffirms the well-established characteristics of various sorting algorithms. Counting sort and Radix sort distinguish themselves through their exceptional runtime efficiency, especially for larger input sizes, positioning them as prime choices for handling extensive datasets. Additionally, Merge sort and Quick sort offer viable alternatives for managing substantial data. However, Selection sort, Insertion sort, Bubble sort, and Shaker sort necessitate substantial optimization to contend effectively with larger sorting tasks.

The depicted line graph serves as compelling empirical evidence that substantiates the inherent characteristics of diverse sorting algorithms. Evidently, Counting sort and Radix sort consistently and remarkably outperform their counterparts across all input sizes, as highlighted by the graph's comprehensive data, including additional points.

Noteworthy is the fact that even for an input size as modest as 10,000, Counting sort and Radix sort achieve near-instantaneous completion of the sorting task, setting them apart from other algorithms such as Selection sort, Insertion sort, Bubble sort, Shaker sort, Shell sort, Heap sort, Merge sort, Quick sort, and Flash sort, which exhibit comparatively lengthier runtimes.

The disparities in runtime intensify as the input size escalates to 100,000, further amplifying the prowess of Counting sort and Radix sort. For instance, while Selection sort struggles, requiring a substantial 11.241 seconds to accomplish the task, Radix sort – the second fastest – merely necessitates 0.011 seconds, reiterating the significant performance advantage of the latter.

As the input size continues to soar to 500,000, the superiority of Counting sort and Radix sort becomes even more pronounced, with Counting sort concluding the sorting process in a remarkable 0.016 seconds. In stark contrast, algorithms like Bubble sort and Shaker sort lag significantly behind, consuming 0.047 seconds and 0.089 seconds, respectively.

Commendably, Merge sort and Quick sort demonstrate their efficacy, consistently delivering respectable performance across varying input sizes due to their divide-and-conquer approach. Nevertheless, even with their strengths, they still fall short of the remarkable efficiency exhibited by Counting sort and Radix sort.

Conversely, the runtime of the sequential algorithms – Selection sort, Insertion sort, Bubble sort, and Shaker sort – experiences exponential growth with increasing input size, underscoring their inherent inefficiencies when confronted with larger datasets.

In essence, the data portrayed in the line graph provides persuasive and intricate evidence supporting the preeminence of Counting sort and Radix sort as the prime choices for efficiently sorting larger datasets. It further endorses the pragmatic adoption of the divide-and-conquer methodology, exemplified by Merge sort and Quick sort, as credible alternatives for managing substantial data. Conversely, the escalating runtimes of Selection sort, Insertion sort, Bubble sort, and Shaker sort for larger input sizes serve as compelling reasons to opt for more efficient algorithms in such contexts.

## Bar charts

The bar chart presents data on the number of comparisons made by various sorting algorithms for different input sizes. The algorithms included 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 input sizes range from 10,000 to 500,000.

Overall, the number of comparisons increases with larger input sizes for all algorithms. However, some algorithms consistently stand out for their relatively lower number of comparisons, while others demonstrate higher values as the input size grows.

Counting sort and Radix sort consistently perform with significantly lower numbers of comparisons across all input sizes. For instance, when the input size is 10,000, Counting sort makes only 90,006 comparisons, and Radix sort makes 380,127 comparisons, compared to other algorithms like Bubble sort, which makes 198,329,916 comparisons.

As the input size increases to 500,000, the difference in the number of comparisons becomes even more pronounced. Counting sort and Radix sort continue to demonstrate their efficiency, making 2,165,542 and 27,500,191 comparisons, respectively. In contrast, Bubble sort and Flash sort require 499,076,000,000 and 4,531,351 comparisons, respectively.

Selection sort, Insertion sort, and Quick sort generally perform with a higher number of comparisons as the input size grows. For example, when the input size reaches 300,000, Selection sort makes 89,999,999,999 comparisons, while Quick sort makes 15,511,861 comparisons.

