UNIVERSITY OF SCIENCE - VNUHCM

INFORMATION TECHNOLOGY FACULTY

COURSE NAME

DATA STRUCTURES AND ALGORITHMS

Sorting Algorithms

Group 08:

Nguyễn Tiến Luật	23127221
Nguyễn Tấn Lộc	23127406
Lê Thanh Phong	23127452
23CLC08	

Lecturers:
Phan Thị Phương Uyên
Trần Hoàng Quân

Ho Chi Minh City, June 30th 2024

Contents

List of Figures

List of Tables

1 Introduction

In this project, we will conduct researches on sorting algorithms. Sorting is the process of organizing a collection of data into the ascending or descending order. [26]
You can see an example in this figure.

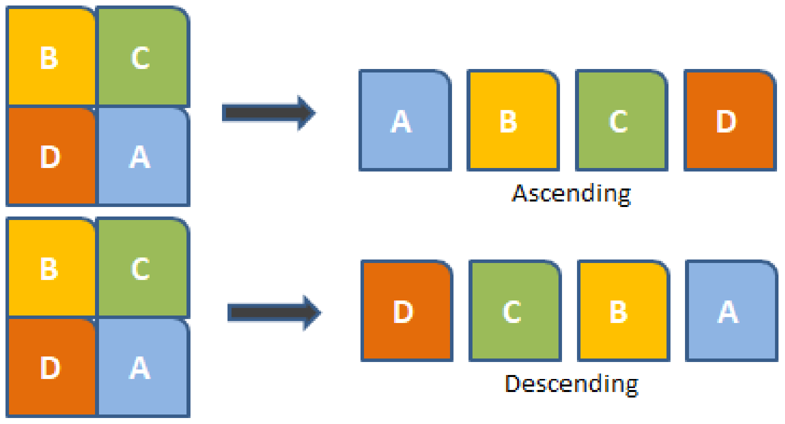

Figure 1: Overview of Sorting

[27]
Some terminologies:

- In-place sorting: An in-place sorting algorithm does not use any additional memory, it means the algorithm uses **constant spaces**. It sorts the list by reorder the elements within the list. Example: Selection Sort, Bubble Sort, Heap Sort, ... [26]
- Internal sorting: It called internal sorting when all data is **placed** in the **main memory** (or internal memory). It means the problem cannot take input beyond its size. Example: Selection Sort, Bubble Sort, Heap Sort, ... [26]
- External sorting: It called external sorting when all the data that needs to be sorted **cannot be placed** in the **main memory at a time**. It means external sorting is used for a massive amount of data. Example: Merge Sort, External Radix Sort, ... [26]
- Stable sorting: A stable sorting algorithm **does not change the positions** of two same data in the same order in sorted list. Example: Merge Sort, Insertion Sort, Bubble Sort, ... [26]
- Unstable sorting: An unstable sorting algorithm **changes the position** of two same data in the different order in sorted list. Example: Quick Sort, Heap Sort, Shell Sort., ... [26]

You can see an example for the stability of sorting algorithm in the below figure.

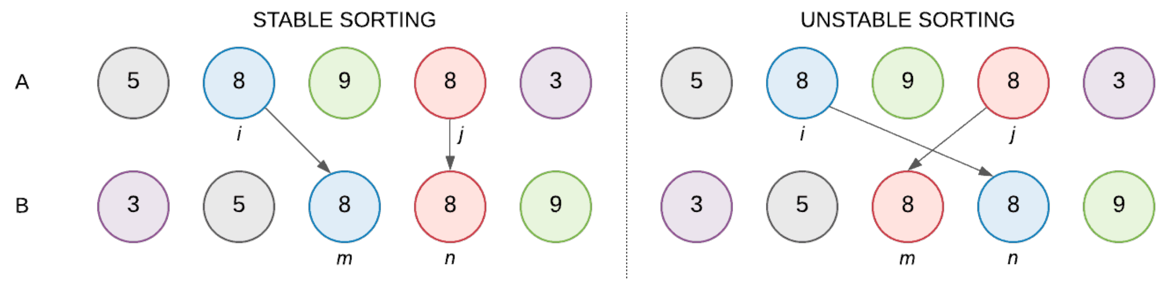

Figure 2: Stability of Sorting Algorithm

[28]

Application of sorting algorithms:

- Searching: Sorting is a crucial step in searching like binary search, ternary search, ... [26]
- Data management: Sorting data make it easier to search, retrieve and analyze. [26]. For example, we can easily perform below tasks:
  - Finding closest pair
  - Finding duplicated elements

  [1]
- Database optimization: Sorting data in databases improves query performance. [26]
- Data analysis: Sorting plays a vital role in statistical analysis, financial modeling, and other data-driven fields. [26]
  - Finding median [1]
  - Identify patterns, trends, and outliers in datasets [26]

In this project, we only discuss about 12 common sorting algorithms:

- Selection Sort
- Insertion Sort
- Shell Sort
- Bubble Sort
- Heap Sort
- Merge Sort
- Quick Sort

- Radix Sort
- Counting Sort
- Shaker Sort
- Flash Sort

We only experiment on the input size is 10000, 30000, 50000, 100000, 300000 and 500000 elements for 4 types of data: random order data, sorted data, nearly sorted data and reverse data.

2 Information

2.1 Computer specifications

- **Processor:** 12th Gen Intel(R) Core(TM) i5-12500H
- **Memory:** 16GB
- **Code editor:** Visual Studio Code
- **Compiler:** g++

2.2 Output specifications

Command lines(for Visual Studio Code):

First you need to enter this command in the terminal and then follow the rules below to run the command.

```
g++ -std=c++17 Main.cpp SortFunction.cpp ShowName.cpp DataGenerator.cpp -o <name>.exe
```

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

In order to run the program, make sure you follow the command lines to run. Here is how the program should work:

Step 1: "-a" for Algorithm Mode, executing a specific sorting algorithm (Command 1, 2, 3). "-c" for Comparison Mode, comparing 2 different sorting algorithms (Command 4, 5).

Step 2: Depending on the specific requirements like sorting algorithm, size of input data, data type and the parameters to call the functions and perform.

