

VIETNAM NATIONAL UNIVERSITY-HO CHI MINH CITY

HO CHI MINH UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

KNOWLEDGE ENGINEERING DEPARTMENT

Sorting Algorithm Report

Report topic: Research on Sorting Algorithm

Course: Data Structure and Algorithm

Student Group:

23127004-Le Nhat Khoi

23127011-Le Anh Duy

23127165-Nguyen Hai Dang

23127189-Tran Trong Hieu

Instructors:

Ms Tran Thi Thao Nhi

Mr Bui Huy Thong

July 4, 2024



INTRODUCTION

"Sorting is the most heavily studied concept in computer science. Algorithms that sort data efficiently and correctly are vital to the performance of other algorithms."

– Robert Sedgewick

Sorting algorithms are a fundamental concept in computer science, playing a crucial role in the efficiency and performance of various applications and systems.

The primary purpose of sorting is to arrange data in a particular order, often in numerical or lexicographical order, which enhances the efficiency of other algorithms and processes that depend on sorted data.

Sorting algorithms are widely studied and implemented because they provide the groundwork for problem-solving in computer science, enabling quicker data retrieval, easier data manipulation, and optimized performance in search algorithms, database management, and more.

This report dives into the ideas with step-by-step descriptions of different sorting algorithms as well as exploring theoretical underpinning and performance characteristics. We will examine various types of sorting algorithms, from simple ones like Bubble Sort and Insertion Sort to more complex and efficient ones like Merge Sort, Quick Sort, and Heap Sort. Each algorithm will be analyzed in terms of its time complexity, space complexity, stability, and suitability for different types of data and use cases.

Through this research, we aim to provide a comprehensive understanding of how sorting algorithms work, their advantages and disadvantages, and the scenarios in which each algorithm performs best.

Contents

1	Algorithm presentation	5
1.1	Selection Sort	5
1.1.1	Idea	5
1.1.2	Description and Pseudo Code	5
1.1.3	Complexity evaluation	6
1.1.4	Variants	6
1.2	Insertion Sort	7
1.2.1	Idea	7
1.2.2	Description and Pseudocode	7
1.2.3	Complexity evaluation	8
1.2.4	Variants and improved versions	9
1.3	Bubble Sort	10
1.3.1	Idea	10
1.3.2	Description and Pseudo Code	10
1.3.3	Complexity evaluation	11
1.3.4	Variant and Improved version	12
1.4	Shaker Sort	12
1.4.1	Idea	12
1.4.2	Description and Pseudo Code	12
1.4.3	Complexity evaluation	14
1.4.4	Variants and improved versions	15
1.5	Shell Sort	16
1.5.1	Idea	16
1.5.2	Description and Pseudo Code	16
1.5.3	Complexity evaluation	16
1.5.4	Variants	18
1.6	Heap Sort	18
1.6.1	Idea	18
1.6.2	Description and Pseudo Code	18
1.6.3	Complexity Evaluation	24
1.6.4	Variants	25
1.7	Merge Sort	25
1.7.1	Idea	25
1.7.2	Description and Pseudo Code	25
1.7.3	Complexity evaluation	27
1.7.4	Variants	28
1.8	Quick Sort	28
1.8.1	Idea	28
1.8.2	Description and Pseudo Code	28
1.8.3	Complexity evaluation	30
1.8.4	Variants	31
1.9	Counting Sort	32
1.9.1	Idea	32
1.9.2	Description and Pseudo Code	32

1.9.3	Complexity evaluation	33
1.9.4	Variants	34
1.10	Radix Sort	34
1.10.1	Idea	34
1.10.2	Description and Pseudo Code	34
1.10.3	Complexity evaluation	35
1.10.4	Variants	36
1.11	Flash Sort	36
1.11.1	Idea	36
1.11.2	Description and Pseudo Code	36
1.11.3	Complexity evaluation	38
1.11.4	Variants	38
2	Experimental results and comments	39
2.1	Experimental computer's specification	39
2.1.1	How did we run experiments	39
2.2	Experimental results	40
2.2.1	Sorted data	40
2.2.2	Nearly sorted data	41
2.2.3	Reversed sorted data	42
2.2.4	Randomized data	43
2.3	Line graphs	44
2.3.1	Sorted data	44
2.3.2	Nearly sorted data	45
2.3.3	Reverse sorted data	46
2.3.4	Randomized data	47
2.4	Bar charts	48
2.4.1	Sorted data	48
2.4.2	Nearly sorted data	49
2.4.3	Reverse sorted data	50
2.4.4	Randomized data	51
2.5	Conclusions	52
2.5.1	Speed	52
2.5.2	Stability	52
2.5.3	Data Size	52
3	Project organization and Programming notes	53
3.1	Project organization	53
3.1.1	Sorting	53
3.1.2	Utilities	54
3.1.3	Detail function usage in each header file	54
3.2	Programming notes	56
3.2.1	Libraries used	56
3.2.2	How to build	56
3.2.3	Running example	57

4	List of reference	58
5	AI & tools usages	59

1 Algorithm presentation

1.1 Selection Sort

1.1.1 Idea

The idea behind **Selection Sort** is to iteratively build a sorted array by selecting the smallest (or largest) element from the unsorted portion of the array and placing it in its correct position in the sorted array. This process continues until all elements are sorted.

1.1.2 Description and Pseudo Code

Step by step description

Step 1: Start with the entire list of elements to be sorted.

Step 2: Find the minimum (or maximum) element in the unsorted part.

Step 3: Once the end of the unsorted part is reached, swap the found minimum (or maximum) element with the first element of the unsorted part of the list.

Step 4: The first element of the unsorted part is now considered sorted.

Step 5: Repeat steps 2 to 4, with the new unsorted part

Step 6: Continue this process until the entire list is sorted.

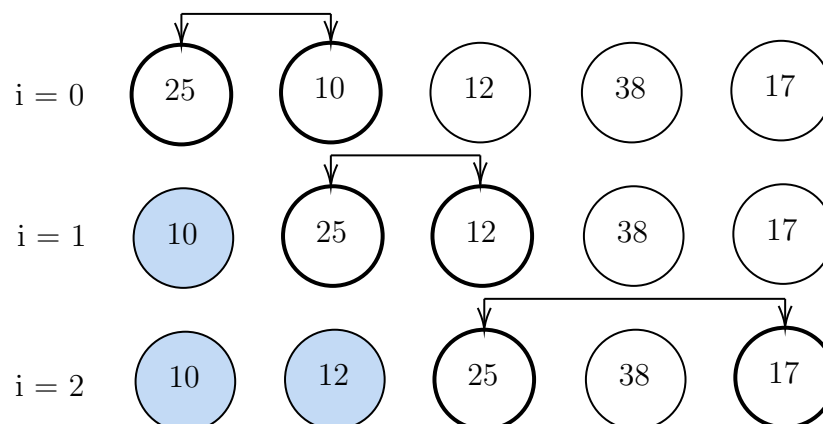
Algorithm 1 Selection Sort

```

1: for  $i \leftarrow 0$  to  $n - 2$  do
2:    $min \leftarrow i$ 
3:   for  $j \leftarrow i + 1$  to  $n - 1$  do
4:     if  $A[j] < A[min]$  then
5:        $min \leftarrow j$ 
6:     end if
7:   end for
8:   If  $min \neq i$  then swap  $A[i]$  and  $A[min]$ 
9: end for

```

Below is the first three passes of the Selection Sort algorithm in sorting $\{25, 10, 12, 38, 17\}$:



In the first pass, $A[\text{min}] = 10$ will be swapped with $A[0] = 25$. Next, $A[\text{min}] = 12$ is swapped with $A[1] = 25$. By keeping doing the algorithm repeatedly, the sorted region will be expanded (the blue part in the image above), as the result, the entire list are sorted.

1.1.3 Complexity evaluation

The analysis of **Selection Sort** is obvious, assume the size of the list is n and the basic operation is the comparison $A[j] < A[\text{min}]$ then we can evaluate the maximum number of times it is executed depends on the size of the list:

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

However, in the best case, when the list has already sorted, the swapping operation only works exact $n - 1$ times:

Time complexity			Space complexity
Worst case	Average case	Best case	Worst case
$O(n^2)$	$\Theta(n^2)$	$\Omega(n)$	$O(1)$

Therefore, **Selection Sort** is well-suited for small datasets due to its simplicity, minimal overhead, and predictable performance. Despite its $O(n^2)$ time complexity, the number of comparisons and swaps remains manageable for small datasets. Its in-place sorting nature requires only $O(1)$ additional memory, making it space-efficient. These factors make **Selection Sort** a practical and easy-to-implement choice for small-scale sorting tasks.

1.1.4 Variants

There are three major and noteworthy variants of **selection sort**:

Heap sort: This variant significantly enhances the basic algorithm by utilizing an implicit heap data structure, which speeds up the process of finding and removing the smallest element.

Bidirectional selection sort: Also known as double selection sort or sometimes cocktail sort due to its resemblance to cocktail shaker sort, this variant finds both the minimum and maximum values in the list during each pass.

Bingo sort: In this variant, items are sorted by repeatedly scanning the remaining items to find the greatest value and moving all items with that value to their final position. Similar to counting sort, this variant is efficient when there are many duplicate values: selection sort makes one pass through the remaining items for each item moved, while Bingo sort makes one pass for each value. After an initial pass to find the greatest value, subsequent passes relocate every item with that value to its final position while identifying the next value.

These variants illustrate the diversity in strategies and optimizations applied to **Selection Sort**, each tailored to address specific characteristics of the input data and improve sorting efficiency in different contexts.

1.2 Insertion Sort

1.2.1 Idea

Insertion Sort is comparison sort algorithm that basically works on inserting the elements right to the correct position. Just like the Selection Sort algorithm above, the list will be divided into two regions: sorted and unsorted parts by an imaginary wall.

Then we repeatedly insert the element in unsorted part to the right position in the sorted part until the entire list is fully sorted.

The way Insertion Sort algorithm works is basically the same as the way we insert new card to our sorted deck of cards, just like the way we insert the card element 7 from the "unsorted region" to the right position in our "sorted region":



1.2.2 Description and Pseudocode

Step by step description

Step 1: Set the increment value is $i = 1$

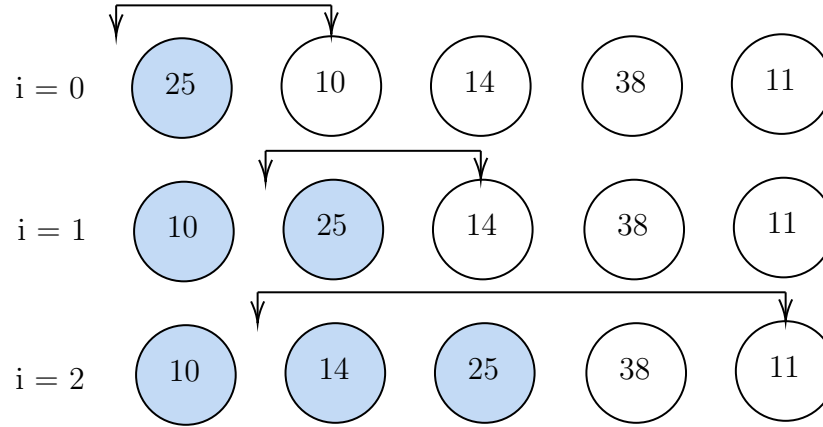
Step 2: Find the correct position for $A[i]$, then increase i by 1

Step 3: Check whether the end of the array is reached by the comparison $i \leq n - 1$. If the comparison is true, go back to Step 2, else the algorithm stops

Algorithm 2 Insertion Sort

```
1: for  $i \leftarrow 1$  to  $n - 1$  do
2:    $key \leftarrow A[i]$ 
3:    $j \leftarrow i - 1$ 
4:   while  $j \geq 0$  and  $A[j] > key$  do
5:      $A[j + 1] \leftarrow A[j]$ 
6:      $j \leftarrow j - 1$ 
7:   end while
8:    $A[j + 1] \leftarrow key$ 
9: end for
```

Below is the first three passes of Insertion Sort algorithm in sorting the list $\{25, 10, 14, 38, 11\}$



The blue part is the sorted region of the list, in each pass, the first element in the unsorted region will be inserted to the right position in sorted region.

1.2.3 Complexity evaluation

The basic operation of the algorithm is the comparison $A[j] > key$ since comparison $j \geq 0$ can be replaceable. Assume the size of the list is n , we can evaluate the number of key comparisons the algorithm makes based on n . In the worst case, input is a list with decreasing values:

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

In the best case, the number of key operations is similar to Selection Sort, the comparison $A[j] > key$ is executed only once in each iteration of the outer loop:

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

For the average case(base on [1]), we state that \overline{C}_i is the expected number of key operation inside the i -th loop, we have to calculate the average total number:

$$\overline{C} = \sum_{i=1}^{n-1} \overline{C}_i$$

Since the iteration is at the i -th loop, the sorted region has i elements in total and we have to insert the $A[i]$ right into the correct position.

In a deeper analysis, if $A[i]$ moves $M_{ij} = i - j$ steps to insert before $A[j]$, the number of comparison $C_{ij} = i - j + 1$ for all $j = i, i - 1, \dots, 1$. However when the element $A[i]$ is inserted right to the top of the list, the number of movement and comparison are equal: $M_{i0} = C_{i0} = i$

Therefore, the formula for \overline{C}_i is the number of possible comparison by moving $A[i]$ divided by the number of movement $i + 1$:

$$\overline{C}_i = \frac{1}{i+1} \sum_{j=0}^i C_{ij} = \frac{1}{i+1} \left(\sum_{j=1}^i (i-j+1) + i \right) = \frac{1}{i+1} \left[\frac{i(i+1)}{2} + i \right] = \frac{i}{2} + \left(1 - \frac{1}{i+1} \right)$$

$$\Rightarrow \overline{C} = \sum_{i=1}^{n-1} \overline{C}_i = \sum_{i=1}^{n-1} \left(1 + \frac{i}{2} - \frac{1}{i+1} \right) = n - 1 + \frac{n(n-1)}{4} - \sum_{i=1}^{n-1} \frac{1}{i+1} = n + \frac{n(n-1)}{4} - H_n$$

With H_n is n-th harmonic number that satisfied the following inequality:

$$\ln(n) + 1 > H_n > \ln(n) \Rightarrow H_n \in \Theta(\log n)$$

which will result in the final conclusion to the average case of Insertion Sort: $\overline{C} \in \Theta(n^2)$

To sum up, the time complexity and space complexity of Insertion Sort can be represented as the below table:

Time complexity			Space complexity
Worst case	Average case	Best case	Worst case
$O(n^2)$	$\Theta(n^2)$	$\Omega(n)$	$O(1)$

1.2.4 Variants and improved versions

In the improved version, to insert to the correct position, we can replace linear search with binary search. The improved version is called "Binary Insertion Sort":

Algorithm 3 Binary Insertion Sort

```

1: for  $i \leftarrow 1$  to  $n - 1$  do
2:    $key \leftarrow A[i]$ 
3:    $first \leftarrow 0$  and  $last \leftarrow i - 1$ 
4:   while  $first \leq last$  do
5:      $mid \leftarrow (first + last)/2$ 
6:     if  $key < A[mid]$  then
7:        $last \leftarrow mid - 1$ 
8:     else
9:        $first \leftarrow mid + 1$ 
10:    end if
11:  end while
12:  for  $j \leftarrow i - 1$  to  $first$  do
13:     $A[j + 1] = A[j]$ 
14:  end for
15:   $A[first] \leftarrow key$ 
16: end for

```

1.3 Bubble Sort

Bubble Sort is one of the simplest comparison sort that basically bases on comparing the adjacent element as the main operation. The word "bubble" from its name is the way elements eventually bubble up to their proper positions.

1.3.1 Idea

The main idea of Bubble Sort algorithm is straightforward, the list of number is divided into two part: sorted part and unsorted part. Then compare the adjacent elements in the unsorted part and swap them if they are out of order. By doing comparison repeatedly, the unsorted part becomes narrow and after going through $n - 1$ passes the entire list is sorted.

1.3.2 Description and Pseudo Code

There are two ways to implement Bubble Sort equivalent to the numbers of suitable locations of sorted part:

Case 1: the sorted part is located at the end of the list and the larger elements are bubbled up first

Case 2: the sorted part is located at the beginning of the list and the smaller elements are bubbled up first

Each case will have a different swapping condition:

Case 1: the largest elements are bubbled up to the end of the list

If an element is larger than the behind ones, they are swapped

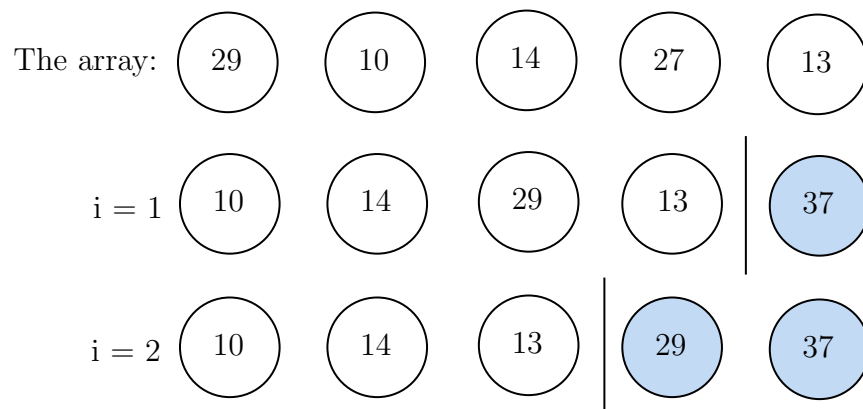
Algorithm 4 Bubble Sort($A[0, \dots, n-1]$) Ver 1

```

1: for  $i \leftarrow 1$  to  $n - 1$  do
2:   for  $j \leftarrow 0$  to  $n - i - 1$  do
3:     if  $A[j] > A[j + 1]$  then
4:       Swap ( $A[j], A[j + 1]$ )
5:     end if
6:   end for
7: end for

```

Below is the first two passes of Bubble Sort algorithm in sorting the list $\{29, 10, 14, 37, 13\}$:



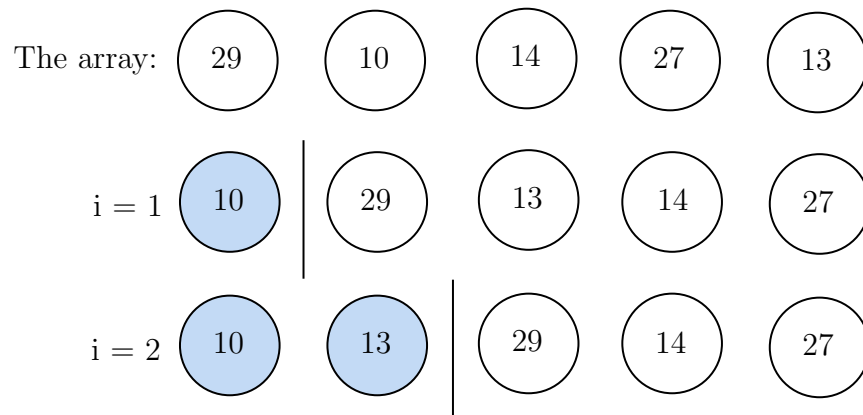
Case 2: the smallest elements are bubbled up to the top of the list:
If an element is smaller than the behind ones, they are swapped

Algorithm 5 Bubble Sort($A[0, \dots, n-1]$) Ver 2

```

1: for  $i \leftarrow 1$  to  $n - 1$  do
2:   for  $j \leftarrow n - 1$  to  $i$  do
3:     if  $A[j] < A[j - 1]$  then
4:       Swap ( $A[j], A[j - 1]$ )
5:     end if
6:   end for
7: end for
  
```

Below is another two first passes of Bubble Sort algorithm in sorting the list $\{29, 10, 14, 37, 13\}$



1.3.3 Complexity evaluation

We can evaluate the number of key comparisons which is the main operation for bubble sort by counting them respectively in nested loops. Assume the number of elements in the array is n and $C(n)$ is the number of key comparison. Without loss of generality, the version of bubble sort we analyse is the first case:

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

And in the worst case, number of swapping operation is the same:

$$S_{worst}(n) = C(n) \in \Theta(n^2)$$

However, if the list has already sorted, the number of swapping operation is $n \in \Theta(n)$ which is the best case for bubble sort.