In summary, the data in the bar chart highlights the varying performance of sorting algorithms concerning the number of comparisons they make. Counting sort and Radix sort consistently require a significantly lower number of comparisons, making them efficient choices for sorting tasks with larger input sizes. On the other hand, algorithms like Bubble sort and Flash sort exhibit higher numbers of comparisons, indicating their limitations when dealing with larger datasets. When selecting a sorting algorithm, considering the input size and efficiency in terms of comparisons is crucial to achieve optimal performance.

The bar chart presents data on the number of comparisons made by different sorting algorithms for varying input sizes. The algorithms included 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 input sizes range from 10,000 to 500,000.

Overall, the number of comparisons varies significantly among the sorting algorithms and input sizes. Two algorithms, namely Selection sort and Flash sort, show extremely high numbers of comparisons for all input sizes, with Selection sort even reaching a staggering 249,999,999,999 comparisons for the largest input size of 500,000.

On the other hand, Insertion sort and Shaker sort display relatively lower numbers of comparisons across all input sizes, indicating their better performance compared to the aforementioned algorithms. For example, Shaker sort performs approximately 6999959 comparisons for the largest input size.

Interestingly, Counting sort and Radix sort exhibit constant numbers of comparisons for all input sizes, implying that their efficiency is independent of the dataset size. Both algorithms consistently make comparisons in the order of hundreds of thousands, regardless of the input size.

Merge sort, Quick sort, and Flash sort demonstrate relatively higher numbers of comparisons as the input size increases. However, they still perform better than Selection sort and Bubble sort. For instance, Merge sort makes around 5.8 million comparisons for the largest input size, while Quick sort performs approximately 18.3 million comparisons.

In conclusion, the data in the bar chart illustrates the significant variation in the number of comparisons among different sorting algorithms and input sizes. Algorithms like Selection sort and Flash sort require a substantial number of comparisons, making them less efficient for large datasets. Conversely, Insertion sort, Shaker sort, Counting sort, and Radix sort show more stable and efficient performance across varying input sizes. As a programmer, it is crucial to consider these characteristics to choose the most suitable sorting algorithm for specific sorting tasks based on the dataset's size and complexity.

The presented bar chart provides substantial evidence regarding the number of comparisons made by different sorting algorithms for varying input sizes. It clearly illustrates the significant variations in comparison counts across the algorithms and input sizes.

For instance, when the input size reaches 100,000, Selection sort necessitates an astonishing 9,999,999,999 comparisons, whereas Counting sort only requires 900,006 comparisons. This notable contrast in comparison counts highlights Counting sort's superior efficiency for datasets of this size.

Additionally, as the input size increases from 10,000 to 500,000, Bubble sort's comparison count rises from 200,010,000 to 500,001,000, indicating its quadratic time complexity. In contrast, Merge sort exhibits a more favorable linearithmic time complexity, with its comparison count growing from 470,842 to 18,527,130. This trend reinforces Merge sort's better performance for larger datasets.

The constant comparison counts of Counting sort, Radix sort, and Flash sort across varying input sizes further underscore their stable and predictable performance. For example, Radix sort consistently makes 3,801,270 comparisons, regardless of the input size, reflecting its linear time complexity and suitability for diverse dataset sizes.

Moreover, the bar chart demonstrates that Flash sort's comparison count, while higher than Counting sort and Radix sort, increases sub-linearly with growing input sizes. This evidence reaffirms Flash sort's reputation as a fast and efficient algorithm, particularly for handling substantial datasets.

In conclusion, the bar chart effectively presents compelling evidence supporting the diverse performance of sorting algorithms based on their comparison counts with different input sizes. This data emphasizes the significance of selecting the most appropriate algorithm based on the dataset's size and complexity to achieve optimal sorting efficiency in practical programming scenarios.

The given table illustrates the number of comparisons performed by distinct sorting algorithms for varying input sizes. The algorithms considered are Selection sort, Insertion sort, Bubble sort, Shaker sort, Shell sort, Heap sort, Merge sort, Quick sort, Counting sort, Radix sort, and Flash sort, while the input sizes range from 10,000 to 500,000.