Step 3: Calculating the parameters **time**, algorithm's run time and **comp**, number of comparison need for the sorting process.

Step 4: Display the required value and write down on the file input.txt if necessary.

Note:

1. Mode:
   - -a : Algorithm Mode
   - -c : Comparison Mode
2. Sorting Algorithm: name-sort
   - selection-sort: Selection Sort
   - insertion-sort: Insertion Sort
   - shell-sort: Shell Sort
   - bubble-sort: Bubble Sort
   - heap-sort: Heap Sort
   - merge-sort: Merge Sort
   - quick-sort: Quick Sort
   - radix-sort: Radix Sort
   - counting-sort: Counting Sort
   - shaker-sort: Shaker Sort
   - flash-sort: Flash Sort
3. Data size
4. Data type
   - -rand : Random data
   - -sorted : Sorted data
   - -nsorted : Nearly Sorted data
   - -reverse : Reverse data

5. Parameters (Algorithm Mode)
    - -time : Running time
    - -comp : Number of comparison operator
    - -both : Running time and number of comparison operator

3 Sorting Algorithms

3.1 Selection Sort

3.1.1 Ideas

- The list is divided into two sub-lists, sorted and unsorted, which are divided by an imaginary wall.
- Selects the smallest (or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted part. This process is repeated for the remaining unsorted portion until the entire list is sorted.

[23]

3.1.2 How it works

Step 1: Start with the first element as the minimum.

Step 2: Scan the remaining elements to find the actual minimum.

Step 3: Swap the found minimum with the first element.

Step 4: Move to the next element and repeat until the list is sorted.

[23]

Pseudo Code:

Algorithm 1 Selection Sort

```
Require: An array of $n$ elements $A = [a_1, a_2, \dots, a_n]$
Ensure: A sorted array in ascending order
 1: for $i = 1$ to $n - 1$ do
 2:     $minIndex \leftarrow i$
 3:     for $j = i + 1$ to $n$ do
 4:         if $A[j] < A[minIndex]$ then
 5:             $minIndex \leftarrow j$
 6:         end if
 7:     end for
 8:     if $minIndex \neq i$ then
 9:         swap $A[i]$ and $A[minIndex]$
10:     end if
11: end for
```

[23]

3.1.3 Analyze

- **Best case**: $O(n^2)$.
- **Average case**: $O(n^2)$
- **Worst case**: $O(n^2)$.
- **Stability**: not stable
- **Space Complexity**: O(1).

[23]

Note: Comparison of Selection Sort

The n − 1 calls to swap result in n − 1 exchanges. Each exchange requires three assignments, or data moves. Thus, the calls to swap require $3 * (n - 1)$ moves.

Number of key comparisons $= 1 + 2 + ... + n - 1 = n * \frac{(n-1)}{2}$.

[6]

Variant of Selection Sort is **Heap Sort**. Heap sort uses a binary heap to improve the efficiency of selection sort, reducing the time complexity to $O(n \log n)$. [12]

3.1.4 Strength

- Simple and easy to understand, easy to deploy.
- The behavior of the selection sort algorithm does not depend on the initial organization of data.
- It only requires O(n) moves.
- A selection sort could be a good choice if data moves are costly but key comparisons are not costly (short keys, long records).
- If we stop the algorithm midway, we will still get a sorted array of processed elements. Ex: top 100 of 1000 student.

[19]

3.1.5 Weakness

- If sorting a very large array, selection sort algorithm probably too inefficient to use because the selection sort algorithm requires $O(n^2)$ key comparisons.
- Selection sort is unstable sort, it can change the relative order of elements with equal values.

[19]

3.2 Insertion Sort

3.2.1 Ideas

Insertion sort inserts each element of an unsorted list into its correct position in a sorted portion of the list. Different from selection sort, insertion sort a stable sorting algorithm. [14]

3.2.2 How it works

Step 1: Start with the second element.

Step 2: Compare it with elements before it and insert it in the correct position.

Step 3: Repeat until the list is sorted.

[14]

Pseudo Code:

Algorithm 2 Insertion Sort

```
Require: An array of $n$ elements $A = [a_1, a_2, \ldots, a_n]$
Ensure: A sorted array in ascending order
1: for $i = 2$ to $n$ do
2:     $key \leftarrow A[i]$
3:     $j \leftarrow i - 1$
4:     while $j > 0$ 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
```

[14]

3.2.3 Analyze

- **Worst case**: O(n^2)
- **Average case**: O(n^2)
- In sorted array, the inner for loop exits immediately, making the sort O(n) in its best case.
- **Space Complexity**: O(1).
- **Stability**: stable

[14]

Note: Comparison of Insertion sort

Number of key comparisons $= 1 + 2 + ... + n - 1 = n * \frac{(n-1)}{2}$ in worst case and average case. [6]

Variants of Insertion Sort are **Shell Sort** and **Binary Insertion Sort**. Shell Sort uses a gap sequence which element to compare and sort. Binary Insertion Sort uses Binary Search to find the suitable position to insert the element. [25] [3]

3.2.4 Strength

- Insertion sort a stable sorting algorithm, meaning that elements with equal values maintain their relative order in the sorted output. [14]
- Simple and easy to understand, easy to implement. [14]
- Insertion sort is effective for small arrays that are nearly sorted. [14]

3.2.5 Weakness

- Insertion sort is not suitable for inversely sorted data. [14]
- High average time complexity. This makes the algorithm unsuitable for large datasets, as execution time increases rapidly with the number of elements. [14]
- It takes a lot of time to move data.

3.3 Shell Sort

3.3.1 Ideas

The idea of Shell Sort is to allow the exchange of far items. In Shell sort, we make the array h-sorted for a large value of h. We keep reducing the value of h until it becomes 1. An array is said to be h-sorted if all sub-lists of every h^{th} element are sorted. [25]

3.3.2 How it works

Step 1: Start with a large interval.

Step 2: Perform insertion sort on elements separated by the interval.

Step 3: Reduce the interval and repeat until the interval is 1.