Time complexity			Space complexity
Worst case	Average case	Best case	Worst case
$O(n^2)$	$\Theta(n^2)$	$\Omega(n)$	$\Theta(1)$

Although Bubble Sort is easy for programmers to implement and no other storage needed, but in practise, when the amount of data is huge, bubble sort will perform worse and slower than other preferable algorithms such as Quick Sort, Heap Sort, Merge Sort,... which are implemented on several programming language sorting functions.

1.3.4 Varient and Improved version

However, as following the brute-force strategy, the above version of Bubble Sort is not efficiency since the list can be sorted just in a few passes. Thus, there will be unnecessary loops and we have to stop the loops when the list has already sorted using variable flag:

Algorithm 6 Bubble Sort (improved version)

```
1: unsorted  $\leftarrow$  true
2: pass  $\leftarrow$  0
3: while unsorted = true do
4:   unsorted  $\leftarrow$  false
5:   pass  $\leftarrow$  pass + 1
6:   for j  $\leftarrow$  0 to n - pass - 1 do
7:     if A[j] > A[j + 1] then
8:       Swap (A[j], A[j + 1])
9:       unsorted  $\leftarrow$  true
10:    end if
11:  end for
12: end while
```

With this Optimised Bubble Sort, we reduce the time complexity from $O(n^2)$ to $O(n)$ for the list that is already sorted. Another improved version of Bubble Sort is Shaker Sort algorithm which will be mentioned in the next section.

1.4 Shaker Sort

1.4.1 Idea

Shaker Sort, which is also known as Cocktail Sort. The idea behind is to enhance the sorting efficiency by sorting in both directions within each pass through the list. Unlike traditional Bubble Sort, which only moves in one direction, Shaker Sort alternates between forward and backward passes to more quickly move elements to their correct positions

1.4.2 Description and Pseudo Code

Step by step description

Step 1: Start with the entire list of elements to be sorted.

Step 2: Begin with an unsorted portion of the list. Initially, assume the entire list is unsorted.

Step 3: Perform a forward pass through the unsorted portion:

- Compare each pair of adjacent elements.

- If the first element is greater than the second, swap them.
- Continue until the end of the unsorted portion is reached.

Step 4: If any swaps were made during the forward pass, repeat steps 3 and 4. Otherwise, the list is considered sorted.

Step 5: Perform a backward pass through the unsorted portion:

- Compare each pair of adjacent elements, starting from the end of the unsorted portion.
- If the first element is greater than the second, swap them.
- Continue until the beginning of the unsorted portion is reached.

Step 6: If any swaps were made during the backward pass, repeat steps 5 and 6. Otherwise, the list is considered sorted

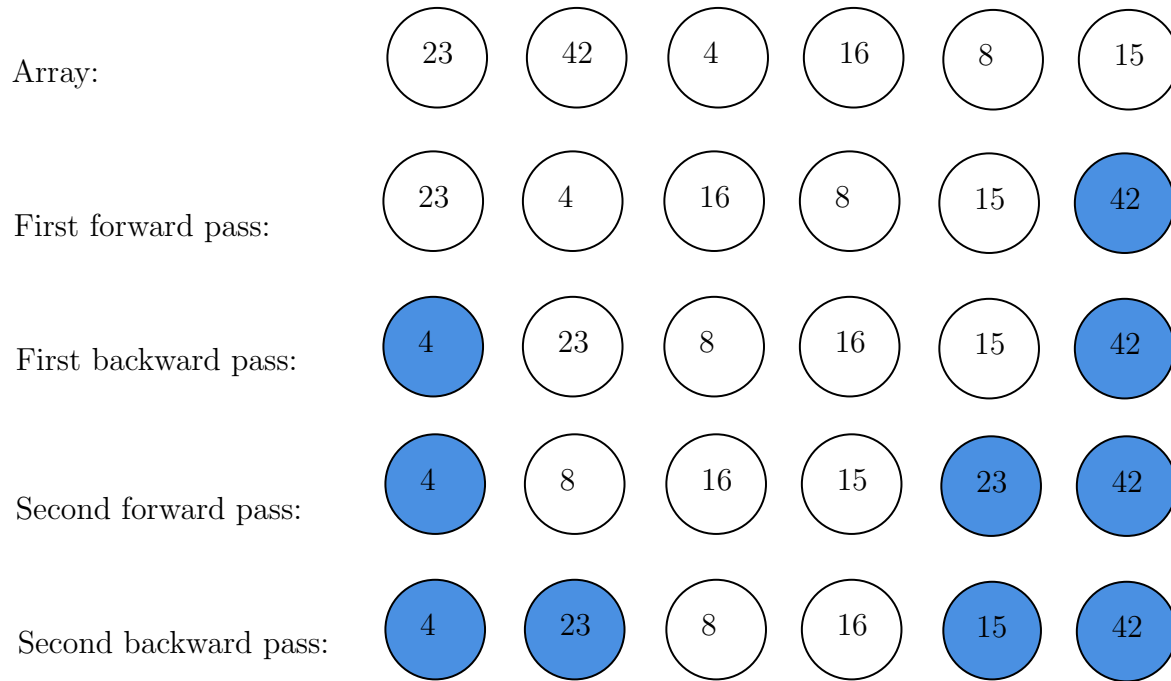
Algorithm 7 Shaker Sort

```

1: left  $\leftarrow$  1 and right  $\leftarrow$  n - 1
2: while left  $\leq$  right do
3:   swapped  $\leftarrow$  false
4:   for i  $\leftarrow$  left to right - 1 do
5:     if A[i] > A[i + 1] then
6:       swap A[i] and A[i + 1]
7:       swapped  $\leftarrow$  true
8:     end if
9:   end for
10:  if not swapped then
11:    break
12:  end if
13:  right  $\leftarrow$  right - 1
14:  swapped  $\leftarrow$  false
15:  for i  $\leftarrow$  right to left step -1 do
16:    if A[i] < A[i - 1] then
17:      swap A[i] and A[i - 1]
18:      swapped  $\leftarrow$  true
19:    end if
20:  end for
21:  if not swapped then
22:    break
23:  end if
24:  left  $\leftarrow$  left + 1
25: end while

```

Below is the first three passes of the Shaker Sort algorithm in sorting $\{23, 42, 4, 16, 8, 15\}$:



After completing a pass in Shaker Sort, whether forward or backward, we need to check if any swaps were made during the pass. If swaps were made, we continue with the next pass in the opposite direction, ensuring that the smallest elements “bubble” up to the start and the largest elements “sink” to the end, progressively narrowing the range of elements that need sorting.

1.4.3 Complexity evaluation

The analysis of **Shaker Sort** is similar to that of Bubble Sort, but with the added improvement of moving in both directions. Assume the size of the list is n and the basic operation is the comparison $A[j] > A[j + 1]$. We can evaluate the maximum times it is executed: In each pass, both a forward and a backward traversal occur. For simplicity, let's analyze the forward traversal and note that the backward traversal will have a similar complexity.

Forward Pass: In the worst case, the first forward pass makes $n - 1$ comparisons, the second forward pass makes $n - 3$ comparisons, and so on. This can be expressed as:

$$\sum_{i=0}^{\lfloor \frac{n-1}{2} \rfloor - 1} (n - 2i - 1)$$

Backward Pass: Similarly, the backward passes also reduce the number of comparisons in a similar manner. Therefore, the total number of comparisons for both passes can be approximated as:

$$2 \sum_{i=0}^{\lfloor \frac{n-1}{2} \rfloor - 1} (n - 2i - 1)$$

This sum is simplified to a quadratic complexity, which can be shown as:

$$C(n) \approx \sum_{i=0}^{n-1} (n - i - 1) \approx \sum_{i=1}^n i = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Thus, the worst-case time complexity of Shaker Sort is: $\Theta(n^2)$. In the best case, where the list is already sorted, Shaker Sort can detect this in one forward and backward pass, giving it a best-case time complexity of: $\Theta(n)$

Time complexity			Space complexity
Worst case	Average case	Best case	Worst case
$O(n^2)$	$\Theta(n^2)$	$\Omega(n)$	$O(1)$

In conclusion, Shaker Sort offers a slight improvement over Bubble Sort by traversing the list in both directions. Despite this improvement, its worst-case time complexity remains $\Theta(n^2)$, making it inefficient for large datasets. However, in the best-case scenario of an already sorted list, Shaker Sort achieves an optimal time complexity of $\Theta(n)$, demonstrating its ability to recognize sorted order quickly.

1.4.4 Variants and improved versions

Comb Sort is considered an improvement over Shaker Sort primarily due to its more aggressive gap reduction strategy, which allows it to potentially make more efficient passes through the array.

Algorithm 8 Comb Sort

```

1: gap ← n ← length(A)
2: shrink ← 1.3
3: sorted ← false
4: while sorted is false do
5:   gap ← floor(gap / shrink)
6:   if gap > 1 then
7:     sorted ← false
8:   else
9:     gap ← 1
10:    sorted ← true
11:  end if
12:  i ← 0
13:  while i + gap < n do
14:    if A[i] > A[i + gap] then
15:      swap(A[i], A[i + gap])
16:      sorted ← false
17:    end if
18:    i ← i + 1
19:  end while
20: end while

```

1.5 Shell Sort

1.5.1 Idea

Shell sort, named after its inventor Donald Shell, is a sorting algorithm that builds on the insertion sort method by allowing the comparison and exchange of elements that are far apart before progressively reducing the gap between elements to sort the entire list. By starting with a larger gap and continually reducing it, Shell sort aims to reduce the amount of work needed to move elements to their correct positions, making it more efficient than simple insertion sort for larger datasets.

1.5.2 Description and Pseudo Code

Step by step Description

Step 1: Start with the array of elements to be sorted.

Step 2: Begin with an initial gap size. Typically, set gap to $\lfloor \frac{n}{2} \rfloor$, where n is the number of elements in the array.

Step 3: Iterate over the gap sizes, reducing the gap size by half each time ($gap \leftarrow \lfloor \frac{gap}{2} \rfloor$).

Step 4: For each gap size, perform a gapped insertion sort:

- Start from the element at index gap and compare it with the elements that are at gap positions before it.
- If the element at index $j - gap$, (j minus gap), is greater than the current element, swap them and continue this process until the correct position for the current element is found.
- Move to the next element in the gap-sorted sequence.

Step 5: Continue sorting each gap-sized subarray until the entire array is sorted for the current gap size.

Step 6: Repeat steps 3-5 until the gap size is 1. This final pass is a standard insertion sort that ensures the array is completely sorted.

1.5.3 Complexity evaluation

The analysis of **Shell Sort** involves considering the sequence of gaps used and their impact on sorting efficiency. Let's denote the size of the list as n .

Gap Sequence Impact: Shell Sort starts with a gap size of $\lfloor \frac{n}{2} \rfloor$ and progressively reduces the gap until it reaches 1. The efficiency of Shell Sort heavily depends on the gap sequence chosen. Common sequences like $\lfloor \frac{n}{2} \rfloor$, $\lfloor \frac{n}{4} \rfloor$, $\lfloor \frac{n}{8} \rfloor$, ..., or sequences derived from mathematical formulas can influence its performance.

Algorithm 9 Shell Sort

```

1: gap  $\leftarrow \lfloor \frac{n}{2} \rfloor$  ▷ Start with a big gap, then reduce
2: while gap > 0 do
3:   for i  $\leftarrow$  gap to n - 1 do
4:     temp  $\leftarrow$  arr[i]
5:     j  $\leftarrow$  i
6:     while j  $\geq$  gap and arr[j - gap] > temp do
7:       arr[j]  $\leftarrow$  arr[j - gap]
8:       j  $\leftarrow$  j - gap
9:     end while
10:    arr[j]  $\leftarrow$  temp
11:  end for
12:  gap  $\leftarrow \lfloor \frac{gap}{2} \rfloor$  ▷ Reduce the gap
13: end while

```

Worst-case Time Complexity: In the worst case, the time complexity of Shell Sort is approximately:

$$\Theta(n^2)$$

This occurs when the gap sequence does not effectively reduce the number of inversions quickly enough. However, practical implementations with well-chosen gap sequences often perform better than this worst-case scenario.

Average-case Time Complexity: The average-case time complexity of Shell Sort is generally approximated as:

$$\Theta(n^{1.5})$$

Empirical studies and theoretical analyses suggest that Shell Sort tends to perform better than quadratic sorting algorithms like Insertion Sort or Bubble Sort on average.

Best-case Time Complexity: In the best case, Shell Sort can achieve a time complexity of:

$$\Theta(n \log n)$$

This occurs when the gap sequence and input data are such that the array requires a small number of comparisons and shifts.

Time complexity			Space complexity
Worst case	Average case	Best case	Worst case
$O(n^2)$	$\Theta(n^{1.5})$	$\Omega(n \log n)$	$O(1)$

The space complexity of Shell Sort is $\Theta(1)$, meaning it sorts the array in-place without requiring additional memory proportional to the input size.

1.5.4 Variants

Shell sort is an enhancement of insertion sort that allows elements to move over larger distances. It uses a gap sequence to determine which elements to compare and swap, starting with large gaps and reducing them until a final insertion sort is performed with a gap of 1. Variants of Shell sort use different sequences, such as **Hibbard's**, **Sedgewick's**, **Knuth's**, and **Tokuda's**, each affecting the algorithm's performance. Improvements include adaptive gap sequences, parallelization, optimized insertion sorts, cache optimization, and hybrid algorithms. These enhancements aim to improve efficiency and performance, making Shell sort adaptable to various data sets and modern computing environments.

1.6 Heap Sort

1.6.1 Idea

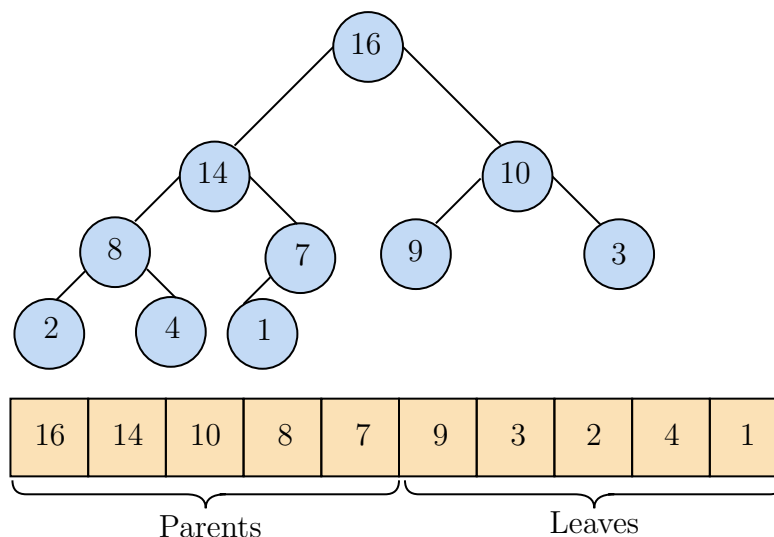
The idea of **Heap Sort** begins by constructing a max heap from the array. It repeatedly swaps the root element (maximum value) with the last element of the heap, reduces the heap size by one, and ensures the heap property (max heap) is maintained through heapifying. This process continues until the entire array is sorted in ascending order.

1.6.2 Description and Pseudo Code

Heap Structure

Before moving to the deep description of Heap Sort algorithm, we review a little bit about Heap data structure. Heap can be defined as a binary tree with key assigned to its node satisfied the two following conditions:

- **Shape condition:** The binary tree is complete all its levels are full except the leaves
- **Parental dominance condition:** the key in each node is greater than or equal to the its children's key



Properties of Heap [2]

Below is the list of importance properties of Heap:

- There exists only one essentially complete binary tree with n given nodes. Its height is equal to $\lfloor \log_2(n) \rfloor$
- The root of a heap always contain the largest element
- The index of all leaves in the heap structure of n nodes is $\lfloor \frac{n-1}{2} \rfloor + 1, \lfloor \frac{n-1}{2} \rfloor + 2, \dots, n-1$

Max Heap and Min Heap

There are two types of Binary Heap structure: Max Heap and Min Heap. As their name, Max Heap is Heap structure satisfied that parent nodes is larger or equal to child node, Min Heap is Heap structure satisfied that parent nodes is smaller or equal to child node.