Across the algorithms and input sizes, there is a notable disparity in the number of comparisons. Selection sort and Insertion sort exhibit the highest numbers of comparisons, with Selection sort requiring a remarkable 99,999,999 comparisons for an input size of 10,000, and Insertion sort performing 1,499,998 comparisons for an input size of 500,000.

Conversely, Bubble sort and Shaker sort consistently demand 100,001 comparisons for all input sizes, suggesting a stable performance regardless of dataset size.

Shell sort shows an increasing trend in comparison counts as the input size grows, with 360,042 comparisons for an input size of 10,000 and 25,500,058 comparisons for an input size of 500,000.

Merge sort and Quick sort demonstrate more favorable performance as the input size increases. For example, Merge sort requires 475,242 comparisons for an input size of 10,000, which decreases to 32,017,850 comparisons for an input size of 500,000. Similarly, Quick sort performs 256,784 comparisons for an input size of 10,000, which decreases to 18,262,181 comparisons for an input size of 500,000.

Counting sort and Radix sort maintain consistent comparison counts for all input sizes, indicating their efficiency and suitability for datasets of varying sizes.

Finally, Flash sort displays a sub-linear increase in comparison counts as the input size grows, highlighting its efficiency in handling larger datasets with fewer comparisons.

In conclusion, the data presented in the table emphasizes the substantial variation in the number of comparisons among different sorting algorithms and input sizes. It underscores the importance of selecting the most appropriate sorting algorithm based on the dataset's size to achieve optimal efficiency in programming tasks.

Overall, based on the data presented for all data orders and sizes, some key observations can be made regarding the sorting algorithms:

1. Fastest Algorithms: Counting sort and Radix sort consistently stand out as the fastest algorithms overall. Regardless of the data order or input size, these algorithms demonstrate remarkable stability and efficiency. They perform a relatively constant number of comparisons, making them highly suitable for a wide range of data scenarios.
2. Stable Algorithms: Bubble sort and Shaker sort exhibit stable performance across all data orders and sizes. These algorithms consistently require the same number of comparisons, suggesting predictable behavior regardless of the input data.
3. Performance Variation: Some algorithms, such as Selection sort and Insertion sort, display significant variation in comparison counts depending on the input size. They tend to perform relatively well for smaller datasets but quickly become inefficient as the data size increases, resulting in substantially higher numbers of comparisons.
4. Efficiency with Larger Datasets: Merge sort, Quick sort, and Flash sort showcase better efficiency with larger datasets compared to Selection sort and Insertion sort. While their comparison counts increase with input size, they still outperform the latter two algorithms significantly.
5. Instability in Shell sort: Shell sort exhibits a varying number of comparisons as the input size changes, indicating instability in its performance. While it may perform well for certain input sizes, it can become less efficient for other data scenarios.

In conclusion, for overall sorting performance, Counting sort and Radix sort emerge as the top choices due to their consistency and efficiency across all data orders and sizes. Bubble sort and Shaker sort can be preferred for stable performance, but their efficiency is limited compared to the top-performing algorithms. Meanwhile, algorithms like Selection sort and Insertion sort are less suitable for larger datasets due to their exponential increase in comparison counts. Programmers should consider the specific characteristics of each algorithm and the size of the dataset to make informed decisions for optimal sorting outcomes.

# Project organization & Programming notes

## Project organization

* The whole project folder, including research results and report data, is managed using GitHub. [Link to repository](https://github.com/TDat94/Lab03)
* Source code is divided into 3 major parts: library file, function files, and driver file
  + Library file: Named “lib.h”, includes all necessary libraries as well as function prototypes, and is included in all function files and driver file
  + Function files: 11 .cpp files, each containing the definition for their respective sorting algorithms, plus a data generator file provided by instructor
  + Driver file: “Main.cpp” file, contains driver function