[25]

Pseudo Code:

Algorithm 3 Shell Sort

```
Require: An array of $n$ elements $A = [a_1, a_2, \ldots, a_n]$
Ensure: A sorted array in ascending order
 1: $gap \leftarrow \lfloor n/2 \rfloor$
 2: while $gap > 0$ do
 3:     for $i = gap + 1$ to $n$ do
 4:         $temp \leftarrow A[i]$
 5:         $j \leftarrow i$
 6:         while $j > gap$ and $A[j - gap] > temp$ do
 7:             $A[j] \leftarrow A[j - gap]$
 8:             $j \leftarrow j - gap$
 9:         end while
10:         $A[j] \leftarrow temp$
11:     end for
12:     $gap \leftarrow \lfloor gap/2 \rfloor$
13: end while
```

3.3.3 Analyze

- **Best case** : $O(n \log n)$, sorted data, sequence of gaps leads to few comparisons.
- **Worst case**: O(n^2), many elements are far from their final position.
- **Average case**: O(n^2)
- **Stability**: not stable.
- **Space Complexity**: O(1)

[25]

Note: Gap

In addition to the shell gap selection, we have many other gap options, and it greatly affects the time complexity [10]

- Hibbard's Sequence (1, 3, 7, 15, ..., $O(n^k - 1)$): Worst-case complexity $O(n^{\frac{3}{2}})$.
- Sedgewick's Sequence (1, 5, 19, 41, 109, ...): Worst-case complexity $O(n^{\frac{4}{3}})$.
- Pratt's Sequence (terms of the form $2^i j$): Worst-case complexity $O(n \log n)$.

[6]

Variant of Shell Sort is using different gaps. **Pratt's Sequence** often has better performance than Hibbard's Sequence and Sedgewick's Sequence. [6]

3.3.4 Strength

- Shell sort effective for small and medium-sized arrays.
- In insertion sort to reduce the number of operations.
- When the input array is nearly sorted, shell sort can operate very efficiently with a complexity close to O($n \log n$).
- Shell sort can be customized with various gap sequences, allowing adjustment of performance based on the characteristics of the input data.
- Shell sort is easy to understand and implement.

[25]

3.3.5 Weakness

- When working with random data and large sizes, algorithms like Quick Sort or Merge Sort may be more effective. [24]
- Shell Sort is not stable, meaning it may change the relative order of elements with equal keys. This can be problematic in certain applications where stability is crucial. [24]

3.4 Bubble Sort

3.4.1 Ideas

- Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. [5]

3.4.2 How it works

Step 1: Check if the first element in the input array is greater than the next element in the array.

Step 2: If it is greater, swap the two elements; otherwise move the pointer forward in the array.

Step 3: Repeat Step 2 until we reach the end of the array.

Step 4: Check if the elements are sorted; if not, repeat the same process (Step 1 to Step 3) from the last element of the array to the first.

Step 5: The final output achieved is the sorted array.

[5]

Pseudo Code:

Algorithm 4 Bubble Sort

```
Require: An array of $n$ elements $A = [a_1, a_2, \dots, a_n]$
Ensure: A sorted array in ascending order
for $i = 1$ to $n - 1$ do
    for $j = 1$ to $n - i$ do
        if $A[j] > A[j+1]$ then
            swap $A[j]$ and $A[j+1]$
        end if
    end for
end for
```

3.4.3 Analyze

[6]

- **Best case** : O(n^2), sorted data, Bubble Sort traverse once, during which n − 1 comparisons and no exchanges occur.
- **Worst case**: O(n^2)
- **Average case**: O(n^2)
- **Stability**: stable.
- **Space Complexity**: O(1).

[5]

Note: Comparison of Bubble Sort

A bubble sort will require a total of $(n-1) + (n-2) + ... + 1 = n * \frac{n-1}{2}$ comparisons and the same number of exchanges.

Recall that each exchange requires three data moves. Thus, altogether there are $2 * n * (n-1) = 2 * n^2 - 2 * n$ major operations in the worst case.

[6]

Optimization of Bubble Sort uses a flag **swapped** to check if the remaining elements are sorted or not. If remaining elements are sorted, stop the loop. So that it decrease the comparison operators.

Variant of Bubble Sort is **Shaker Sort**. Shaker traverse both directions at a time. Otherwise, Bubble Sort traverse in one direction. [5] [7]

Algorithm 5 Bubble Sort with Flag

Require: An array of n elements $A = [a_1, a_2, \dots, a_n]$

Ensure: A sorted array in ascending order

```
1: for $i = 1$ to $n - 1$ do
2:     $flag \leftarrow false$
3:     for $j = 1$ to $n - i$ do
4:         if $A[j] > A[j + 1]$ then
5:             swap $A[j]$ and $A[j + 1]$
6:             $flag \leftarrow true$
7:         end if
8:     end for
9:     if $flag = false$ then
10:        break
11:    end if
12: end for
```

[5]

3.4.4 Strength

- Bubble sort is easy to understand and implement.
- It does not require any additional memory space.
- It is a stable sorting algorithm, meaning that elements with the same key value maintain their relative order in the sorted output.
- Bubble sort is effective for small arrays that are nearly sorted.

[5]

3.4.5 Weakness

- Bubble sort has a time complexity of $O(n^2)$ which makes it very slow for large data sets.
- Bubble sort is a comparison-based sorting algorithm that uses a comparison operator to determine the order of elements. This can limit its efficiency in some cases.[5]

3.5 Heap Sort

3.5.1 Ideas

- Heap is a collection of n elements $(a_0, a_1, ...a_{n-1})$ in which every element (at position i) in the first half is greater than or equal to the elements at position 2i+1 and 2i+2. [19]
- Heap in above definition is called max-heap. [19]

- First, convert the array into a heap using heapify. Then, repeatedly delete the root of the max-heap, replace it with the last node, and heapify the root until the heap size is 1. [12]

3.5.2 How it works

Step 1: Build a max heap from the list.

Step 2: Swap the root of the heap with the last element.

Step 3: Reduce the heap size and heapify the root.

Step 4: Repeat until the heap is empty.