Heapify Operation

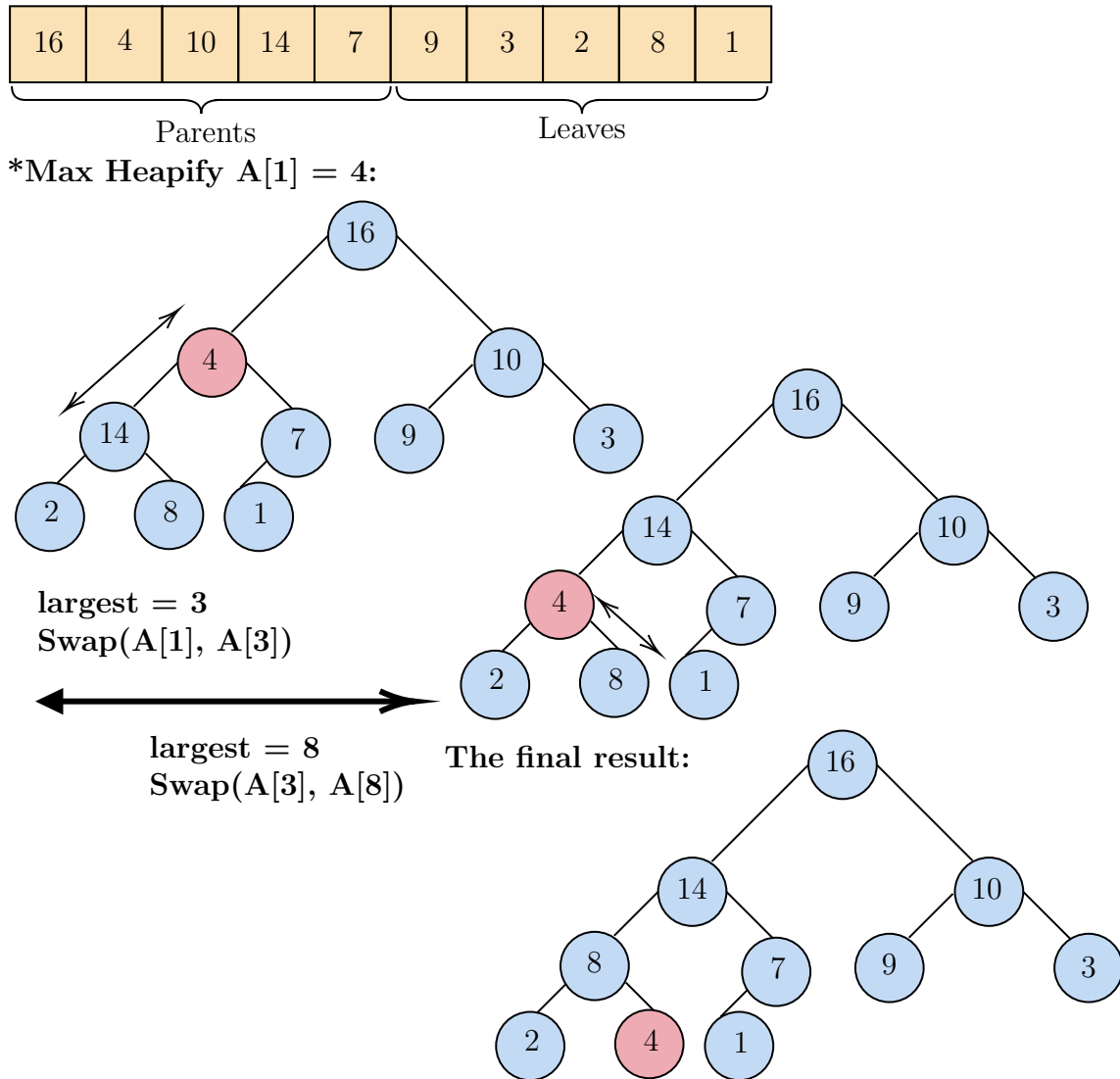
However, the Heap structure that are constructed from list of key sometimes can't be Max Heap or Min Heap. Thus, to maintain the property of Max Heap or Min Heap, we need an operation called heapify. In this section, we will focus on max heapify as the main operation.

Max heapify is an operation to find the largest elements among $A[i]$, $A[Left(i)]$, $A[Right(i)]$ then swap $A[i]$ with $A[largest]$ and continue the process with the new largest index:

Algorithm 10 Heapify($A, n, largest$)

```

1:  $largest \leftarrow i$ 
2:  $left \leftarrow 2 * i + 1$  and  $right \leftarrow 2 * i + 2$ 
3: if  $left < n$  and  $A[left] > A[largest]$  then
4:    $largest \leftarrow left$ 
5: end if
6: if  $right < n$  and  $A[right] > A[largest]$  then
7:    $largest \leftarrow right$ 
8: end if
9: if  $largest \neq i$  then
10:   Swap  $A[i]$  with  $A[largest]$ 
11:   HEAPIFY( $A, n, largest$ )
12: end if
```



Build Max Heap from heapify operation

From a normal Heap structure, we can build up Max Heap structure using the algorithm by keeping Heapify node index $\lfloor \frac{n}{2} \rfloor$ to the root

Algorithm 11 Build Max Heap

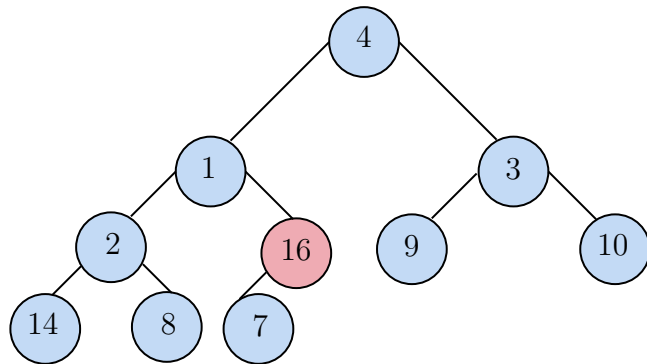
```

1: procedure MAX HEAP(arr, n)
2:   for  $i \leftarrow \lfloor \frac{n}{2} \rfloor - 1$  to 0 do
3:     HEAPIFY(arr, n, i)
4:   end for
5: end procedure

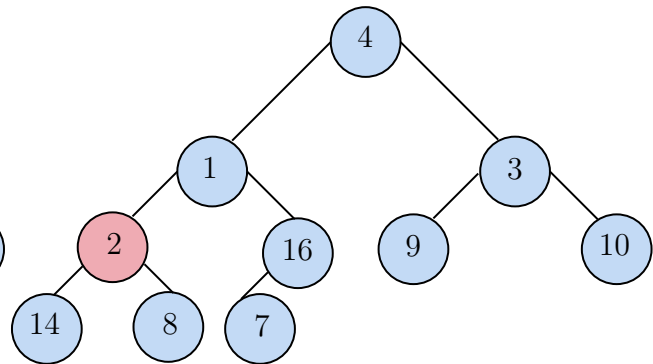
```

Below is the Heap Sort algorithm for sorting the list: {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}:

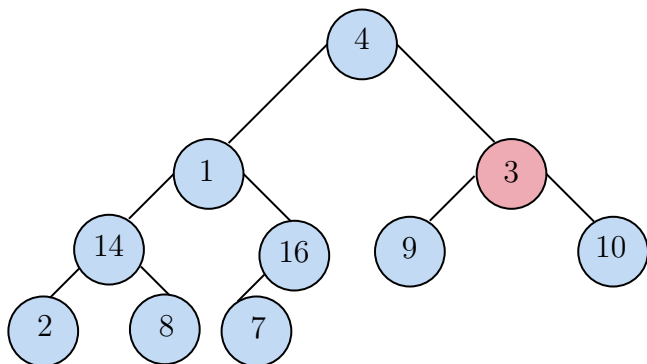
*Max Heapify A[4] = 16:



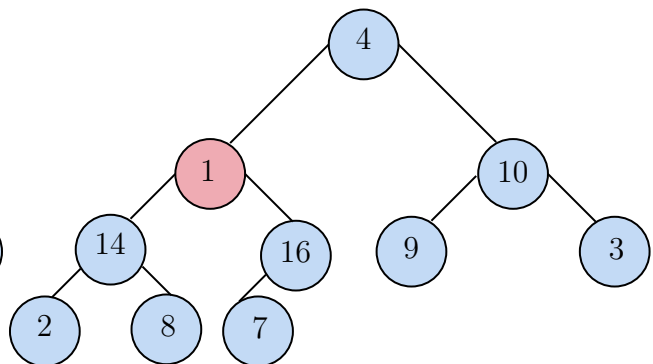
*Max Heapify A[3] = 2:



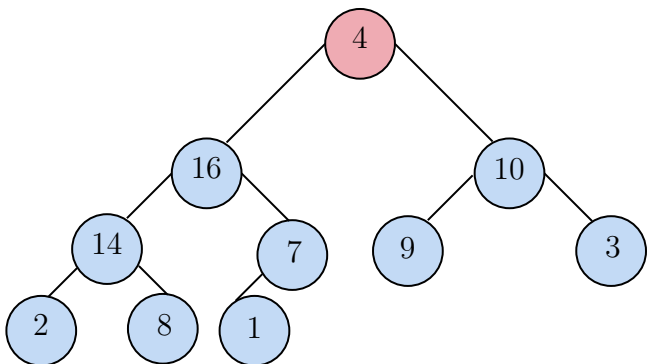
*Max Heapify A[2] = 3:



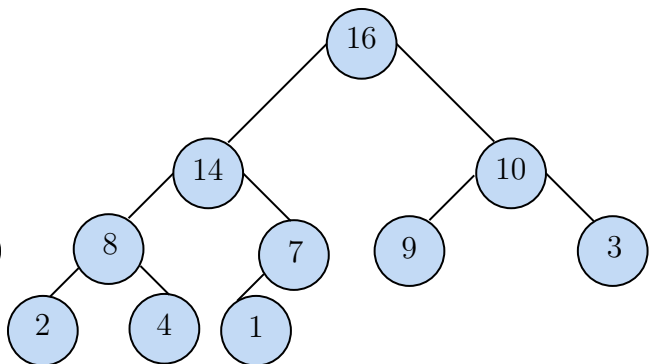
*Max Heapify A[1] = 1:



*Max Heapify A[0] = 4:



*Max Heap build:



Step by step description of Heap Sort algorithm

Step 1: **Build the Heap:** The first step of Heap Sort is to build a heap from the input array. The heap is a binary tree that satisfies the heap property, which is that for every node, its value must be greater (for max-heap) or smaller (for min-heap) than or equal to the values of its children. The heap can be built in place within the input array.

Step 2: **Max-Heapify (or Min-Heapify):** To build a heap, we need to ensure the heap property is maintained for each subtree rooted at an index i . The Max-Heapify (or Min-Heapify) algorithm is used to achieve this. It compares the parent node with its children and swaps them if necessary to satisfy the heap property. The process continues recursively until the entire heap is built.

- Step 3: **Heapify the Array:** Starting from the last non-leaf node (index $n/2 - 1$) to the root (index 0), apply Max-Heapify (or Min-Heapify) to each node to convert the array into a valid heap
- Step 4: **Extract Elements:** After building the heap, the largest (for max-heap) or smallest (for min-heap) element will be at the root of the heap (index 0). Swap this element with the last element in the array (index $n-1$).
- Step 5: **Heap Size Reduction:** Decrease the heap size by one (heap size = heap size - 1) to exclude the last element, which is now in its correct sorted position.
- Step 6: **Maintain Heap Property:** To maintain the heap property, apply Max-Heapify (or Min-Heapify) to the root element (index 0) again.
- Step 7: **Repeat Extraction:** Repeat steps 4 to 6 until the heap size is reduced to 1. After each iteration, the largest (for max-heap) or smallest (for min-heap) element will be placed at the end of the array.

Pseudocode of Heap Sort

Algorithm 12 Heap Sort

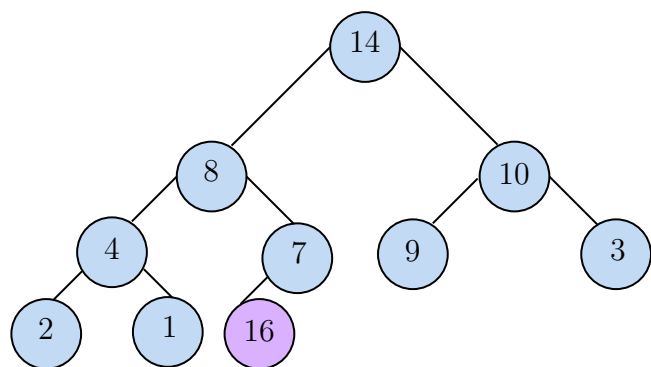
```

1: procedure HEAPSORT( $arr, n$ )
2:   MAX_HEAP( $arr, n$ )
3:   for  $i \leftarrow n - 1$  downto 1 do
4:     SWAP( $arr[0], arr[i]$ )
5:     HEAPIFY( $arr, i, 0$ )
6:   end for
7: end procedure

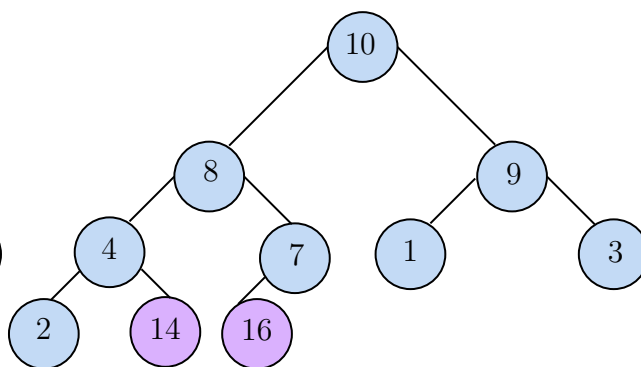
```

The next page will simulate the process of Heap Sort algorithm in sorting the list of key $\{14, 8, 10, 4, 7, 9, 3, 2, 1, 16\}$ by keeping the process of building Max Heap and swapping the root element with the furthest leaf.

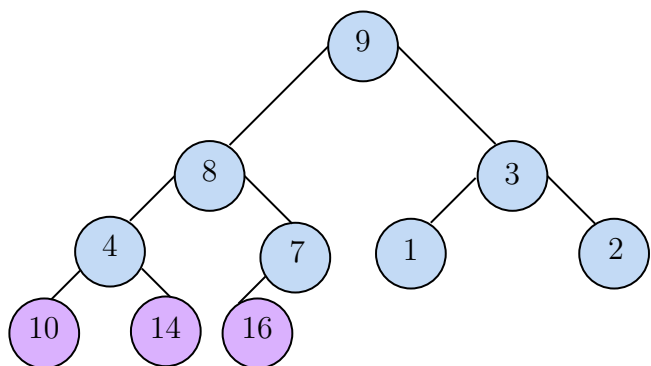
i = 9:



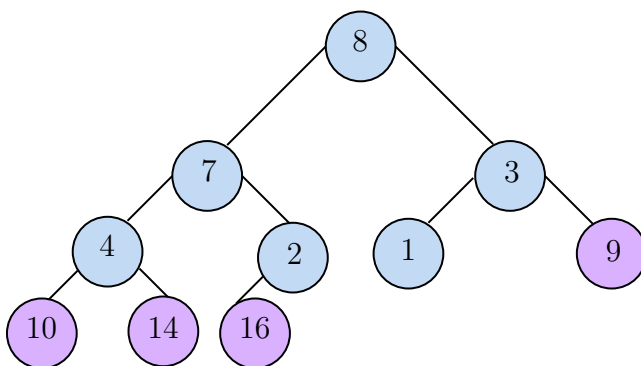
i = 8:



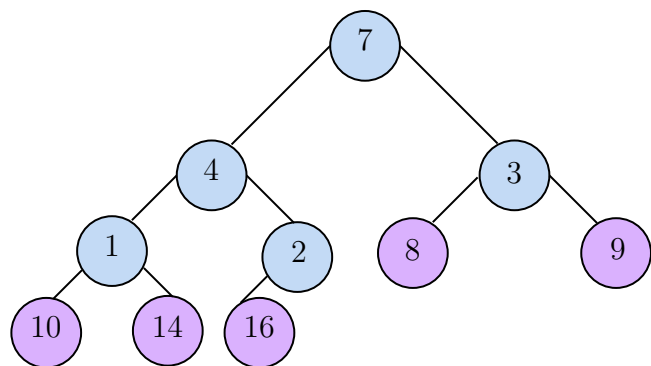
i = 7:



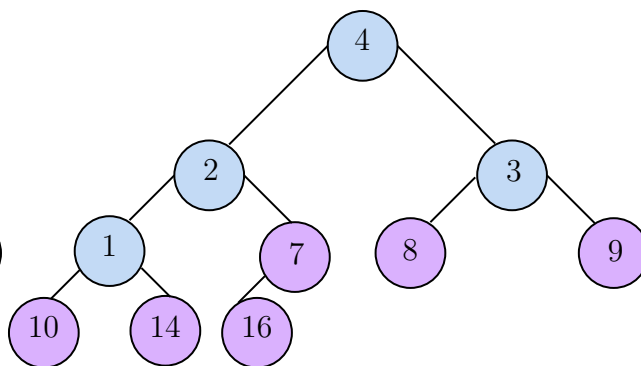
i = 6:



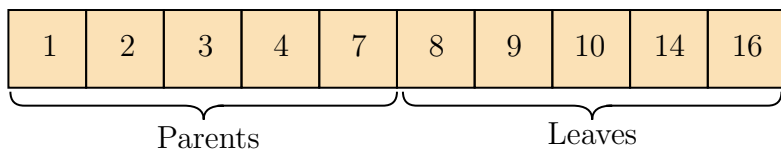
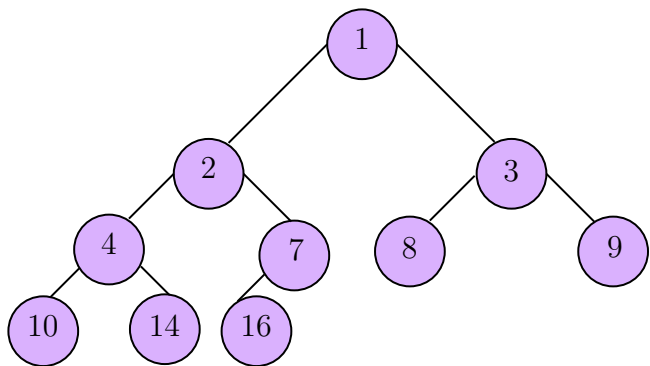
i = 5:



i = 4:



i = 3, 2, 1:



1.6.3 Complexity Evaluation

To evaluate the time complexity of Heap Sort we have to separately research on the two steps of Heap Sort algorithm: Max Heap construction and Repeat Extraction.[3]

Build Max Heap

For simplicity, we assume $n = 2^k - 1$ so that the entire heap structure is full. Let h becomes the height of the heap, as the property of heap above, we can find the relationship between h and k :

$$h = \lfloor \log_2 n \rfloor = \lceil \log_2(n+1) \rceil - 1 = k - 1$$

For the worst case, the node with i level will have to travel to the leaf at level h which requires $2(h-i)$ comparisons. Thus, the total comparison in the worst case for building the Max Heap is:

$$C_{worst}(n) = \sum_{i=0}^{h-1} \sum_{\text{level } i \text{ keys}} 2(h-i) = \sum_{i=0}^{h-1} 2(h-i)2^i$$

$$\Leftrightarrow C_{worst}(n) = h \sum_{i=0}^{h-1} 2^{i+1} - \sum_{i=0}^{h-1} i2^i = h(2^{h+1} - 2) - (h-2)2^h - 2 = 2(n - \log_2(n+1)) \in \Theta(n)$$

Repeat Extraction

For the second stage, the number of key comparison satisfied the inequality:

$$C(n) \leq 2 \sum_{i=1}^{n-1} \lfloor \log_2(i) \rfloor \leq \sum_{i=1}^{n-1} \log_2(i) \leq 2 \sum_{i=1}^{n-1} \log_2(n-1) = 2(n-1) \log_2(n-1) \in O(n \log n)$$

In overall, the time complexity is $O(n) + O(n \log n) = O(n \log n)$ but in the deeper evaluation we conclude:

Time complexity			Space complexity
Worst case	Average case	Best case	Worst case
$O(n \log n)$	$\Theta(n \log n)$	$\Omega(n \log n)$	$O(1)$

In conclusion, Heap Sort is a highly efficient sorting algorithm that leverages the binary heap data structure to achieve consistent $O(n \log n)$ time complexity for sorting arrays. Its ability to sort in-place, without requiring additional space proportional to the input size, makes it particularly suitable for scenarios where memory usage is a concern.