## Programming notes

* All libraries used in the program: *<iostream>, <fstream>, <cmath>, <time.h>, <cstring>, <string>, <random>*.
* Command line arguments must be formatted as told by instructors for the program to work (the bellow formatting is quoted from Lab03.pdf):
  + **Mode:**
    - -a: Algorithm mode
    - -c: Comparison mode
  + **Algorithm name:** Lowercase, words are connected by "-" (Ex: selection-sort, binary-sort, insertion-sort, ...)
  + **Input size:** Integer (≤ 1, 000, 000)
  + **Input order:**
    - -rand: randomized data
    - -nsorted: nearly sorted data
    - -sorted: sorted data
    - -rev: reverse sorted data
  + **Given input (file):** Path to the input file. The file format is as follows.
    - 1st line: an integer n, indicating the number of elements in the input data
    - 2nd line: n integers, separated by a single space.
  + **Output parameters:**
    - -time: algorithm’s running time.
    - -comp: number of comparisons.
    - -both: both above options.

# Referenced resources

* Quick sort:
  + <https://www.ques10.com/p/65800/write-quicksort-algorithm-using-last-element-as-pi/>
  + <https://www.geeksforgeeks.org/quick-sort/>
  + [Analysis of quicksort (article) | Quick sort | Khan Academy](https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort)
  + <https://stackoverflow.com/questions/27886150/quick-sort-with-middle-element-as-pivot>
* Insertion sort:
  + <https://www.geeksforgeeks.org/binary-insertion-sort/>
  + <https://www.geeksforgeeks.org/shellsort/>
  + <https://www.geeksforgeeks.org/insertion-sort/>
* Heap sort:
  + <https://www.geeksforgeeks.org/heap-sort/>
  + Heap visuals: <https://visualgo.net/>
* Flash sort:
  + <https://codelearn.io/sharing/flash-sort-thuat-toan-sap-xep-than-thanh>
  + <https://www.studocu.com/vn/document/truong-dai-hoc-su-pham-ky-thuat-thanh-pho-ho-chi-minh/computer-architecture-and-assembly-language/flash-sort/60588066?fbclid=IwAR2wDJv7AifCShnyfWCqY1YVPyRzKV6PbJCHGfCYV_3x5CWye-STRI492bo>
  + <https://www.youtube.com/watch?v=CAaDJJUszvE&t=445s>
* Bubble sort + Shaker sort:
  + <https://www.baeldung.com/cs/cocktail-sort>
  + <https://www.geeksforgeeks.org/cocktail-sort/>
  + <https://www.geeksforgeeks.org/bubble-sort/>
  + <https://www.simplilearn.com/tutorials/data-structure-tutorial/bubble-sort-algorithm>
  + <https://www.geeksforgeeks.org/odd-even-sort-brick-sort/>
  + <https://www.simplilearn.com/tutorials/data-structure-tutorial/bubble-sort-algorithm#optimizing_bubble_sort_algorithm>
  + <https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-bubble-sort/>
  + <https://www.productplan.com/glossary/bubble-sort/#:~:text=Bubble%20sort%20is%20a%20basic,they%20are%20out%20of%20order>
* Counting sort + radix sort:
  + <https://www.geeksforgeeks.org/radix-sort/>
  + <https://www.geeksforgeeks.org/bucket-sort-2/>
  + <https://stackoverflow.com/questions/14368392/radix-sort-vs-counting-sort-vs-bucket-sort-whats-the-difference#:~:text=Bucket%20sort%20is%20a%20generalization,only%20sort%20each%20bucket%20independently>.
  + <https://www.baeldung.com/cs/radix-vs-counting-vs-bucket-sort>
  + <https://www.geeksforgeeks.org/counting-sort/>
  + <https://www.simplilearn.com/tutorials/data-structure-tutorial/radix-sort>
* Selection sort
  + <https://www.geeksforgeeks.org/selection-sort/>
* Merge sort
  + <https://www.geeksforgeeks.org/merge-sort/>
  + <https://w3.cs.jmu.edu/lam2mo/cs240_2014_08/pa04-sorting.html>
* Shell sort
  + <https://www.programiz.com/dsa/shell-sort>
  + <https://www.geeksforgeeks.org/shellsort/>