[12]

Pseudo Code:

Algorithm 6 Heap Sort

```
Require: An array of $n$ elements $A = [a_1, a_2, \dots, a_n]$
Ensure: A sorted array in ascending order
1: Build a max heap from the array
2: for $i = n$ down to 2 do
3:     swap $A[1]$ and $A[i]$
4:     Heapify the array $A[1 \dots i-1]$
5: end for
```

[12]

3.5.3 Analyze

- **Best case**: $O(n \log n)$.
- **Worst case**: $O(n \log n)$.
- **Average case**: $O(n \log n)$.
- **Stability**: not stable.
- **Space complexity**: $O(1)$.

[12]

Variant of Heap Sort is **Smooth Sort**. Smooth Sort (Leonardo Sort) uses Leonardo heap, a binary tree with the property that the root of each subtree has one fewer element than its parent and the shape of tree is determined by the Leonardo number. Leonardo number follows the rule: $L(n) = L(n-1) + L(n-2) + 1$, with $L(0) = 1$ and $L(1) = 1$. [15]

Pseudo Code:

Algorithm 7 Smooth Sort

```
Initialize Leonardo heap sizes array $L$
$L \leftarrow [1, 1]$
while $L[-1] + L[-2] + 1 \leq n$ do
    $L \leftarrow L \cup [L[-1] + L[-2] + 1]$
end while
for $i \leftarrow 0$ to $n - 1$ do
    Add $A[i]$ to the Leonardo heap
    Adjust the heap to maintain the Leonardo property
end for
for $i \leftarrow n - 1$ to 1 do
    Remove the maximum element from the heap
    Adjust the heap to maintain the Leonardo property
end for
```

[15]

3.5.4 Strength

- With a time complexity of O(n log n) on average and worst-case, Heap Sort performs well for sorting large arrays.
- Unlike algorithms like Quick Sort, Heap Sort avoids the risk of stack overflow for very large datasets due to its iterative nature.
- Priority queues: The first element of the max-heap is always the largest.

[13]

3.5.5 Weakness

- Heap sort is not easy to understand and more complex to implement.
- Heap Sort is not stable, meaning it may change the relative order of elements with equal keys. This can be problematic in certain applications where stability is crucial.
- Despite its good theoretical time complexity, Heap Sort can be slower than other sorting algorithms for smaller datasets due to the constant factors in heap operations. [13]

3.6 Merge Sort

3.6.1 Ideas

This technique can be divided into the following three parts:

- Divide: This involves dividing the problem into smaller sub-problems.
- Conquer: Solve sub-problems by calling recursively until solved.

- Combine: Combine the sub-problems to get the final solution of the

[19]

3.6.2 How it works

Step 1: Divide the list into two halves.

Step 2: Recursively sort the two halves.

Step 3: Merge the two sorted halves.

[16]

Pseudo Code:

Algorithm 8 Merge Sort

```
Require: An array of $n$ elements $A = [a_1, a_2, \ldots, a_n]$
Ensure: A sorted array in ascending order
1: if $n > 1$ then
2:     $mid \leftarrow \lfloor n/2 \rfloor$
3:     $L \leftarrow A[1 \ldots mid]$
4:     $R \leftarrow A[mid + 1 \ldots n]$
5:     Merge Sort($L$)
6:     Merge Sort($R$)
7:     Merge $L$ and $R$ into $A$
8: end if
```

3.6.3 Analyze

- **Best Case**: O($n \log n$), sorted or nearly sorted data.
- **Worse case**: ($n \log n$). reverse order data.
- **Average case**: O($n \log n$).
- **Stability**: not stable.
- **Space Complexity**: O(n), additional space is required for the temporary array used during merging.

[16]

Note: Comparison of Merge Sort

In the worst case, the number of comparisons made by merge sort is $O(n \log n)$, specifically:

$$\begin{aligned} C(n) &= C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C\left(\left\lceil \frac{n}{2} \right\rceil\right) + (n-1) \\ &= \sum_{k=0}^{\log_2 n-1} (n-1) \\ &= n \log_2 n - n + 1 \end{aligned}$$

[6]

Variant of Merge Sort is **Tim Sort**. Tim Sort is a hybrid algorithm derived from Merge Sort and Insertion Sort. Tim Sort exploits the existing order in the data to minimize the number of comparisons and swaps Tim Sort is often used for nearly sorted data. Otherwise, Merge Sort is efficient for random order data. [29]

Algorithm 9 Tim Sort Algorithm

```
Data: Array $arr$ of size $n$
Result: Sorted array $arr$
Function TimSort($arr$, $n$):
    if $n \leq 1$ then
        return
    end
    if $n < insertion\_sort\_threshold$ then
        Run Insertion Sort on $arr$ return
    end
    Divide $arr$ into two halves: $left$ and $right$ TimSort($left$, $\frac{n}{2}$) TimSort($right$, $n - \frac{n}{2}$)
     Merge $left$ and $right$ into $arr$
TimSort($arr$, $n$)
```

[29]

3.6.4 Strength

- Merge Sort is efficient for external sorting with large datasets that don't fit in memory. It breaks data into smaller chunks, sorts them on disk, and then merges them.[17]
- It is a stable sorting algorithm, meaning that elements with the same key value maintain their relative order in the sorted output.
- Merge Sort has a worst-case time complexity of O($n \log n$) , which means it performs well even on large datasets.[16]

3.6.5 Weakness

- Merge Sort requires additional memory to store the sub-arrays during merging. This can be a disadvantage for small datasets or systems with limited resources.[17]

3.7 Quick Sort

3.7.1 Ideas

Quick Sort selects a pivot element and partitions the array into two halves, then recursively sorts each half. [20]

3.7.2 How it works

Step 1: Choose a pivot element.

Step 2: Partition the array around the pivot.

Step 3: Recursively sort the subarrays.

[20]

Pseudo Code:

Algorithm 10 Quick Sort

```
Require: An array of $n$ elements $A = [a_1, a_2, \ldots, a_n]$
Ensure: A sorted array in ascending order
1: if $n > 1$ then
2:     Choose a pivot element
3:     Partition the array into two sub-arrays: $L$ and $R$
4:     $L$ contains elements less than or equal to the pivot
5:     $R$ contains elements greater than the pivot
6:     Quick Sort($L$)
7:     Quick Sort($R$)
8:     Combine $L$, pivot, and $R$ into $A$
9: end if
```

[20]

3.7.3 Analyze

- **Best case:** O($n \log n$), depend on number of digits (k)
- **Worst case:** O(n^2)
- **Average case:** O($n \log n$).
- **Stability:** not stable
- **Space Complexity:** O($\log n$) and larger than if in worst case.