Despite its efficient performance in terms of time complexity, Heap Sort may not be the first choice for small datasets due to its relatively high constant factors compared to simpler algorithms like Insertion Sort or Selection Sort.

However, for large datasets where performance is critical and memory usage needs to be optimized, Heap Sort remains a robust and reliable choice, offering stable sorting with predictable performance characteristics.

1.6.4 Variants

Floyd's Heap Construction: Floyd's heap construction is indeed an efficient algorithm used to convert an array into a heap structure in $O(n)$ time complexity. It works by starting from the last non-leaf node and performing sift-down operations to ensure each subtree rooted at these nodes satisfies the heap property. This method is efficient and widely used in practical implementations of heap sort.

Ternary Heapsort: Ternary heapsort is a variant where each node in the heap has three children instead of two, as in a binary heap. While theoretically, each sift-down operation in a ternary heap might involve fewer comparisons and swaps compared to a binary heap (three comparisons and one swap for ternary vs. two comparisons and one swap for binary), the practical benefits are often minimal or even outweighed by increased complexity in implementation.

1.7 Merge Sort

1.7.1 Idea

Merge Sort is a divide-and-conquer sorting algorithm. It works by splitting the array into two halves, recursively sorting each half, and then merging the sorted halves back together. This continues until the subarrays have only one element (which are already sorted). The merging process involves comparing elements from each subarray and inserting the smaller one into the final sorted array.

1.7.2 Description and Pseudo Code

Step by step description

Step 1: Split the array into two nearly equal parts.

Step 2: Recursively apply **Merge Sort** to each half, dividing them further until they contain only one element.

Step 3: Merge the sorted sub-arrays by comparing the elements and placing them in the correct order.

Step 4: Repeat the process until the initial array become sorted.

Algorithm 13 Merge Sort

```
1: procedure MERGESORT(arr, begin, end)
2:    $mid \leftarrow \lfloor \frac{begin+end+1}{2} \rfloor$ 
3:
4:   Call MergeSort(arr, begin, mid)
5:   Call MergeSort(arr, mid + 1, end)
6:   Call Merge(arr, left, mid, right)
7: end procedure
```

Merging process

Step 1: Initialize two pointer point to the first elements of the arrays being merged.

Step 2: compare two pointed elements, the smaller being added to the new array.

Step 3: the pointer of the smaller element is moved to the next elements.

Step 4: Repeat the process until one of the two given arrays is exhausted

Step 5: The remaining elements of the other array are copied to the end of the new array.

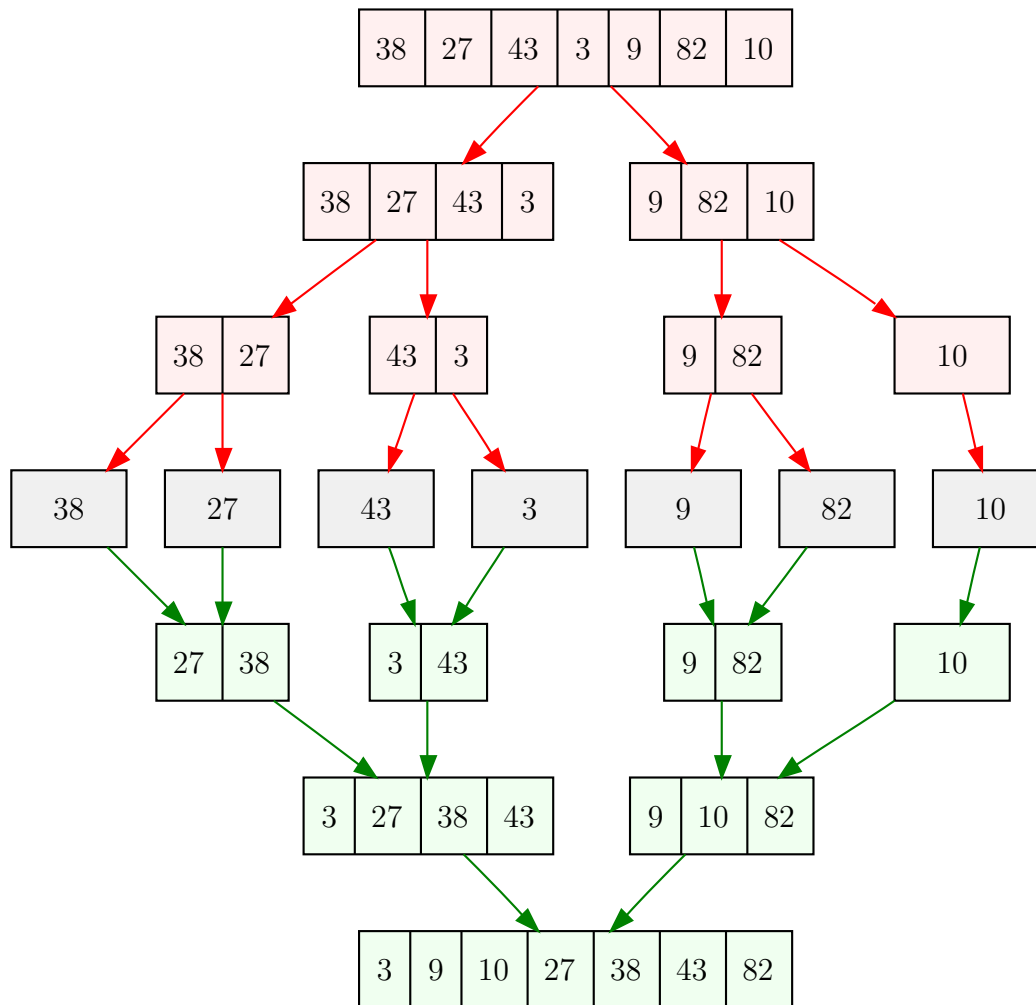
Algorithm 14 Merge

```

1: procedure MERGE(arr, begin, mid, end)
2:    $\text{len1} \leftarrow \text{mid} - \text{begin} + 1$ 
3:    $\text{len2} \leftarrow \text{end} - \text{mid}$ 
4:
5:    $\text{leftArr}[\text{len1}] \leftarrow \text{arr}[\text{begin} \cdots \text{mid}]$  ▷ Create two temporary arrays
6:    $\text{rightArr}[\text{len2}] \leftarrow \text{arr}[\text{mid} + 1 \cdots \text{end}]$ 
7:    $i \leftarrow 0$  ▷ Declares two pointers for merging
8:    $j \leftarrow 0$ 
9:    $\text{index} \leftarrow 0$  ▷ Declares index of the resulting array
10:
11:   while  $i < \text{len1}$  and  $j < \text{len2}$  do ▷ Performs merging
12:     if  $\text{leftArr}[i] \leq \text{rightArr}[j]$  then
13:        $\text{arr}[\text{index}] \leftarrow \text{leftArr}[i]$ 
14:        $i \leftarrow i + 1$ 
15:     else
16:        $\text{arr}[\text{index}] \leftarrow \text{rightArr}[j]$ 
17:        $j \leftarrow j + 1$ 
18:     end if
19:      $\text{index} \leftarrow \text{index} + 1$ 
20:   end while
21:
22:   while  $i < \text{len1}$  do ▷ If there still elements remain in the left halve
23:      $\text{arr}[\text{index}] \leftarrow \text{leftArr}[i]$ 
24:      $i \leftarrow i + 1$ 
25:      $\text{index} \leftarrow \text{index} + 1$ 
26:   end while
27:
28:   while  $j < \text{len2}$  do ▷ If there still elements remain in the right halve
29:      $\text{arr}[\text{index}] \leftarrow \text{rightArr}[j]$ 
30:      $j \leftarrow j + 1$ 
31:      $\text{index} \leftarrow \text{index} + 1$ 
32:   end while
33: end procedure

```

To give a better visualization of merge sort, you can see how the algorithm runs in the below figure, consist of $n = 6$ elements. The black arrows show the splitting process, the red arrows show the merging process.



[21]

1.7.3 Complexity evaluation

Time Complexity: Merge sort has the time complexity of $O(n \log n)$. We can intuitively prove this by looking at the above figure. After each recursion call, the array being split in half, therefore, only about $\lceil \log_2 n \rceil$ layers of recursion. At each layer, the total number of element will not exceed n . With $\lceil \log_2 n \rceil$ layers, each performs one splitting and one merging, and the complexity of splitting and merging are $O(n)$, so the resulting complexity is $O(n \cdot \log n)$.

Space Complexity: Merge sort usually needs an $O(n)$ extra space for the merging process, but there still a version that implements in-place merge sort but with a high constant factor. So in this project, we will consider merge sort has $O(n)$ space complexity.

Time complexity			Space complexity
Worst case	Average case	Best case	Worst case
$O(n \log n)$	$\Theta(n \log n)$	$\Omega(n \log n)$	$O(n)$

1.7.4 Variants

One of the weaknesses of Merge Sort is that it requires additional memory to store subarrays. To address this, we can use variants such as Ping-Pong Merge Sort and Bottom-Up Merge Sort.

Ping-Pong Merge Sort: merges four blocks at a time instead of two. Initially, four sorted blocks are merged simultaneously into two sorted blocks using auxiliary space. These two sorted blocks are then merged back into the main memory. This approach reduces the total number of moves by half and omits the copy operation, making the process more efficient. [17]

Bottom-Up Merge Sort: is an iterative version of Merge Sort that avoids recursion, which reduces memory consumption. It starts by treating each element as a sorted block and iteratively merges adjacent blocks of increasing size until the entire array is sorted. This iterative approach not only saves memory but can also be more cache-friendly, improving performance for large datasets. [18]

1.8 Quick Sort

1.8.1 Idea

The idea of **Quick Sort** is to repeatedly select a pivot element from the array, then partition the array into two sub-arrays, the prefix contains elements that are smaller or equal than the pivot and the suffix contains elements greater than the pivot. The algorithm recursively sorts the sub-arrays with the aforementioned strategies until it reach a recursive call with only one element.

1.8.2 Description and Pseudo Code

Usually, quicksort is implemented in-place, which means it performs swapping operation between its element, the algorithm description is as follow:

- Step 1: We define a sorting procedure which do the following steps on the sub-array indicated by boundaries **L** and **R**.
- Step 2: Enters the procedure, checks if there are more than one elements to sort, else exit the procedure.
- Step 3: **Chooses a pivot** element from the array
- Step 4: **Partitions the array** into two sub-arrays by moving smaller elements to left and greater elements to the right. After that, the sub-arrays will be indicated by its boundaries, two pairs $[L_1, R_1]$ and $[L_2, R_2]$.
- Step 5: **Recursively sort two sub-arrays** above by calling the above procedure with boundaries $[L_1, R_1]$ and $[L_2, R_2]$.

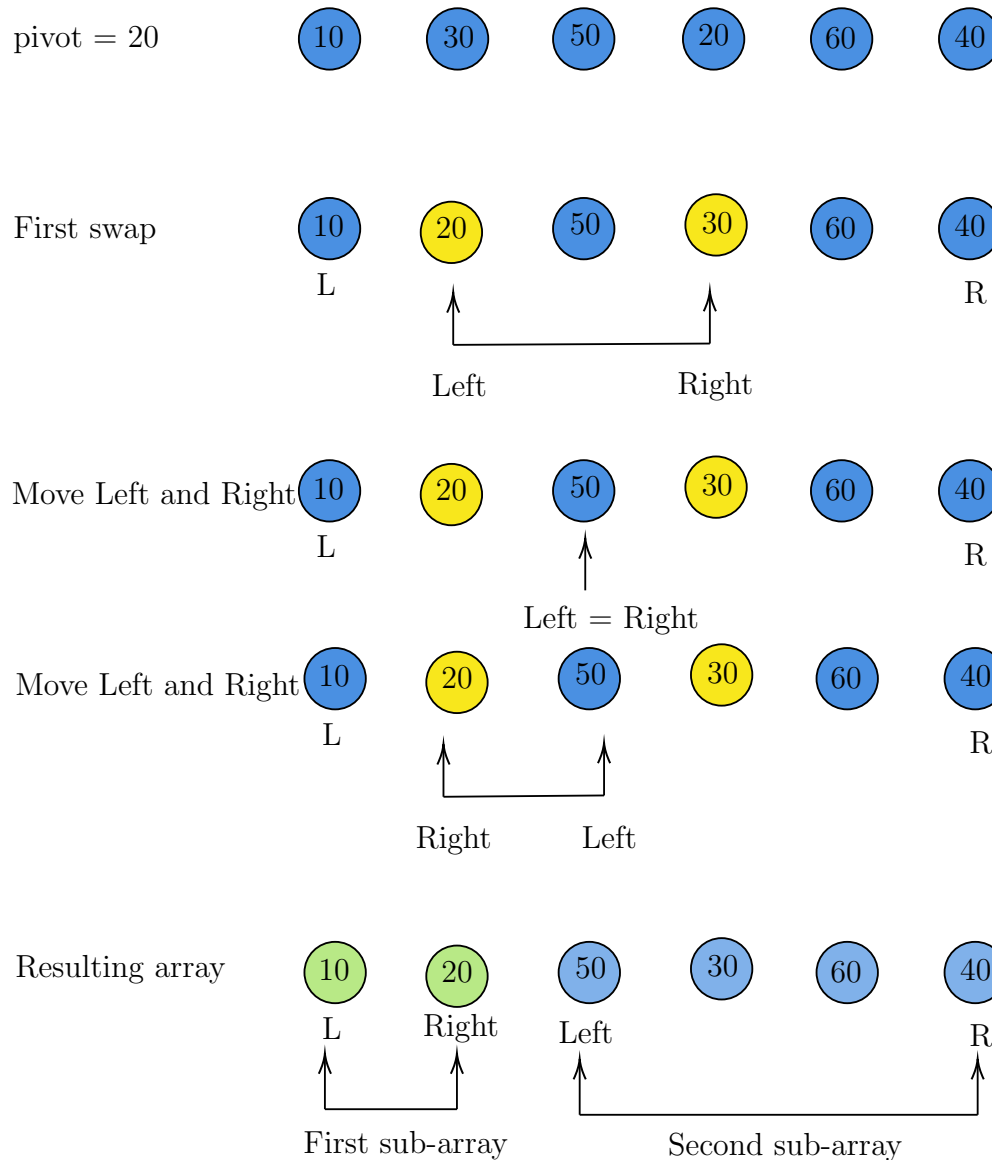
Randomized Quick Sort is an advanced sorting algorithm that improves upon the efficiency of traditional Quick Sort by incorporating a randomized element in its pivot selection strategy. Unlike the deterministic Quick Sort, which typically selects the first or last element as the pivot, Randomized Quick Sort randomly selects a pivot element from the array.

This random selection helps mitigate the worst-case time complexity scenarios that can occur with deterministic pivoting strategies, such as when the array is already sorted or nearly sorted. Below is the pseudocode of randomized quick sort:

Algorithm 15 Quick Sort

```
1: procedure QUICKSORT(arr, l, r)
2:   pivot  $\leftarrow$  arr[random picked from [l, r]]
3:   left  $\leftarrow$  l
4:   right  $\leftarrow$  r
5:
6:   while left  $\leq$  right do
7:     while arr[left] < pivot do
8:       left  $\leftarrow$  left + 1
9:     end while
10:    while pivot < arr[right] do
11:      right  $\leftarrow$  right - 1
12:    end while
13:    if left  $\leq$  right then
14:      swap arr[left] and arr[right]
15:      left  $\leftarrow$  left + 1
16:      right  $\leftarrow$  right - 1
17:    end if
18:    if left < r then
19:      QuickSort(arr, left, r)
20:    end if
21:    if l < right then
22:      QuickSort(arr, l, right)
23:    end if
24:  end while
25: end procedure
```

The below figure is the visualization of the first **QuickSort** call, and the pivot is randomly chosen to be **20**. As described, it sequentially takes one element greater than or equal to pivot from the left, and smaller than or equal to pivot from the right, then swap these numbers, after that the points keep moving, eventually passed each other. Now we have two sub-arrays that need to be sorted by calling the same procedure.



1.8.3 Complexity evaluation

Space complexity analysis: The **Quick sort** implementation above is **in-place**. There for the space complexity is average $O(\log(n))$, which is the number of stack call, and $O(n)$ in the worst case.

Time complexity analysis: Mathematical analysis of **Quick sort** shows that, on average, the algorithm takes $O(n \log n)$ comparisons to sort n items. In the worst case, it makes $O(n^2)$ comparisons. [8]

Depends on different strategies of choosing the pivot, **Quick sort** could produce different overall performance. Choosing a such simple pivot as the first or the last element makes the algorithm runs in $O(n^2)$ complexity on sorted data.

Worst case analysis: This happens when the largest or smallest element is chosen to be the pivot. Suppose we have a sub-array of element from position **L** to position **R** and take the largest element **M** as pivot, let see what will happen to our sub-array if initially we have:

$$[a[\mathbf{L}], \dots, \mathbf{M}, \dots, a[\mathbf{R}]]$$

After the partition phase, the array becomes:

$$[a[\mathbf{L}], \dots, a[\mathbf{R}], \mathbf{M}]$$

Now **a[R]** is at position $R - 1$, hence only the pivot moving to the right. Because of this, the next sub-array being processed, the red-colored sub-array, has the length of $R - L$. The same story happens when the pivot is the minimum elements, now the pivot being moved to front, the next sub-array need to be processed also has the length of $R - L$.

With above observations, after each procedure call the number of elements need to be sorted is decreased by one. Therefore, for a **n** elements array, there are **n** procedure calls, at each call, if the number of element is now **m**, there are about **m - 1** comparisons. So in total we have about:

$$\sum_{i=1}^n (i - 1) = \frac{n \cdot (n - 1)}{2} \quad (\text{comparisons}) \in O(n^2)$$

Best case analysis: In the most balanced case, each time we perform a partition we divide the list into two nearly equal pieces. This means each recursive call processes a list of half the size. Consequently, we can make only $\log_2(n) \cdot n$ nested calls before we reach a list of size 1. This means that the depth of the call tree is $\log_2(n)$. But no two calls at the same level of the call tree process the same part of the original list; thus, each level of calls needs only $O(n)$ time all together (each call has some constant overhead, but since there are only $O(n)$ calls at each level, this is subsumed in the $O(n)$ factor). The result is that the algorithm uses only $O(n \cdot \log_2(n))$ time.[9]

Time complexity			Space complexity
Worst case	Average case	Best case	Worst case
$O(n^2)$	$O(n \log n)$	$\Omega(n \log n)$	$O(\log n)$

1.8.4 Variants

Improved space complexity: When it is carefully implemented using the following strategies. The in-place version of **quicksort** can have complexity of $O(\log(n))$, even in the worst case:

- In-place partitioning is used. This unstable partition requires $O(1)$ space.
- After partitioning, the partition with the fewest elements is (recursively) sorted first, requiring at most $O(\log(n))$ space.
- Then the other partition is sorted using tail recursion or iteration, which doesn't add to the call stack.