[20]

Note: Comparison of Quick Sort

If the pivot is placed at k, we have a recursive cost of $C_k + C_{n-k}$. Each of these n possibilities is equally likely, leading to the average recursive cost of [21]

$$\begin{aligned}
&\frac{1}{n}\sum_{k=1}^{n}[C(k-1)+C(n-k)]\\
&=\frac{1}{n}\sum_{k=1}^{n}C(k-1)+\frac{1}{n}\sum_{k=1}^{n}C(n-k)\\
&=\frac{2}{n}\sum_{k=0}^{n-1}C(k)
\end{aligned}$$

In the worst case, Merge Sort uses approximately 39 percent fewer comparisons than Quick Sort does in its average case, and in terms of moves, Merge Sort's worst case complexity is $O(n\log n)$ - the same complexity as Quick Sort's best case. [21]

$$\begin{aligned}
&C_n=(n+1)+\frac{2}{n}\sum_{k=0}^{n-1}C_k\\
&nC_n=n(n+1)+2\sum_{k=0}^{n-1}C_k\\
&(n-1)C_{n-1}=(n-1)n+2\sum_{k=0}^{n-2}C_k\\
&nC_n-(n-1)C_{n-1}=2n+2C_{n-1}\\
&\frac{C_n}{n+1}=\frac{C_{n-1}}{n}+\frac{2}{n+1}\\
&\frac{C_n}{n+1}=\frac{2}{3}+\frac{2}{4}+\cdots+\frac{2}{n+1}\\
&C_n=2(n+1)\left(\frac{1}{3}+\frac{1}{4}+\cdots+\frac{1}{n+1}\right)\\
&C_n\sim 2(n+1)\ln n\\
&C_n\approx 1.39n\log_2 n
\end{aligned}$$

[6]

3.7.4 Strength

- Very efficient for large datasets.
- Good average-case performance.

- Require small amount

[20]

3.7.5 Weakness

- Worst-case performance can be O(n^2).
- Not stable without modifications.
- Not efficient for small datasets.

[20]

3.8 Radix Sort

3.8.1 Ideas

Radix Sort sorts numbers digit by digit starting from the least significant digit to the most significant digit using a stable counting sort. [22]

3.8.2 How it works

Step 1: Sort based on the least significant digit.

Step 2: Move to the next significant digit and repeat until the most significant digit.

[22]

Pseudo Code:

Algorithm 11 Radix Sort

```
Require: An array of $n$ elements $A = [a_1, a_2, \ldots, a_n]$
Ensure: A sorted array in ascending order
1: Find the maximum number to know the number of digits
2: for each digit $d$ from least significant to most significant do
3:     Sort the array based on the digit $d$ using a stable counting sort
4: end for
```

[22]

3.8.3 Analyze

- **Best case:** O(nk), depend on number of digits (k)
- **Worst case:** O(nk)
- **Average case:** O(nk)
- **Stability:** stable

- **Space Complexity:** O(nk)

[22]

Note:

- **n:** the number of element need to be sorted.
- **k:** the number of digits of maximum value.

Optimizations of Radix Sort are **LSD Radix Sort** (Least Significant Digit Radix Sort) and **MSD Radix Sort** (Most Significant Radix Sort). LSD Radix Sort process digits from the least significant to the most significant digit and MSD Radix Sort process digits in opposite direction. LSD Radix Sort is generally preferred due to its simplicity and effectiveness in most cases. MSD radix sort is used when dealing with data of varying lengths and when higher performance is necessary. [18]

Algorithm 12 LSD Radix Sort

```
Require: An array of $n$ integers $A = [a_1, a_2, \ldots, a_n]$ with maximum number of digits $d$
Ensure: A sorted array in ascending order
1: function LSD-Radix-Sort($A, n, d$)
2: for $digit \leftarrow 1$ to $d$ do
3:     Create 10 buckets, $B[0] \ldots B[9]$
4:     for $i \leftarrow 0$ to $n - 1$ do
5:         Extract the $digit^{th}$ digit from $A[i]$
6:         Place $A[i]$ into bucket $B[digit]$
7:     end for
8:     Gather elements back from buckets into $A$
9: end for
```

[18]

Algorithm 13 MSD Radix Sort

```
Require: An array of $n$ integers $A = [a_1, a_2, \ldots, a_n]$ with maximum number of digits $d$
Ensure: A sorted array in ascending order
1: function MSD-Radix-Sort($A, l, r, digit$)
2: if $l < r$ and $digit \leq d$ then
3:     Create 10 buckets, $B[0] \ldots B[9]$
4:     for $i \leftarrow l$ to $r$ do
5:         Extract the $digit^{th}$ digit from $A[i]$
6:         Place $A[i]$ into bucket $B[digit]$
7:     end for
8:     Reorder elements in $A$ according to buckets
9:     for $b \leftarrow 0$ to 9 do
10:        call MSD-Radix-Sort($A, l, r, digit + 1$) {Recursive call for each bucket}
11:    end for
12: end if
```

[18]

3.8.4 Strength

- Very efficient for sorting integers or strings with a fixed number of digits.
- Linear time complexity relative to the number of digits.
- Stable

[2]

3.8.5 Weakness

- Requires additional space for the output array.
- Limited to certain types of data (e.g., integers, fixed-length strings).
- Not efficient for small data sets or data sets with a small number of unique keys.
- Require the data being sorted can be represented in a fixed number of digits.

[2]

3.9 Counting Sort

3.9.1 Ideas

Counting Sort counts the occurrences of each distinct element and uses this information to place the elements in the correct position. [8]

3.9.2 How it works

Step 1: Count occurrences of each element.

Step 2: Compute the starting index for each element.

Step 3: Place elements in the output array based on the starting index.

[8]

Pseudo Code:

Algorithm 14 Counting Sort

```
Require: An array of $n$ elements $A = [a_1, a_2, \ldots, a_n]$, with integers in range 0 to $k$
Ensure: A sorted array in ascending order
1: Create a count array $C[0 \ldots k]$ and initialize to zero
2: for $i = 1$ to $n$ do
3:     $C[A[i]] \leftarrow C[A[i]] + 1$
4: end for
5: for $i = 1$ to $k$ do
6:     $C[i] \leftarrow C[i] + C[i-1]$
7: end for
8: for $i = n$ down to 1 do
9:     $output[C[A[i]]] \leftarrow A[i]$
10:    $C[A[i]] \leftarrow C[A[i]] - 1$
11: end for
12: for $i = 1$ to $n$ do
13:    $A[i] \leftarrow output[i]$
14: end for
```

[8]

3.9.3 Analyze

- **Best case:** O(n + k), depend the range of the input (k)
- **Worst case:** O(n + k)
- **Average case:** O(n + k)
- **Stability:** stable
- **Space Complexity:** O(n + k)

[8]

Note:

- **n:** the number of element need to be sorted.
- **k:** the range of elements' value, k = maximum value - minimum value + 1

Variant of Counting Sort is **Radix Sort**. Radix Sort uses Counting Sort as a subroutine to sort numbers based on their individual digits. [22]

3.9.4 Strength

- Efficient for lists with a small range of values, not greater than the input (n).
- Linear time complexity relative to the range and number of elements.
- Stable
- Generally performs faster than all comparison-based sorting algorithms, such as Merge sort, Quick Sort,...
- Easy to implement

[8]

3.9.5 Weakness

- Requires additional space for the count array.
- Inefficient for lists with a large range of values relative to the number of elements.
- Not work efficiently on decimal values.

[8]

3.10 Binary Insertion Sort

3.10.1 Ideas

Binary Insertion Sort is an improvement of Insertion Sort. It uses Binary Search to find the right position to insert the element. [3]

3.10.2 How it works

Step 1: Iterate through each element of the array.

Step 2: For each element, use binary search to find the correct position in the sorted portion of the array.

Step 3: Shift elements to make space for the current element and insert it at the correct position.

Step 4: Repeat until the entire array is sorted.

[3]

Pseudo Code:

Algorithm 15 Binary Insertion Sort

Require: An array of n elements $A = [a_1, a_2, \ldots, a_n]$

Ensure: A sorted array in ascending order

```
for $i = 2$ to $n$ do
    $key \leftarrow A[i]$
    $left \leftarrow 1$
    $right \leftarrow i - 1$
    while $left \leq right$ do
        $mid \leftarrow \lfloor (left + right)/2 \rfloor$
        if $A[mid] > key$ then
            $right \leftarrow mid - 1$
        else
            $left \leftarrow mid + 1$
        end if
    end while
    for $j = i - 1$ down to $left$ do
        $A[j + 1] \leftarrow A[j]$
    end for
    $A[left] \leftarrow key$
end for
```

[3]

3.10.3 Analyze

- **Best case:** O($n \log n$), sorted datasets.
- **Worst case:** O(n^2), reverse datasets.
- **Average case:** O(n^2)
- **Stability:** stable
- **Space complexity:** O(1) [4]

3.10.4 Strength

- Fewer comparisons compared to standard Insertion Sort due to binary search.
- Suitable for small to medium-sized arrays.

[4]

3.10.5 Weakness

- Still has O(n^2) time complexity in the worst case due to the element shifts.
- More complex to implement than standard Insertion Sort.

[4]

3.11 Shaker Sort

3.11.1 Ideas

Shaker Sort is a variant of Bubble Sort. Shaker Sort traverse through a list both direction alternatively. [7]

3.11.2 How it works

Step 1: Perform a forward pass, bubbling the largest element to the end.

Step 2: Perform a backward pass, bubbling the smallest element to the beginning.

Step 3: Repeat until no swaps are needed.

[7]

Pseudo Code:

Algorithm 16 Shaker Sort (Cocktail Sort)

```
Require: An array of $n$ elements $A = [a_1, a_2, \dots, a_n]$
Ensure: A sorted array in ascending order
$left \leftarrow 1$
$right \leftarrow n$
while $left \leq right$ do
    for $i = left$ to $right - 1$ do
        if $A[i] > A[i+1]$ then
            swap $A[i]$ and $A[i+1]$
        end if
    end for
    $right \leftarrow right - 1$
    for $i = right$ down to $left + 1$ do
        if $A[i] < A[i-1]$ then
            swap $A[i]$ and $A[i-1]$
        end if
    end for
    $left \leftarrow left + 1$
end while
```

3.11.3 Analyze

- **Best case:** O(n), sorted datasets.
- **Worst case:** O(n^2), reverse datasets.
- **Average case:** O(n^2)
- **Stability:** stable
- **Space Complexity:** O(1)

[7]

3.11.4 Strength

- Simple to implement.
- Improves over Bubble Sort by sorting in both directions.

[7]

3.11.5 Weakness

- Inefficient for large datasets due to quadratic time complexity.
- More passes through the array compared to Bubble Sort.

[7]

3.12 Flash Sort

3.12.1 Ideas

Flash Sort classifies elements into different classes and then permutes elements to their appropriate class before sorting each class individually. [9]

3.12.2 How it works

Step 1: Classify elements into classes.

Step 2: Permute elements to their appropriate class.

Step 3: Sort each class individually.

[9]

Pseudo Code:

Algorithm 17 Flash Sort

```
Require: An array of $n$ elements $A = [a_1, a_2, \ldots, a_n]$
Ensure: A sorted array in ascending order
1: Find the minimum and maximum elements of the array
2: Create and initialize the classification array $L$
3: Classify the elements of the array into $m$ classes
4: Permute the elements into their respective classes
5: for each class $i$ from 1 to $m$ do
6:     Sort the elements of the class using insertion sort
7: end for
```

[9]

3.12.3 Analyze

- **Best case:** O(n), elements are uniformly distributed.
- **Worst case:** O(n^2), highly non-uniform distribution
- **Average case:** O($n \log n$)
- **Stability:** not stable
- **Space complexity:** O(1)

3.12.4 Strength

- Very fast for large datasets with a uniform distribution.
- Linear best-case time complexity.