This idea, as discussed above, was described by R. Sedgewick, and keeps the stack depth bounded by $O(\log(n))$. [4] [5]

Multi-pivot quicksort: Instead of partitioning into two sub-arrays using a single pivot, multi-pivot quicksort (also **multiquick**sort) partitions its input into some s number of subarrays using $s - 1$ pivots. While the dual-pivot case ($s = 3$) was considered by *Sedgewick* and others already in the mid-1970s, the resulting algorithms were not faster in practice than the "classical" quicksort. [6]

Three-way radix quicksort: This algorithm is a combination of radix sort and quicksort. Pick an element from the array (the pivot) and consider the first character (key) of the string (multikey). Partition the remaining elements into three sets: those whose corresponding character is less than, equal to, and greater than the pivot's character. Recursively sort the "less than" and "greater than" partitions on the same character. [7]

1.9 Counting Sort

1.9.1 Idea

The idea of **Counting Sort** is designed to utilize information about the input range and the distribution of elements within the array to implement an efficient sorting algorithm with linear time complexity.

1.9.2 Description and Pseudo Code

Step by step description

- Step 1: Declare an frequency array (`countArray`) of size `max_element_of(inputArray) + 1` and initialize its elements with zeros.
- Step 2: For each element **X** in `inputArray` we increase the value of `countArray` at position **X** by one.
- Step 3: Calculate the prefix sum of the array `countArray`. By doing this, `countArray[X]` will be the correct position of the value **X** in the sorted array.
- Step 4: Declare `outputArray` of size **N** to store the result.
- Step 5: Traverse array `inputArray` from right to left and place its elements into its correct position in the `outputArray` by using `countArray` elements as index.

Algorithm 16 Counting Sort

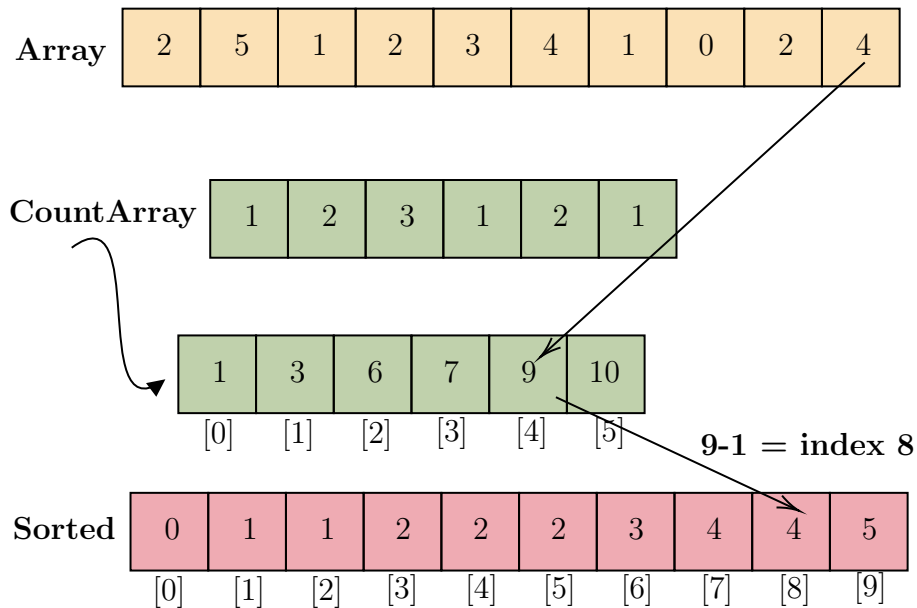
1: $count[0 \dots k] \leftarrow \text{array of zeros}$	▷ Initialize count array
2: for $i \leftarrow 0$ to $\text{length}(\text{array}) - 1$ do	
3: $count[\text{array}[i]] \leftarrow count[\text{array}[i]] + 1$	▷ Count frequencies
4: end for	
5: for $i \leftarrow 1$ to k do	
6: $count[i] \leftarrow count[i] + count[i - 1]$	▷ Compute positions
7: end for	

Algorithm 17 Counting Sort (continue)

```

8: output[length(array)]  $\leftarrow$  array to store sorted elements
9: for  $i \leftarrow$  length(array) - 1 downto 0 do
10:   output[count[array[ $i$ ]] - 1]  $\leftarrow$  array[ $i$ ]
11:   count[array[ $i$ ]]  $\leftarrow$  count[array[ $i$ ]] - 1 ▷ Build output array
12: end for
13: for  $i \leftarrow$  0 to length(array) - 1 do
14:   array[ $i$ ]  $\leftarrow$  output[ $i$ ] ▷ Copy sorted elements to original array
15: end for

```

**1.9.3 Complexity evaluation**

Counting sort operates under specific assumptions about the input data. For instance, it presumes that values will fall within a certain range, such as 0 to 10 or 10 to 99. Additionally, counting sort assumes that the input data consists of positive integers.

The algorithm is efficient when the range of input data (denoted as k) is not significantly larger than the number of elements to be sorted (denoted as n). Consider a scenario where the input sequence ranges from 1 to 10,000, with data points such as 10, 5, 10,000, and 5,000. In this case, k is 10,000 and n is 4. The counting sort algorithm will create a count array of size 10,001 to cover all possible values from 0 to 10,000, leading to a space complexity of $O(k)$. The time complexity in this scenario will be $O(n + k)$, or $O(4 + 10,000)$.

While **Counting Sort** can effectively sort the data in this example, its efficiency diminishes as the range of input values becomes significantly larger than the number of elements. Therefore, in such cases, alternative sorting algorithms like radix sort or comparison-based methods might be more suitable.

Counting Sort is often used as a sub-routine in other sorting algorithms, such as radix sort. Additionally, the algorithm can be adapted to handle negative inputs, extending its applicability to a broader range of datasets.

Time complexity			Space complexity
Worst case	Average case	Best case	Worst case
$O(n + k)$	$O(n + k)$	$\Omega(n + k)$	$O(k)$

1.9.4 Variants

Some variants can be adapted by adjusting the range of keys or modifying the counting process itself. These adaptations allow Counting sort to handle different types of input data or optimize its performance for specific use cases. Some noteworthy variants of **Counting Sort**:

Adjusting Key Range: Counting Sort can be modified to accommodate a wider or narrower range of key values by resizing the count array accordingly. This adjustment enables the sorting of data where the key values fall outside the typical 0 to k range assumed by standard Counting sort implementations.

Changing Counting Process: Variants of Counting Sort may alter how counts are accumulated or how the output array is constructed based on the counts. This flexibility can improve efficiency or adapt the algorithm to different characteristics of the input data, such as handling duplicate keys more effectively.

1.10 Radix Sort

1.10.1 Idea

The idea of **radix sort** is to repeatedly consider the digits from **right to left** and distribute each elements into one of ten "buckets" base on the value of their digit, (0 through 9), but keep their relative ordering in each "bucket".

1.10.2 Description and Pseudo Code

Step by step description

Step 1: Determine the maximum number of digits in the array.

Step 2: Starting from the least significant digit (rightmost), sort the numbers based on that digit using any sorting algorithm (we will using counting sort because the number of different digits is only 10).

Step 3: Repeat step 2 for each subsequent digit position, moving towards the most significant digit. If the number does not have enough number of digits, we pad it with zeros.

Step 4: After iterate all the digits, the array will be sorted.

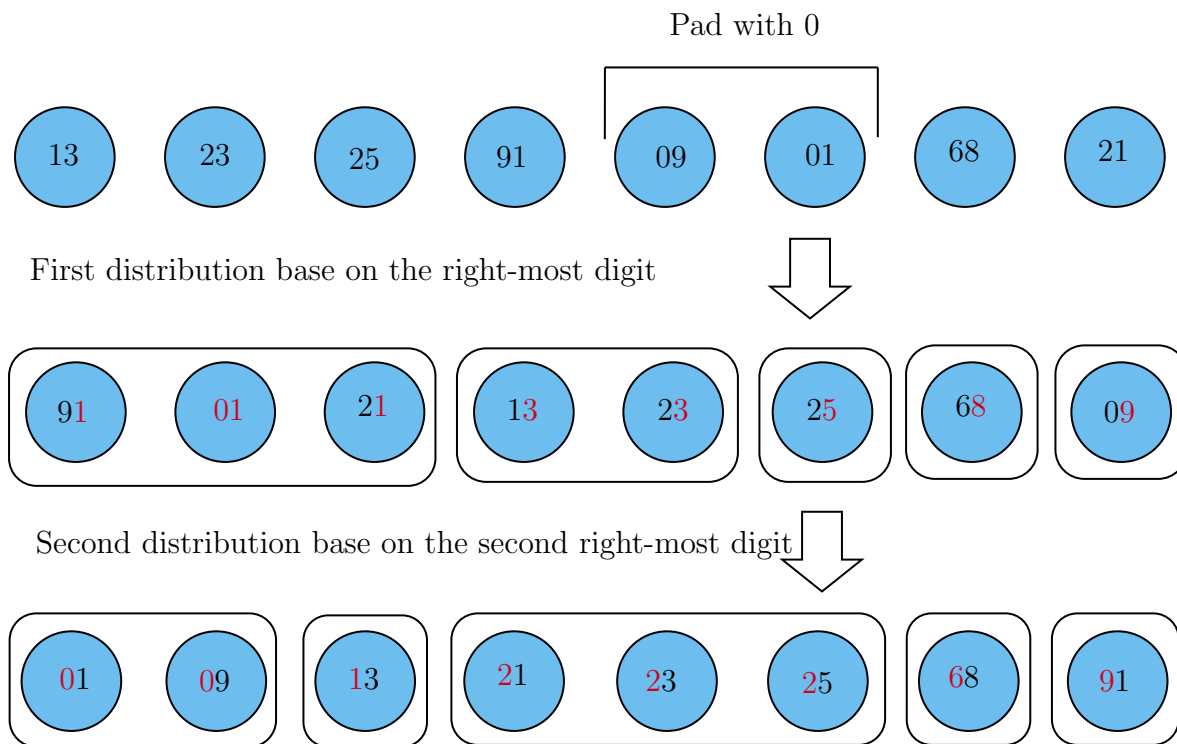
Algorithm 18 Radix Sort

```

1: for  $i \leftarrow 1$  to  $\lfloor \log_{10}(\max\_element(arr)) \rfloor$  do
2:   Perform counting sort with key is the  $i$ -th digit
3: end for

```

The below figure show how the **radix sort** runs on the array consists of number with 1 and 2 digits long. Although the algorithm does not actually padding numbers with zeros, for convenient purposes, we do pad the 1-digit numbers with leading zeros.



1.10.3 Complexity evaluation

Time complexity: The complexity of **Radix sort** is $O(d \cdot n)$, where d is the length of the longest number or string. The complexity can be achieved by implementing counting sort to distribute the elements of the array. We know that there are at most 10 different digits to consider, therefor the complexity comes from the complexity of counting sort times the number of iterations.

Space complexity: With the above implementation using counting sort, we will use $O(n)$ extra space for the temporary array. Hence, the space complexity is $O(n)$. However, there's a version of **radix sort** that can be implemented with $O(1)$ space complexity, which will be noted later.

Time complexity			Space complexity
Worst case	Average case	Best case	Worst case
$O(d \cdot n)$	$O(d \cdot n)$	$\Omega(d \cdot n)$	$O(d + n)$

1.10.4 Variants

Here are some variants that we find interesting: The first one has the idea of sorts the array in-place and the other gives us the idea of a very useful tree-like structure.

In-place MSD radix sort implementations: This version considers the binary representation of the numbers when performs sorting. Because now every digits are either 0 or 1, the distribution process can be performed in-place by swapping every element with the considering digit is 0 from the back with element whose digit is 1 from the front. This procedure is similar to the process of partitioning elements of **quick-sort** algorithm. After we have two parts, then we can recursively process each part with the next significant digit. [10]

Tree-based radix sort: Radix sorting can also be accomplished by building a tree (or radix tree[12]) from the input set, and doing a pre-order traversal. This is similar to the relationship between heapsort and the heap data structure. This tree structure helps us efficiently implement sets or dictionary for words searching purposes. [11]

1.11 Flash Sort

1.11.1 Idea

The **Flash Sort** algorithm was published by Karl-Dietrich Neubert in 1998. It's an in-place sorting algorithm that promises linear time complexity for evenly distributed data. It's an efficient way to implement the bucket sort[13]. It creates buckets (the author calls them classes) and rearranges all elements according to buckets. Lastly, it sorts each bucket. [14]

1.11.2 Description and Pseudo Code

Step by step description

- Step 1: **Initialization:** Find the minimum (minVal) and maximum (maxVal) values in the array 'arr'.
- Step 2: **Bucket Parameters:** Define 'numBuckets' and calculate 'c' as the ratio of bucket count to range of values (maxVal - minVal).
- Step 3: **Counting and Mapping:** Count elements into buckets using 'c' to determine the bucket index for each element.
- Step 4: **Prefix Sum Calculation:** Compute prefix sums to determine where each bucket's elements start in the sorted array.
- Step 5: **Sorting in Buckets:** Place elements into 'sortedArr' based on their bucket index, maintaining order within each bucket using an insertion sort.
- Step 6: **Final Sorting:** Copy elements from 'sortedArr' back into 'arr', ensuring the array is sorted.
- Step 7: **Memory Management:** Free any dynamically allocated memory used during the sorting process.

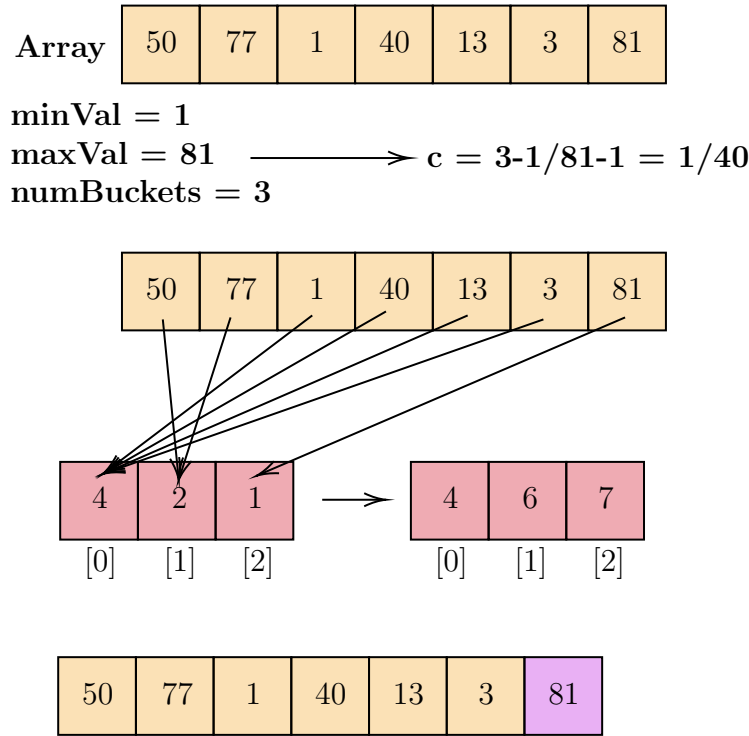
Algorithm 19 Flash Sort

```

1:  $minVal \leftarrow arr[0]$  and  $maxVal \leftarrow arr[0]$ 
2: for  $i \leftarrow 1$  to  $n - 1$  do
3:   if  $arr[i] < minVal$  then
4:      $minVal \leftarrow arr[i]$ 
5:   else if  $arr[i] > maxVal$  then
6:      $maxVal \leftarrow arr[i]$ 
7:   end if
8: end for
9:  $numBuckets \leftarrow 100000$ 
10:  $c \leftarrow \frac{numBuckets-1}{maxVal-minVal}$ 
11:  $bucketCounts \leftarrow$  array of size  $numBuckets$ , initialized to 0
12: for  $i \leftarrow 0$  to  $n - 1$  do
13:    $bucketIndex \leftarrow \lfloor c \cdot (arr[i] - minVal) \rfloor$ 
14:    $bucketCounts[bucketIndex] \leftarrow bucketCounts[bucketIndex] + 1$ 
15: end for
16:  $prefixSum \leftarrow$  array of size  $numBuckets$ , initialized to 0
17: for  $i \leftarrow 1$  to  $numBuckets - 1$  do
18:    $prefixSum[i] \leftarrow prefixSum[i - 1] + bucketCounts[i - 1]$ 
19: end for
20:  $sortedArr \leftarrow$  array of size  $n$ 
21: for  $i \leftarrow 0$  to  $n - 1$  do
22:    $bucketIndex \leftarrow \lfloor c \cdot (arr[i] - minVal) \rfloor$ 
23:    $sortedArr[prefixSum[bucketIndex]] \leftarrow arr[i]$ 
24:    $prefixSum[bucketIndex] \leftarrow prefixSum[bucketIndex] + 1$ 
25: end for
26: for  $i \leftarrow 0$  to  $numBuckets - 1$  do
27:    $start \leftarrow$  if  $i == 0$  then 0 else  $prefixSum[i - 1]$ 
28:    $end \leftarrow prefixSum[i]$ 
29:   for  $j \leftarrow start + 1$  to  $end - 1$  do
30:      $key \leftarrow sortedArr[j]$ 
31:      $k \leftarrow j - 1$ 
32:     while  $k \geq start$  and  $sortedArr[k] > key$  do
33:        $sortedArr[k + 1] \leftarrow sortedArr[k]$ 
34:        $k \leftarrow k + 1$ 
35:     end while
36:      $sortedArr[k + 1] \leftarrow key$ 
37:   end for
38: end for
39: for  $i \leftarrow 0$  to  $n - 1$  do
40:    $arr[i] \leftarrow sortedArr[i]$ 
41: end for
42: Free allocated memory

```

Below is the visualization of Flash Sort algorithm:



1.11.3 Complexity evaluation

The partitioning phase operates in $\mathcal{O}(n)$ time as each element is processed once.

During sorting, each of the m partitions, on average containing $\frac{n}{m}$ elements, is sorted using **Insertion Sort**, which takes $\mathcal{O}\left(\frac{n^2}{m^2}\right)$ time. With m partitions, the total sorting time complexity is $\mathcal{O}(n \cdot m^2)$.

Empirical evidence suggests setting $m = 0.43n$ for optimal performance, reducing the sorting phase to $\mathcal{O}(n)$. In rare worst-case scenarios with uneven data distribution, the complexity may reach $\mathcal{O}(n^2)$.