[30]

3.12.5 Weakness

- Performance degrades with highly non-uniform distributions.
- Not stable.

[30]

4 Experiment Result

4.1 Random data

4.1.1 Time

Table 1: Performance on random data ranging from 10,000 to 50,000 elements

Resulting Statistics	**Running Time (microseconds)**		
	Data Size 10,000	**Data Size 30,000**	**Data Size 50,000**
Selection Sort	97,469	903,949	2,552,549
Insertion Sort	57,545	535,625	437,898
Shell Sort	1,633	5,703	10,573
Bubble Sort	253,452	2,549,868	7,124,671
Heap Sort	1,555	6,419	9,954
Merge Sort	1,422	4,756	8,124
Quick Sort	948	3,190	5,529
Counting Sort	171	436	713
Binary Insertion Sort	34,104	309,756	869,336
Flash Sort	69	923	1,232
Shaker Sort	193,362	1,901,490	5,172,127
Radix Sort	539	1,451	2,880

Table 2: Performance on random data ranging from 100,000 to 500,000 elements

Resulting Statistics	**Running Time (microseconds)**		
	Data Size 100,000	**Data Size 300,000**	**Data Size 500,000**
Selection Sort	10,390,218	87,116,327	324,946,255
Insertion Sort	5,778,073	68,258,758	173,367,348
Shell Sort	22,566	80,316	138,039
Bubble Sort	29,059,046	264,115,718	1,065,304,439
Heap Sort	21,244	71,908	135,721
Merge Sort	17,167	53,697	94,445
Quick Sort	11,539	35,906	63,580
Counting Sort	1,133	4,155	7,723
Binary Insertion Sort	3,496,948	44,174,890	111,505,830
Flash Sort	2,358	13,058	25,403
Shaker Sort	24,467,417	185,107,319	655,214,789
Radix Sort	5,113	15,186	23,085

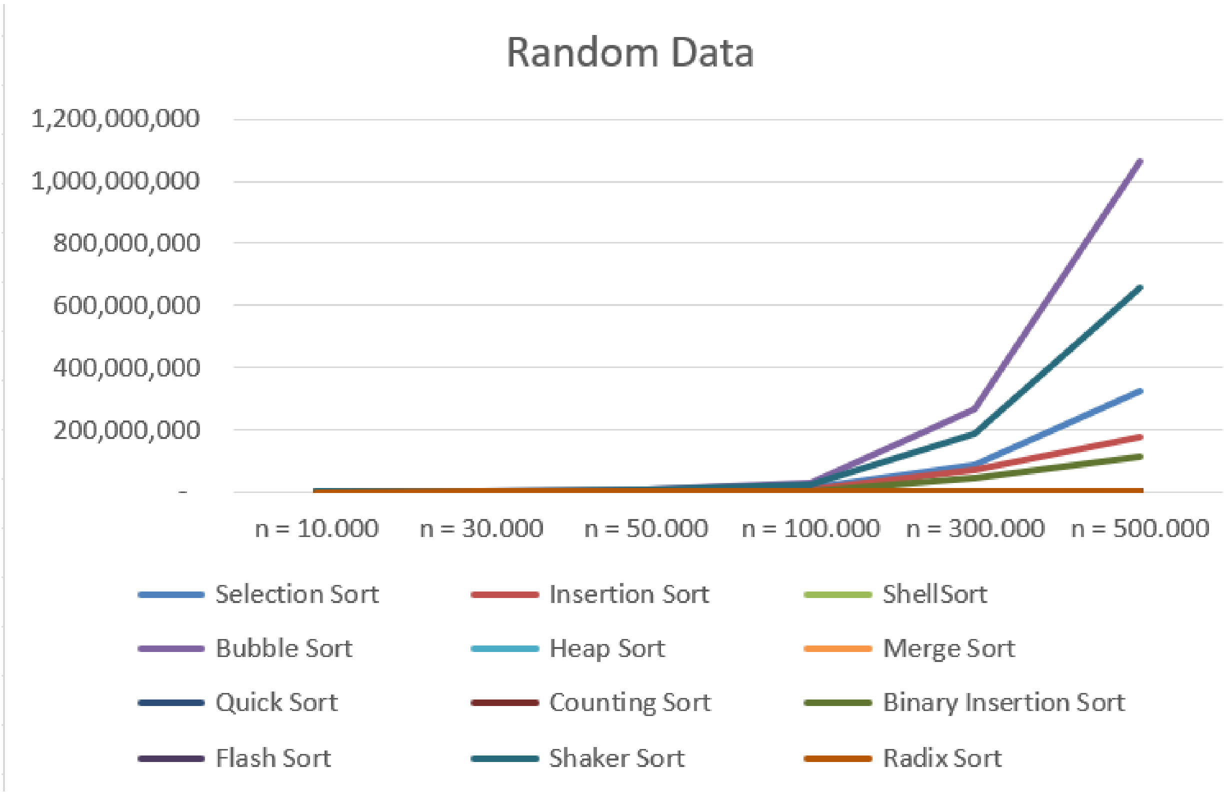

Figure 3: Running time analysis on random data ranging from 10,000 to 500,000 elements

Comments on running times

- The Counting Sort algorithm has the fastest runtime at all data sizes. This is explained by the linear cost of the algorithm, with the input data ensuring two properties: integer values and a not too large range of values. Therefore, Counting Sort has the fastest execution speed.
- For small data sets ($n \leq 30,000$), the runtime of the algorithms does not differ significantly.
- For large data sets, the algorithms can be divided into two distinct groups based on their runtime.
  . The slow group (Selection Sort, Insertion Sort, Bubble Sort, Binary Insertion Sort, Shaker Sort) has a relatively long runtime when processing random data.
  . The fast group (Shell Sort, Heap Sort, Merge Sort, Quick Sort, Counting Sort, Radix Sort, Flash Sort) shows almost no difference in runtime among these 7 algorithms with data sets of 500,000 elements. Thus, the graph shows their run times close to the horizontal axis.
- For the slow group of algorithms:

. Bubble Sort and Shaker Sort (an improved version of Bubble Sort) have the longest run times. Although Shaker Sort has some improvements, with random data, these improvements do not significantly enhance speed.