Memory complexity remains $\mathcal{O}(m) = \mathcal{O}(n)$. Choosing $m = n$ ensures uniform partitioning, optimizing the algorithm under specific dataset constraints.

Time complexity			Space complexity
Worst case	Average case	Best case	Worst case
$\mathcal{O}(n^2)$	$\mathcal{O}(n + m)$	$\Omega(n + m)$	$\mathcal{O}(n + m)$


1.11.4 Variants

Rather than using a fixed number of buckets, an enhancement to Flash Sort involves dynamically adjusting the bucket count based on the data distribution. This adaptive approach aims to optimize sorting performance by accommodating varying data characteristics, potentially yielding improved results in practice.

2 Experimental results and comments

2.1 Experimental computer's specification

Below is the PC configuration that all the experiments are run on. Although the device is plugged on, it still a laptop, it will not have the performance of the same-spec desktop.



Operating System: Windows 11 Home 64-bit (10.0, Build 22631)
Language: English (Regional Setting: English)
System Manufacturer: HP
System Model: HP Spectre x360 2-in-1 Laptop 16-f2xxx
BIOS: F.14
Processor: 13th Gen Intel(R) Core(TM) i7-13700H (20 CPUs), ~2.4GHz
Memory: 16384MB RAM
Page file: 16286MB used, 5449MB available
DirectX Version: DirectX 12

Figure 1: Enter Caption

2.1.1 How did we run experiments

Some sorting algorithms performs poorly on large data set, some may take up to hours to complete measuring time and number of comparison.

So we make a copy of the source code and modified it for the output is put into **csv** formatted files and used **.bat** script to automatically run the commands.

To plot the results of the experiments onto line graphs and bar charts, we used **Excel** and **Mathcha.io** draw the comparison tables.

2.2 Experimental results

2.2.1 Sorted data

Data order: Sorted data						
Data size	10,000		30,000		50,000	
Resulting Static	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection Sort	71 ms	100019998	304 ms	900059998	844 ms	2500099998
Insertion Sort	0 ms	29998	0 ms	89998	1 ms	149998
Bubble Sort	71 ms	100009999	338 ms	900029999	916 ms	2500049999
Shaker Sort	0 ms	20002	0 ms	60002	0 ms	100002
Shell Sort	0 ms	360042	1 ms	1170050	1 ms	2100049
Heap Sort	0 ms	699781	2 ms	2320501	6 ms	4066521
Merge Sort	0 ms	475242	3 ms	1559914	7 ms	2722826
Quick Sort	0 ms	215498	3 ms	714452	3 ms	1231533
Counting Sort	0 ms	70005	0 ms	210005	0 ms	350005
Radix Sort	0 ms	140056	0 ms	510070	2 ms	850070
Flash Sort	0 ms	97865	1 ms	293615	1 ms	489365
Data size	100,000		300,000		500,000	
Resulting Static	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection Sort	3394 ms	10000199998	31814 ms	90000599998	87697 ms	250000999998
Insertion Sort	0 ms	299998	1 ms	899998	0 ms	1499998
Bubble Sort	3703 ms	10000099999	33688 ms	90000299999	93312 ms	250000499999
Shaker Sort	0 ms	200002	0 ms	600002	0 ms	1000002
Shell Sort	1 ms	4500051	7 ms	15300061	11 ms	25500058
Heap Sort	13 ms	8654271	42 ms	28255801	75 ms	48778501
Merge Sort	9 ms	5745658	32 ms	18645946	54 ms	32017850
Quick Sort	3 ms	2554562	13 ms	8271230	22 ms	14402288
Counting Sort	0 ms	700005	2 ms	2100005	3 ms	3500005
Radix Sort	2 ms	1700070	13 ms	6000084	23 ms	10000084
Flash Sort	1 ms	978740	5 ms	2936240	10 ms	4893740

2.2.2 Nearly sorted data

Data order: Nearly sorted data						
Data size	10,000		30,000		50,000	
Resulting Static	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection Sort	71 ms	100019998	317 ms	900059998	871 ms	2500099998
Insertion Sort	0 ms	137734	0 ms	505362	1 ms	689550
Bubble Sort	71 ms	100009999	341 ms	900029999	929 ms	2500049999
Shaker Sort	0 ms	162103	0 ms	541703	0 ms	584985
Shell Sort	0 ms	401798	1 ms	1318062	1 ms	2287579
Heap Sort	0 ms	6699321	7 ms	2320201	10 ms	4065151
Merge Sort	0 ms	507190	3 ms	1666302	5 ms	2851366
Quick Sort	0 ms	213530	1 ms	721766	2 ms	1255331
Counting Sort	0 ms	70005	0 ms	210005	0 ms	350005
Radix Sort	0 ms	140056	1 ms	510070	1 ms	850070
Flash Sort	0 ms	104540	1 ms	293589	1 ms	489341
Data size	100,000		300,000		500,000	
Resulting Static	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection Sort	3728 ms	10000199998	31518 ms	90000599998	88465 ms	250000999998
Insertion Sort	1 ms	919026	0 ms	1162170	1 ms	1777910
Bubble Sort	3835 ms	10000099999	33784 ms	90000299999	147901 ms	250000499999
Shaker Sort	1 ms	785681	1 ms	1123607	0 ms	1522887
Shell Sort	2 ms	4688600	6 ms	15436189	13 ms	25645341
Heap Sort	12 ms	8653941	43 ms	28255651	76 ms	48778481
Merge Sort	10 ms	5856144	32 ms	18757226	53 ms	32108772
Quick Sort	4 ms	2583720	14 ms	9586551	22 ms	14632380
Counting Sort	0 ms	700005	2 ms	2100005	3 ms	3500005
Radix Sort	2 ms	1700070	12 ms	6000084	25 ms	10000084
Flash Sort	1 ms	978717	6 ms	2936219	11 ms	4893719

2.2.3 Reversed sorted data

Data order: Reversed sorted data						
Data size	10,000		30,000		50,000	
Resulting Static	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection Sort	71 ms	100019998	322 ms	900059998	898 ms	2500099998
Insertion Sort	95 ms	100009999	433 ms	900029999	1200 ms	2500049999
Bubble Sort	312 ms	100009999	1383 ms	900029999	3858 ms	2500049999
Shaker Sort	311 ms	100005001	1385 ms	900015001	3831 ms	2500025001
Shell Sort	0 ms	475175	0 ms	1554051	1 ms	2844628
Heap Sort	0 ms	623481	3 ms	2116061	9 ms	3694461
Merge Sort	0 ms	476441	6 ms	1573465	4 ms	2733945
Quick Sort	7 ms	226236	1 ms	730095	3 ms	1331123
Counting Sort	0 ms	70005	0 ms	210005	0 ms	350005
Radix Sort	0 ms	140056	2 ms	510070	1 ms	850070
Flash Sort	7 ms	6333935	25 ms	56501810	74 ms	156669685
Data size	100,000		300,000		500,000	
Resulting Static	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection Sort	3605 ms	10000199998	32342 ms	90000599998	90395 ms	250000999998
Insertion Sort	4767 ms	10000099999	43411 ms	90000299999	119878 ms	250000499999
Bubble Sort	15307 ms	10000099999	137930 ms	90000299999	448434 ms	250000499999
Shaker Sort	15470 ms	10000050001	138099 ms	90000150001	384496 ms	250000250001
Shell Sort	4 ms	6089190	10 ms	20001852	18 ms	33857581
Heap Sort	12 ms	7887171	39 ms	26081231	75 ms	45342251
Merge Sort	11 ms	5767897	32 ms	18708313	51 ms	32336409
Quick Sort	4 ms	2776932	15 ms	8599240	25 ms	15368051
Counting Sort	0 ms	700005	2 ms	2100005	2 ms	3500005
Radix Sort	2 ms	1700070	13 ms	6000084	22 ms	10000084
Flash Sort	297 ms	625739384	2708 ms	5627218134	7556 ms	15628696884

2.2.4 Randomized data

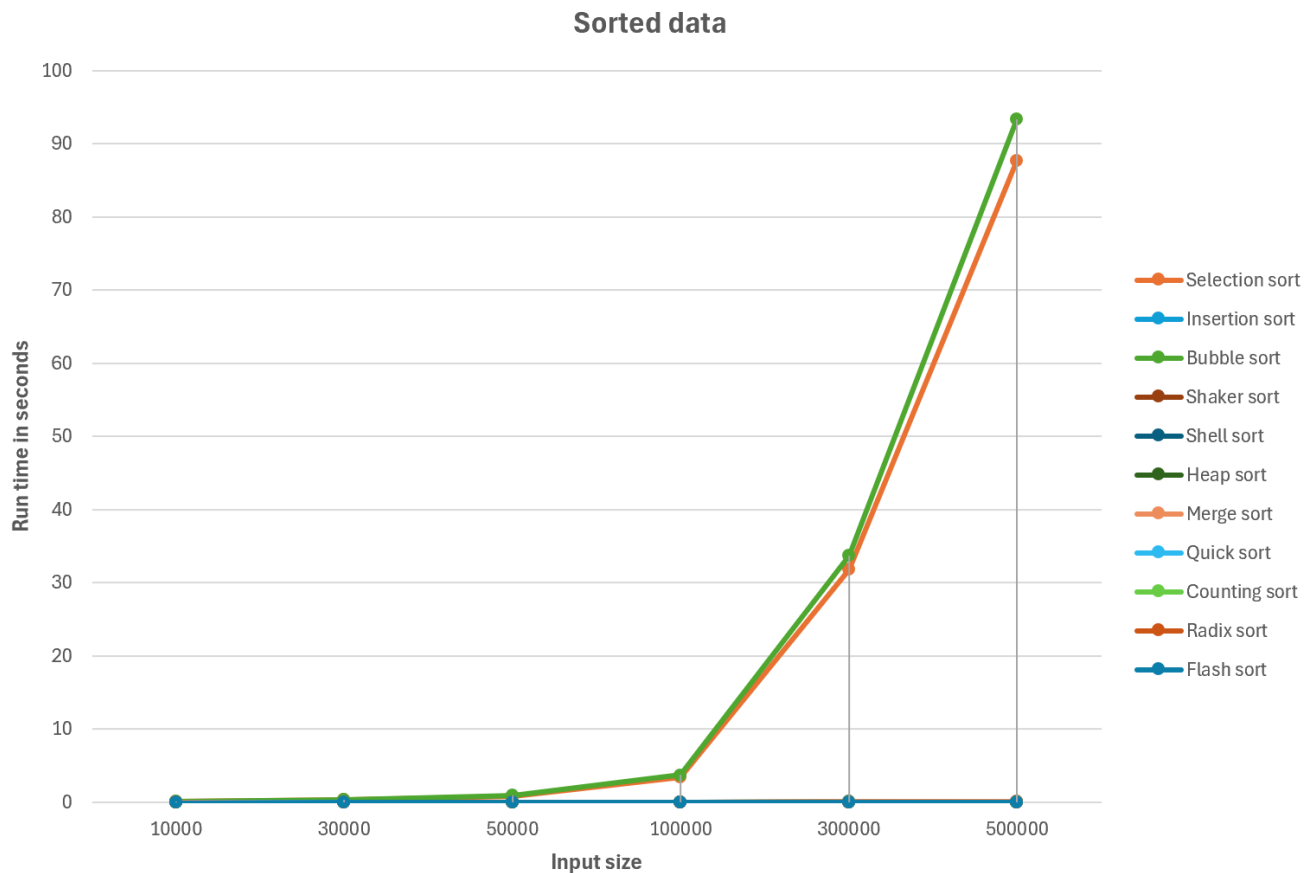
Data order: Randomized sorted data						
Data size	10,000		30,000		50,000	
Resulting Static	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection Sort	88 ms	100019998	429 ms	900059998	879 ms	2500099998
Insertion Sort	64 ms	49386607	269 ms	453219995	570 ms	1250914818
Bubble Sort	352 ms	100009999	1789 ms	900029999	4993 ms	2500049999
Shaker Sort	288 ms	66315631	1257 ms	603945417	3436 ms	1669773768
Shell Sort	8 ms	637155	5 ms	2367866	8 ms	4632397
Heap Sort	0 ms	661361	5 ms	2219041	7 ms	3886906
Merge Sort	0 ms	583635	4 ms	1937394	7 ms	3382776
Quick Sort	0 ms	284500	3 ms	983366	5 ms	1622379
Counting Sort	0 ms	70001	0 ms	210005	1 ms	315539
Radix Sort	0 ms	140056	2 ms	510070	0 ms	850070
Flash Sort	15 ms	13895808	88 ms	127062782	74 ms	191295365
Data size	100,000		300,000		500,000	
Resulting Static	Running time	Comparison	Running time	Comparison	Running time	Comparison
Selection Sort	3436 ms	10000199998	31360 ms	90000599998	87859 ms	250000999998
Insertion Sort	2332 ms	4987717317	21544 ms	45011923262	59446 ms	124983546284
Bubble Sort	20675 ms	10000099999	189036 ms	90000299999	526362 ms	250000499999
Shaker Sort	14146 ms	6659252250	128129 ms	60088704043	691902 ms	166755045957
Shell Sort	16 ms	10154536	51 ms	34397821	85 ms	63750737
Heap Sort	19 ms	8273811	58 ms	27183286	105 ms	47120106
Merge Sort	15 ms	7166256	52 ms	23382575	87 ms	40382523
Quick Sort	10 ms	3355162	31 ms	10788376	51 ms	18841755
Counting Sort	1 ms	565541	2 ms	1565541	2 ms	2565541
Radix Sort	2 ms	1700070	11 ms	5100070	17 ms	8500070
Flash Sort	89 ms	180596187	503 ms	1029772531	1690 ms	3466651635

2.3 Line graphs

Because of the PC configuration, we see almost no differences in running time of $O(n \cdot \log n)$ algorithms versus the $O(n)$ algorithms were plotted on the line graphs.

Although it may seem the differences is minor, in fact, with a powerful CPU as a core i7 13700H with 16 GB of ram, such gap can be considerably huge on lower specification devices.

2.3.1 Sorted data

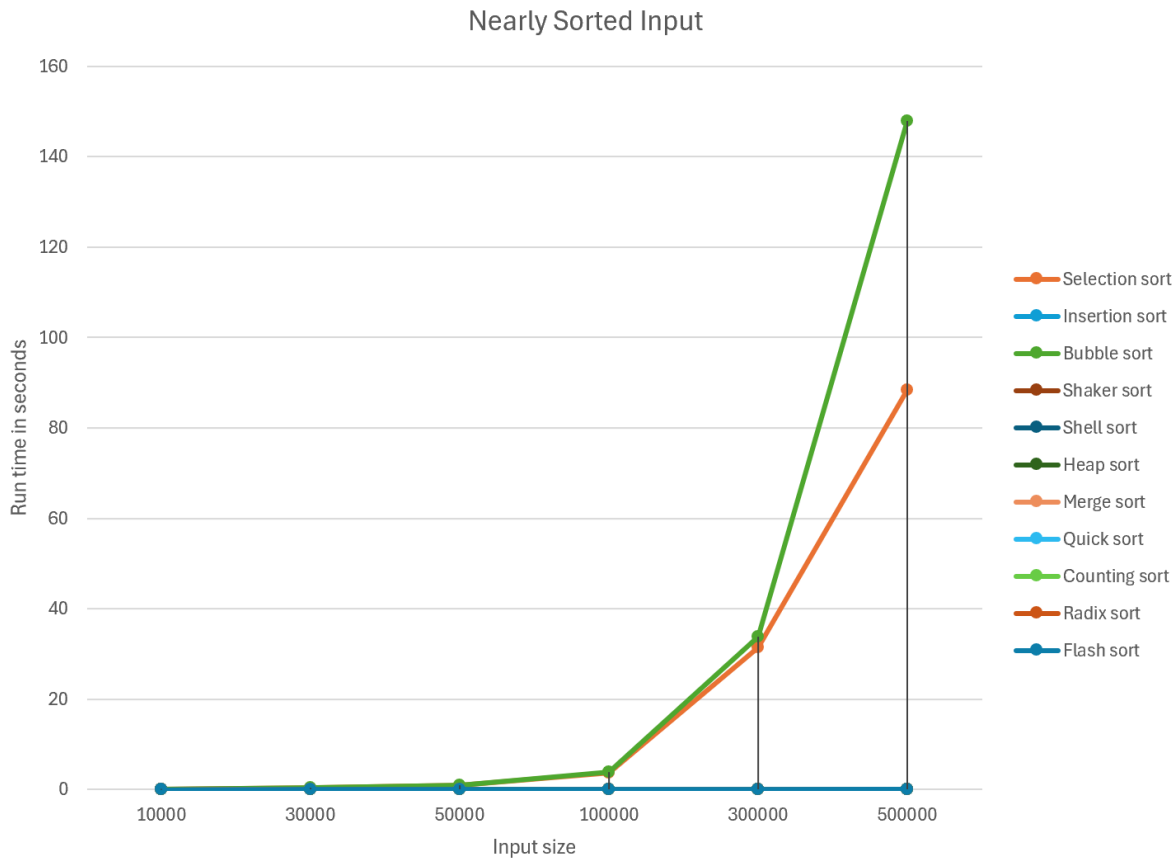


Comment

The two algorithms perform worst are bubble sort and selection sort, because both of them run in $O(n^2)$ complexity regardless of the data order. In sorted data, insertion sort runs in $O(n)$, so we see the running time almost a straight line.

As noted above, every other algorithms with time complexity less than or equal to $O(n \log n)$ will barely different from each other, therefor we see a straight line.

2.3.2 Nearly sorted data



Comment

The two algorithms that perform the worst are bubble sort and selection sort because both of them run with $O(n^2)$ complexity regardless of the data order. For nearly sorted data, we see that insertion sort still performs very well, specifically $O(n)$. It is worth mentioned that, according to the provided generator, the strategy involves randomly swapping two elements ten times, so the number of elements out of order is at most 20. And below is the proof for the $O(n)$ time complexity.

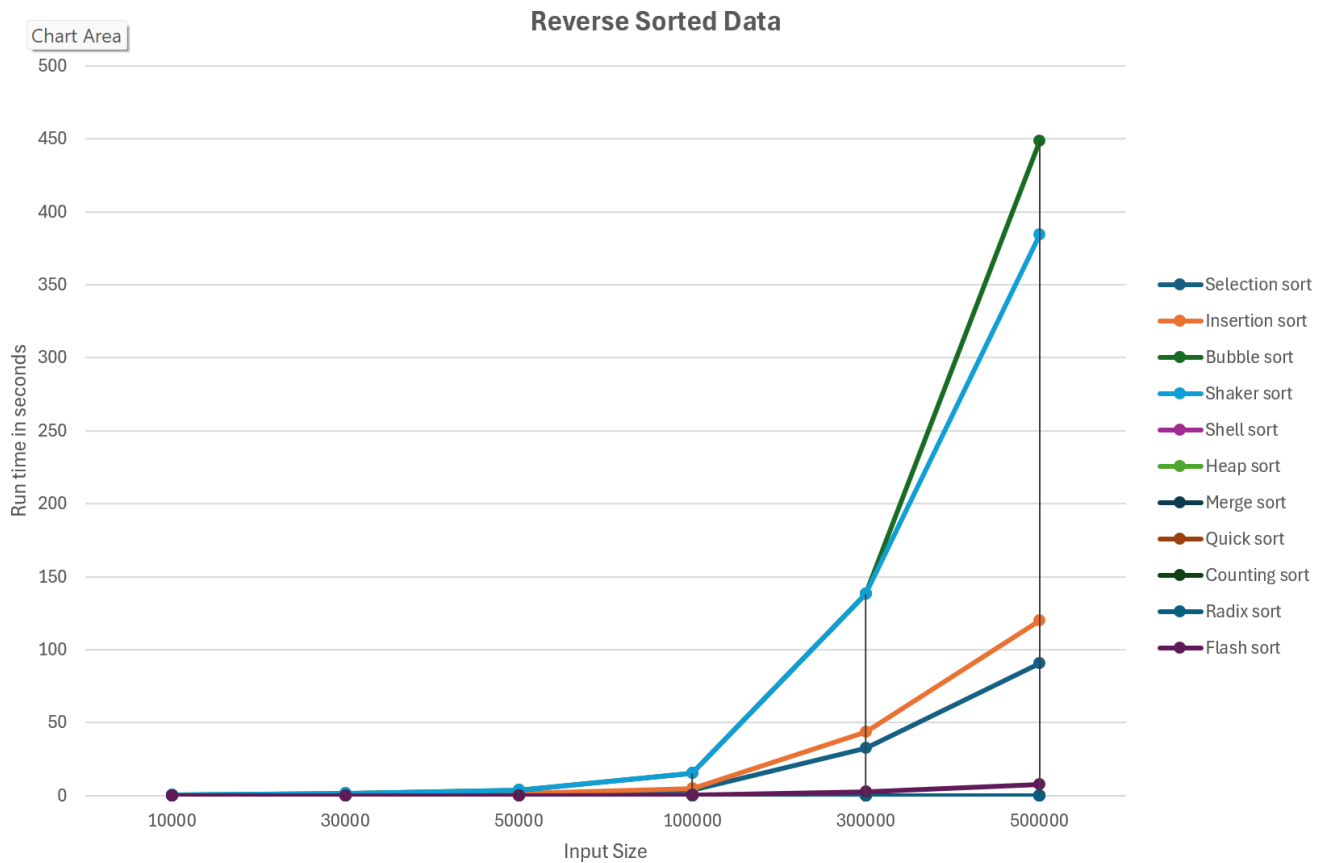
First observation: Suppose each out-of-order element needs to move n steps to the left to reach its correct position. The number of swaps is n .

Second observation: If an element is out of order and needs to move to the right n times, it means that the n elements after it also need to be moved to the left by one position n times. In total, the number of swaps is n .

With the two observations above, we have proven that moving each element to its correct position takes at most n swaps. Therefore, we have an upper $O(n)$ time complexity, (at most $20 \times n$ swaps), which is the state we need to prove.

The remaining algorithms still run in $O(n \cdot \log n)$ and $O(n)$, so the graph remains a straight line.

2.3.3 Reverse sorted data

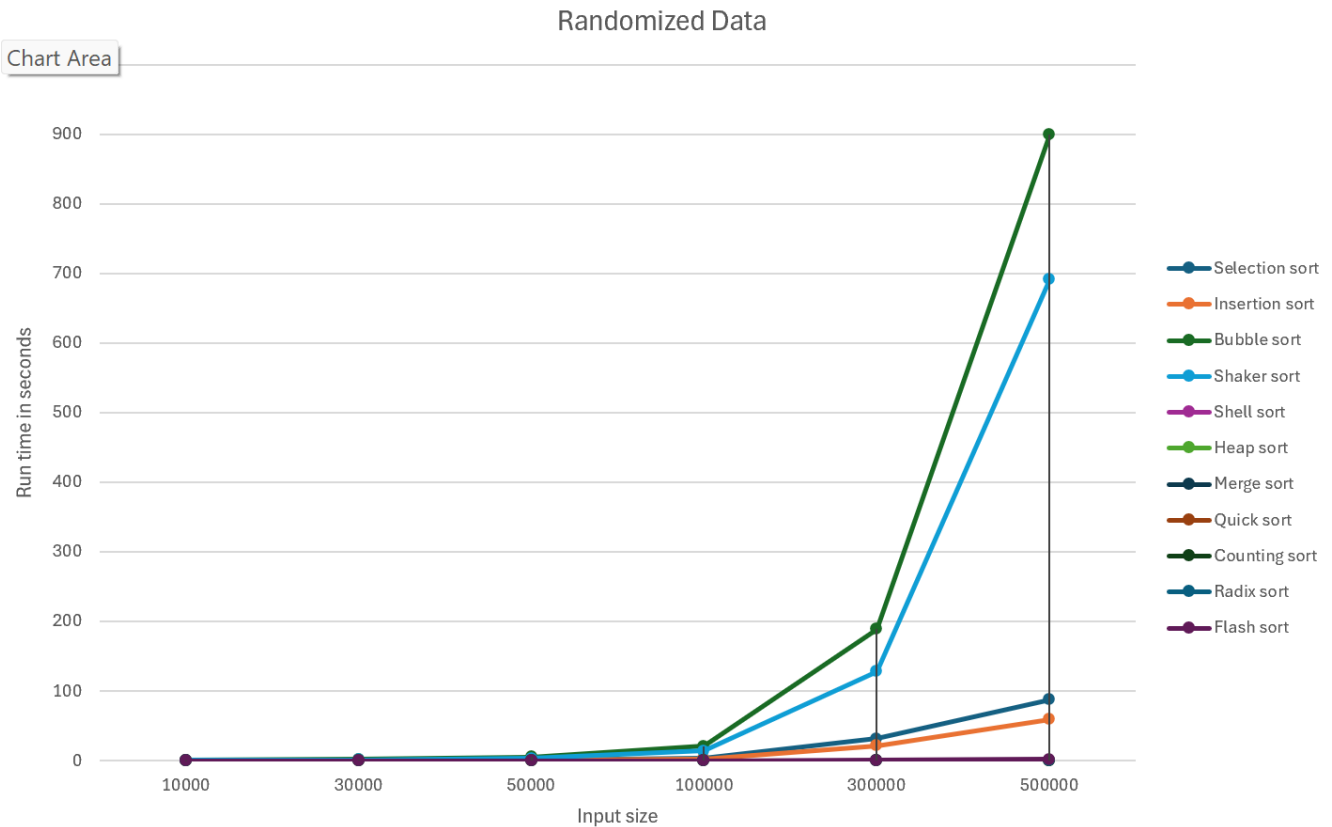


Comment

The worst algorithm is bubble sort, the second worst is shaker sort. Other $O(n^2)$ algorithms also perform poorly. With Insertion Sort and Selection Sort are now performing at their worst case time complexity.

The remaining algorithms still run in $O(n \cdot \log n)$ and $O(n)$, so the graph remains a straight line.

2.3.4 Randomized data



Comment

The worst algorithm is bubble sort, the second worst is shaker sort. Other $O(n^2)$ algorithms also perform poorly.

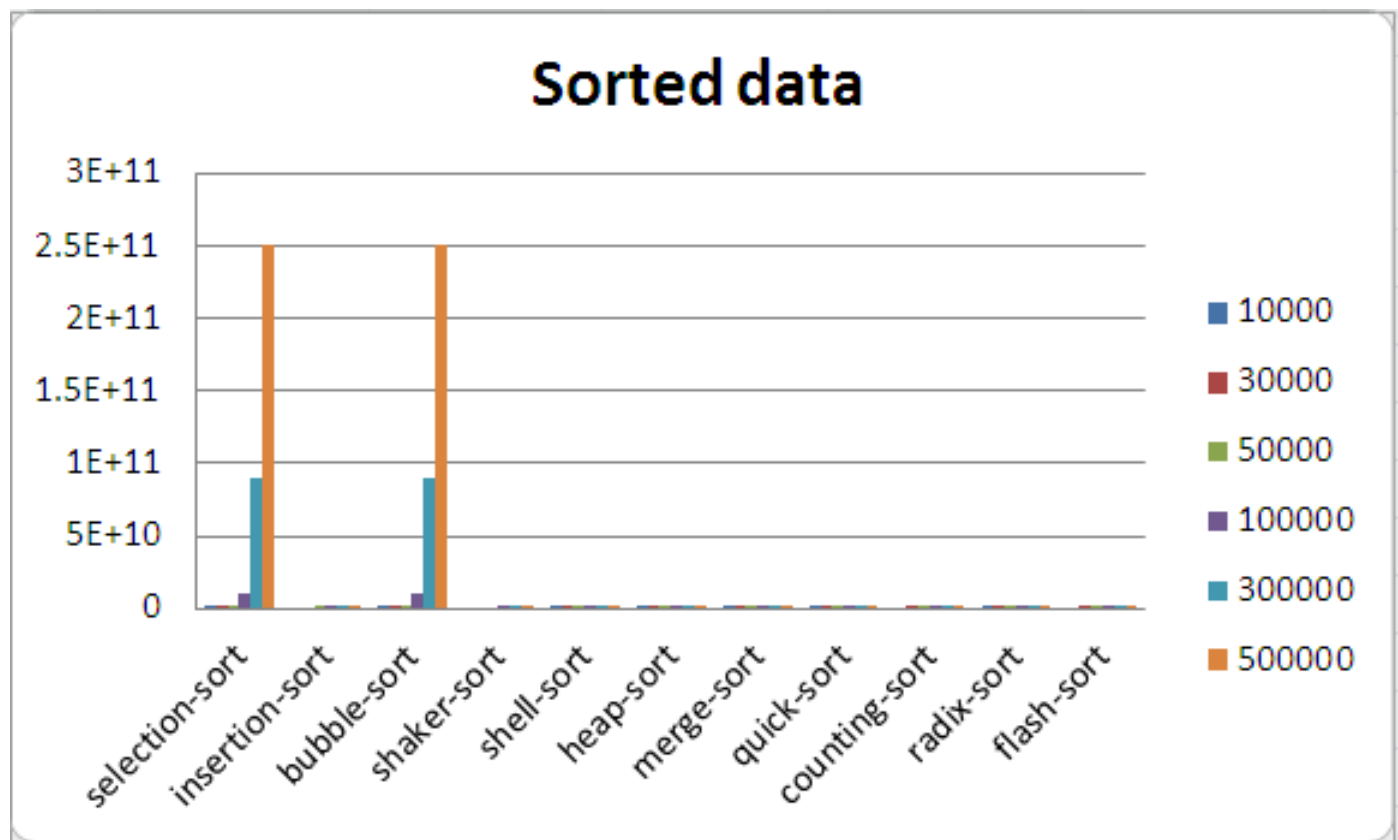
The remaining algorithms still run in $O(n \cdot \log n)$ and $O(n)$, so the graph remains a straight line.

2.4 Bar charts

Some sorting algorithms are comparison-based, so knowing the number of comparisons each sorting algorithm makes is a crucial part of this research. To effectively visualize the number of comparisons made, it is best to use bar charts.

It is worth mentioning that we do not include **non-comparative** algorithms such as **radix sort** and **counting sort** in comparisons of worst or best cases.

2.4.1 Sorted data

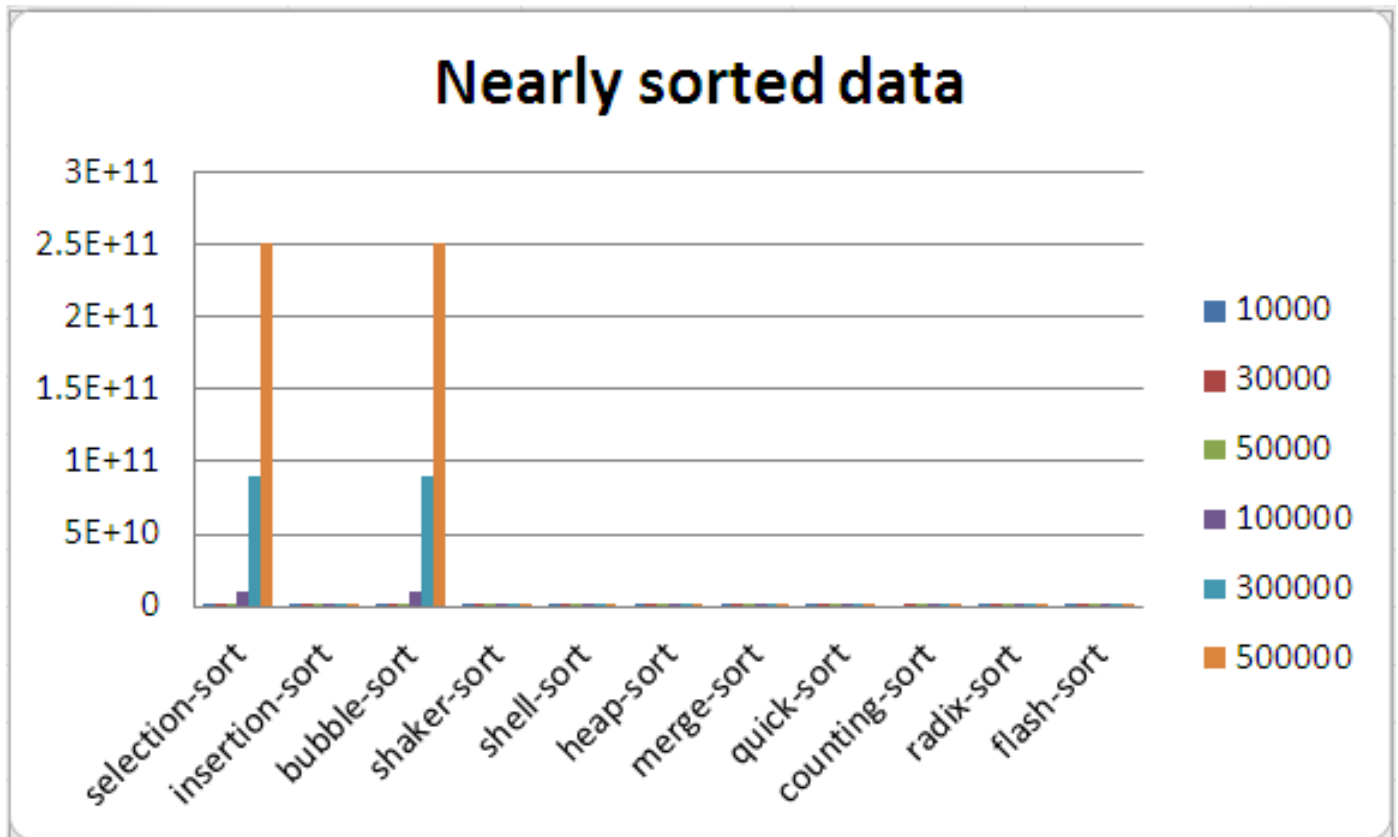


Comment

Bubble Sort and Selection Sort make the most number of comparisons. These algorithms are comparison-based sorting techniques. When the data is already sorted, both Bubble Sort and Selection Sort still perform a significant number of comparisons to verify the order of elements, which makes them less efficient for sorted data.

Quick Sort makes the least number of comparisons. In the case of already sorted data, it can achieve fewer comparisons compared to other sorting algorithms. It efficiently separates the data and eliminates redundant comparisons, leading to improved performance on sorted data.

2.4.2 Nearly sorted data

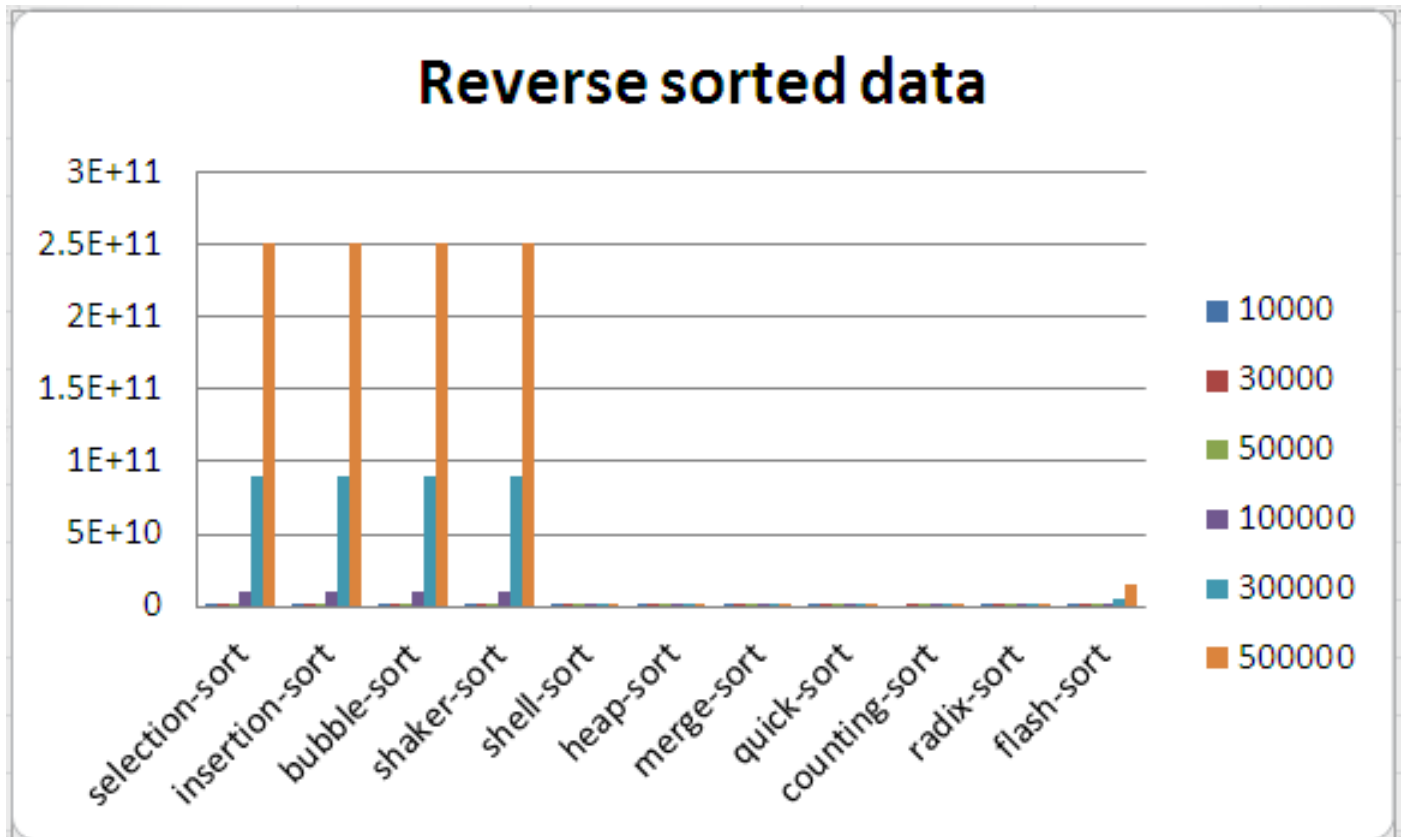


Comment

Bubble Sort and Selection Sort make the most comparisons because they are comparison-based sorting techniques. Even when the data is already sorted, they perform numerous comparisons to verify the order of elements, making them less efficient for nearly sorted data.

Quick Sort makes the fewest comparisons. In the case of already sorted data, it efficiently separates the data and avoids redundant comparisons, leading to an improved performance on nearly sorted data.

2.4.3 Reverse sorted data

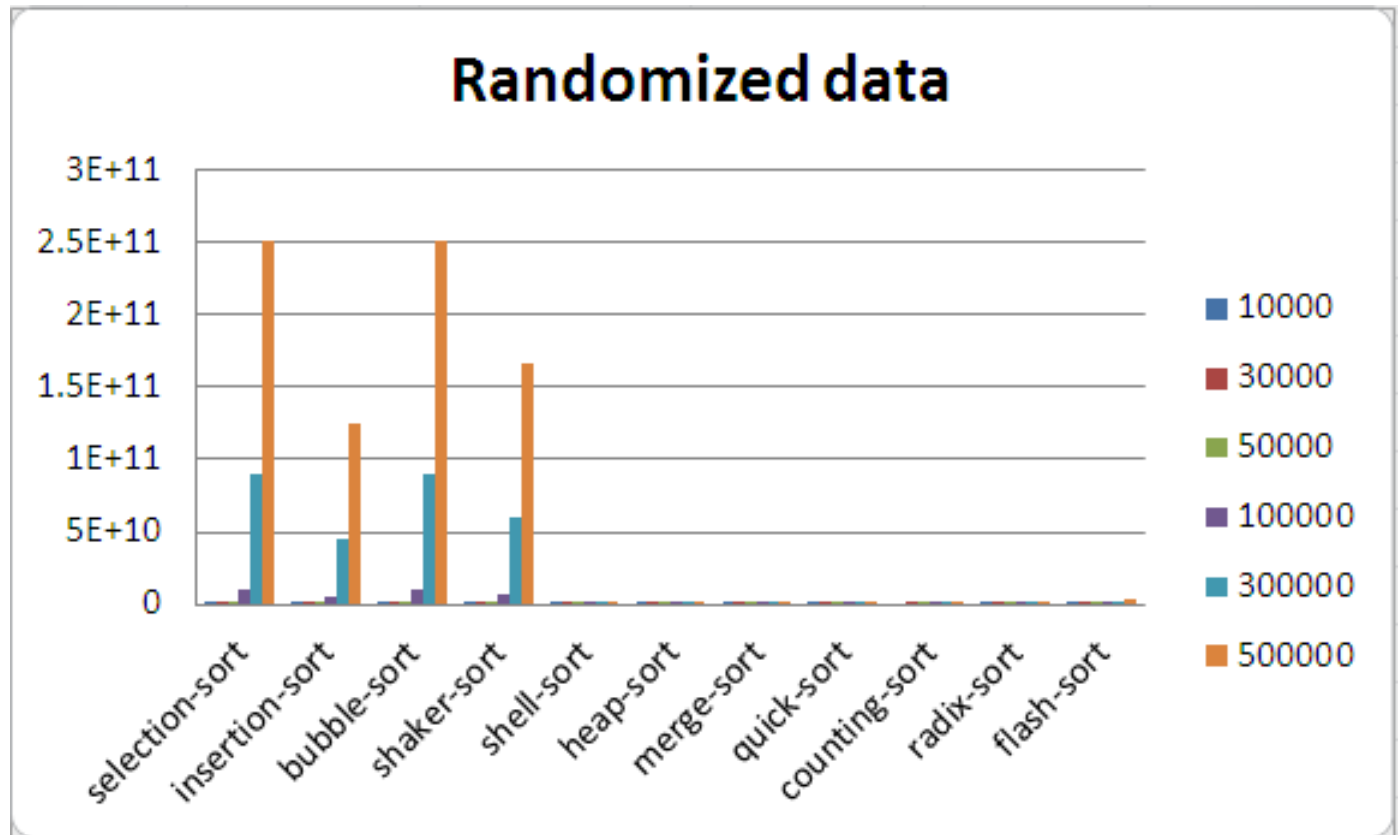


Comment

Bubble Sort, Selection Sort, Insertion Sort, Shaker Sort, and Shell Sort make the most comparisons. These sorting algorithms perform a large number of comparisons when dealing with reverse sorted data. They need to repeatedly swap elements to correctly order the reversed input, resulting in higher comparison counts.

Quick Sort makes fewest comparisons. Despite being efficient in general cases, Quick Sort requires fewer comparisons compared to the aforementioned algorithms when dealing with reverse sorted data. Its partitioning technique allows it to avoid unnecessary comparisons and progress towards sorting the data efficiently.

2.4.4 Randomized data



Comment

Bubble Sort and Selection Sort perform the largest number of comparisons. As comparison-based sorting algorithms, they require a large number of comparisons to arrange elements in the correct order for random data.

Quick Sort perform the least number of comparison. For randomized data, Quick Sort achieves relatively fewer comparisons compared to Bubble Sort and Selection Sort.

2.5 Conclusions

With the visualization and experiments above, we have reached the following conclusions to simplify the task of choosing the right sorting algorithm.

2.5.1 Speed

Fastest (Average):

- Large datasets with random order: Merge Sort, Quick Sort (average case) - $O(n \log n)$ complexity.
- Nearly sorted data: Insertion Sort - $O(n)$ complexity.

Slowest:

- Selection Sort, Bubble Sort, Shaker Sort - $O(n^2)$ complexity for all data orders.

2.5.2 Stability

Stable:

- Insertion Sort, Selection Sort, Merge Sort, Counting Sort, Bubble Sort

Unstable:

- Shaker Sort, Shell Sort, Heap Sort, Quick Sort, Radix Sort, Flash Sort

2.5.3 Data Size

Large Datasets:

- Prioritize Merge Sort or Quick Sort (average case) for their superior $O(n \log n)$ time complexity.
- But when it is tight on memory, we should chose Heap Sort over Merge Sort for its in-place characteristic.

Small Datasets:

- Insertion Sort can be efficient for nearly sorted data ($O(n)$).
- For random order, consider a hybrid approach, using a faster algorithm like Quick Sort for the initial sort and then switching to Insertion Sort for the nearly sorted result (potentially reducing swaps).

Special Datasets:

- For integer data value and the values are small, consider using counting sort or Radix sort. ($O(n + k)$ with k is the maximum value for counting sort and $O(n \cdot k)$ with k is the length of the longest number).
- Consider using Radix sort When sorting in lexicographical order.

3 Project organization and Programming notes

3.1 Project organization

Having an easy to followed and well organized project plan is crucial. It keeps the project maintainable and ensure the workflow of team members. This section shows how files in this project was managed and how the team could efficiently shared resources and worked together without getting jammed by one's mistake.

The source files are separated into several folders base on their functionalities, **GEN**, **SORTS**, **UTILS**, all three folders are contained in one big **SOURCE** folder. The structure is shown by the directory tree below.

In the project there is a header file along with any **C++** file, except for **main.cpp**, so in the below source tree, only the header file, and **main.cpp** file are shown.

```
SOURCE
├── gen
│   └── gen.hpp
├── sorts
│   ├── bubble_sort.hpp
│   ├── counting_sort.hpp
│   └── ...
├── utils
│   ├── algo_mode.hpp
│   ├── comp_mode.hpp
│   ├── function_call.hpp
│   ├── handle_flag.hpp
│   └── timer.hpp
└── main.cpp
```

3.1.1 Sorting

Our intention is to make this project works with any amount of sorting algorithms. To achieve this, we made all the sorting algorithm organized in one big folder, **sort**, each sorting should be separated by a pair of header and source files. Whenever an algorithm is added, we include it into **./utils/function_call.hpp** and add its prototype into **./utils/function_call.cpp** to use.

Every sorting functions need to follow the same prototype template. With the idea of maintaining a clear code structure, that can be called by a single function with identical set of arguments among every sorting algorithms. The template is as follow:

Method	Notes
<code>void <Algorithm Name>_Sort(int* arr, int n);</code>	Used for calling sorting algorithm " Algorithm Name " without any measurement
<code>void <Algorithm Name>_Sort_Count(int* arr, int n, long long& count);</code>	Used for calling sorting algorithm " Algorithm Name " and counting the number of comparisons

Table 1: Sorting template

3.1.2 Utilities

This folder contains all the functions, procedures that are used to handling the command line input and control how files interact with each other.

First, we have **handle-flag.hpp**, this file is used to distinguish which command that user want to run and call the appropriate function. After the program knew which command user want, the program then navigate to either **algo-mode.hpp** or **comp-mode.hpp** to execute the command corresponding to each mode.

The **functional-call.hpp** contains some useful functions, one of them is the function that call other sorting function, and some others such as the function that get the count of comparison, the function that get the executed time of a sorting algorithm.

The last file, **timer.hpp**, contains the prototype of the timer implemented in **timer.cpp** file.

3.1.3 Detail function usage in each header file

We will get into detail usage of each function and understand how every component could flawlessly work together. Each file will have a table explaining its functions.

Method	Usages
<code>void Compare_Mode(int argc, char* argv[]);</code>	handle comparison mode
<code>void Algorithm_Mode(int argc, char* argv[]);</code>	handle algorithm mode

Table 2: **handle_flag.hpp**

Method	Usages
<code>void Call_Sort(const string name, int* arr, int n);</code>	Used to make the appropriate sorting algorithm calls, with name is the algorithm name
<code>void Call_Sort_Counting(const string name, int* arr, int n, long long& cntComp);</code>	same as <code>Call_Sort(...)</code> but will return the number of comparisons
<code>int *Load_Array(string inputPath, int &size);</code>	Use to load an array from file inputPath and return a pointer
<code>void Dump_Array(string outputPath, int *arr, int size);</code>	Used to write an array to file outputPath
<code>void Make_Copy(int* arr, int *copy, int size);</code>	copying contents from arr to copy
<code>pair<double, double> Get_Time(const string& algo, int *arr, int size);</code>	Uses to get the running time of a sorting algorithm, algo , and returns two values of time, firstly is time in seconds, and secondly is the time in milliseconds. This function will call the above <code>Call_Sort</code> function.
<code>long long Get_Count(const string& algo, int *arr, int size);</code>	Uses to get the numbers of comparisons of a running algorithm, algo , and returns that number. This function will call the above <code>Call_Sort_Counting(...)</code> function.

Table 3: **function_call.hpp**

Method	Usages
<code>void Handle_Command_1(string algo1, string givenInput, string outputParameter);</code>	Those will be called by functions in function_call.hpp and perform evaluating tasks
<code>void Handle_Command_2(string algo1, int inputtSize, string inputOrder, string outputParameter);</code>	
<code>void Handle_Command_3(string algo1, int inputtSize, string outputParameter);</code>	

Table 4: **algo_mode.hpp**

Method	Usages
<code>void Handle_Command_4(string algo1, string algo2, string givenInput);</code>	Those will be called by functions in function_call.hpp and perform comparing tasks
<code>void Handle_Command_5(string algo1, string algo2, int inputSize, string inputOrder);</code>	

Table 5: **comp_mode.hpp**

Method	Usages
<code>void start();</code>	Starts the timer whenever this method is invoked
<code>void stop();</code>	Stops the timer whenever this method is invoked
<code>double elapsedMilliseconds();</code>	Returns the running time in milliseconds starting from the moment the timer start if the timer has not stop, else returns the time from the moment timer started to the moment it stopped.
<code>double elapsedSeconds();</code>	Same as the <code>elapsedMilliseconds()</code> method, but return the times in seconds.

Table 6: Timer class in timer.hpp

3.2 Programming notes

3.2.1 Libraries used

This project could not be completed with out the integration of these standard C++ libraries.

- **iostream**: Used for input and output operations. It provides facilities for reading from and writing to the console.
- **fstream**: Provides facilities for file input and output. It allows reading from and writing to files.
- **cstring**: Contains functions for manipulating C-style strings, such as copying, concatenation, and comparison. This library helps us on handling command line flags.
- **cstdlib**: Includes functions for general purpose utilities, such as memory allocation, random number generation, and system commands.
- **ctime**: Provides functions for manipulating and formatting time and date, as well as measuring program execution time.
- **string**: Contains the `std::string` class, which provides a way to use and manipulate strings of characters in a more convenient and safer manner than C-style strings.
- **chrono**: Offers facilities for measuring and working with time durations, clocks, and time points with milliseconds accuracy.
- **cmath**: Provides mathematical functions such as trigonometric, exponential, and logarithmic operations.

3.2.2 How to build

The project was compiled and tested on **MinGW-W64 v8.1.0** compiler. To build this project, you can follow these steps:

Step 1: Navigate to `./SOURCE/` folder.

Step 2: Run the following command, you may change the `name` with any name you want:

```
g++ -std=c++14 ./sorts/*.cpp ./utils/*.cpp ./gen/*.cpp ./*.cpp -o name.exe
```

Step 3: Now run you can run the program by calling its executable file , `".exe"` file.

3.2.3 Running example

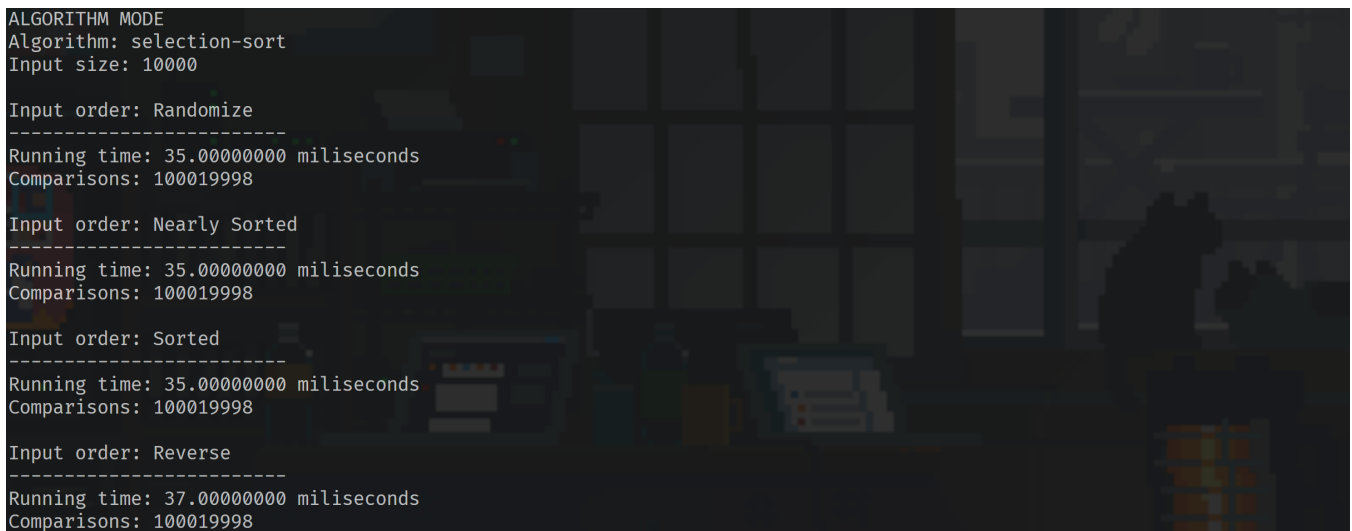
First we build with the command:

```
g++ -std=c++14 ./sorts/*.cpp ./utils/*.cpp ./gen/*.cpp ./*.cpp -o Analyze.exe
```

Then we run with the command:

```
Analyze.exe -a selection-sort 10000 -both
```

And get:



```
ALGORITHM MODE
Algorithm: selection-sort
Input size: 10000

Input order: Randomize
-----
Running time: 35.00000000 milliseconds
Comparisons: 100019998

Input order: Nearly Sorted
-----
Running time: 35.00000000 milliseconds
Comparisons: 100019998

Input order: Sorted
-----
Running time: 35.00000000 milliseconds
Comparisons: 100019998

Input order: Reverse
-----
Running time: 37.00000000 milliseconds
Comparisons: 100019998
```

Figure 2: Screenshot

4 List of reference

References

- [1] Georgy Gimel'farb *Insertion Sort: Analysis of Complexity, COMPSCI 220 Algorithm and Data Structure.*
- [2] Anany Levitin *Introduction to The Design and Analysis of Algorithms, 3rd Edition, page 227-228*
- [3] Anany Levitin *Introduction to The Design and Analysis of Algorithms, 3rd Edition, page 229-230*
- [4] Sedgewick, Robert. *Algorithms in C: Fundamentals, Data Structures, Sorting, Searching, Parts 1-4*
- [5] Sedgewick, R. *"Implementing Quicksort programs"*. Comm. ACM. 21 (10).
- [6] Wild, Sebastian; Nebel, Markus E. (2012). *Average case analysis of Java 7's dual pivot quicksort.* European Symposium on Algorithms.
- [7] WikipediA. Quick Sort. Variants. *Three-way radix quicksort.*
- [8] WikipediA. Quick Sort. Formal analysis. *Average-case analysis.*
- [9] WikipediA. Quick Sort. Formal analysis. *Best-case analysis.*
- [10] WikipediA. Radix Sort. *In-place MSD radix sort implementations.*
- [11] WikipediA. Radix Sort. *Tree-based radix sort.*
- [12] WikipediA. *Radix Tree.*
- [13] Educative. *Bucket Sort*
- [14] Educative. Flash Sort. *Idea and Code Example.*
- [15] IO Stream. *Shaker Sort*
- [16] Programiz. *Heap Sort Algorithm*
- [17] StackExchange. *Ping-pong Merge Sort*
- [18] Baeldung. *Bottom-up Merge Sort*
- [19] GitHub (user: leduythuucs). Quick Sort. *Implementation of Quick Sort.*
- [20] Altcademy Blog. Quick Sort. *Real-World Examples and Scenarios of Heap Sort.*
- [21] VineetKumar *at English Wikipedia - Transferred from en.wikipedia to Commons by Eric Bau-*
man using CommonsHelper., Public Domain

5 AI & tools usages

In our research, we leveraged a variety of advanced tools to streamline the process and improve the quality of our work:

Large Language Models (LLMs) like Chatbot GPT-3.5 and Google Gemini: These powerful AI tools served as research assistants. We utilized their capabilities to gather information enhance our writing process.

Mathcha.io: this online illustration tool helped us bring our algorithms to life. We created clear and informative visuals of the sorting processes.

Microsoft Excel: We used Excel to process and organize our data, allowing us to generate clear and insightful graphs.