. Selection Sort and Insertion Sort can improve on the two algorithms mentioned above, but they are still quite slow compared to the algorithms in the O(n) and O($n \log n$) groups.

- For the relatively fast group of algorithms:

. In this group, Counting Sort runs the fastest with random data due to using space to achieve the best time.

. Quick Sort, Flash Sort, and Radix Sort also have very fast run times. For Quick Sort, choosing the median pivot tends to work better for all types of data compared to other pivot choices. Flash Sort performs well if the data is uniformly distributed. Radix Sort has good runtime due to handling integer and relatively small data. In terms of speed, these three algorithms have relatively equal run times for small data sets. For larger data sets, Quick Sort becomes more versatile.

. Finally, Merge Sort, Shell Sort, and Heap Sort have relatively long run times in this group. Shell Sort has an average cost of O($n \log n$) but can lean towards O(n^2) in cases of poor gap sequence choices. Merge Sort always guarantees a time cost of O($n \log n$), regardless of data distribution, and does not require gap sequence choices like Shell Sort. However, Merge Sort incurs significant memory overhead due to copying elements back and forth, making it slower compared to the O($n \log n$) group.

4.1.2 Comparison

Table 3: Performance on random data ranging from 10,000 to 50,000 elements

Resulting Statistics	**Number of Comparisons**		
	Data Size 10,000	**Data Size 30,000**	**Data Size 50,000**
Selection Sort	100,019,998	900,059,998	2,500,099,998
Insertion Sort	49,728,989	451,071,781	1,245,267,314
Shell Sort	648,946	2,381,758	443,812
Bubble Sort	100,001,505	900,048,120	2,500,066,145
Heap Sort	629,598	2,125,395	3,731,889
Merge Sort	712,921	2,374,273	414,611
Quick Sort	169,830	575,395	994,090
Counting Sort	60,000	180,001	282,771
Binary Insertion Sort	25,310,598	225,736,047	626,759,214
Flash Sort	91,849	293,385	450,815
Shaker Sort	75,204,621	678,511,429	1,884,389,845
Radix Sort	100,054	360,067	600,067

Table 4: Performance on random data ranging from 100,000 to 500,000 elements

Resulting Statistics	**Number of Comparisons**		
	Data Size 100,000	**Data Size 300,000**	**Data Size 500,000**
Selection Sort	10,000,199,998	90,000,599,998	250,000,999,998
Insertion Sort	4,996,938,229	45,028,840,865	124,865,837,842
Shell Sort	10,502,430	35,005,116	64,380,260
Bubble Sort	10,000,188,120	90,000,352,992	250,000,279,200
Heap Sort	7,964,927	26,246,534	45,565,203
Merge Sort	8,789,271	28,734,015	49,706,029
Quick Sort	2,200,188	8,090,115	14,960,770
Counting Sort	532,771	1,532,771	2,532,771
Binary Insertion Sort	2,505,622,614	22,502,429,202	62,638,204,244
Flash Sort	845,951	2,577,281	4,312,058
Shaker Sort	7,495,347,745	67,568,198,699	187,478,248,021
Radix Sort	1,200,067	3,600,067	6,000,067

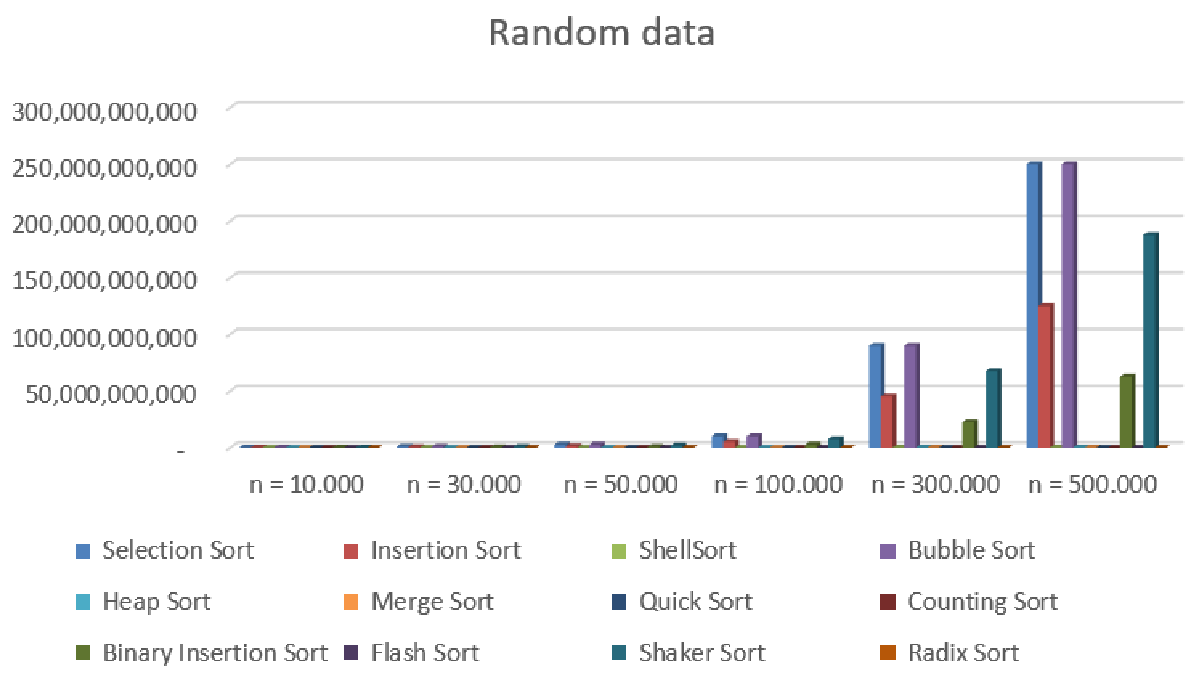

Figure 4: Comparison analysis on random data ranging from 10,000 to 500,000 elements

Comment on comparison

- In random data, Insertion Sort typically requires fewer comparisons than Bubble Sort and Selection Sort by approximately half. This is because Insertion Sort efficiently finds the correct position to insert each element into the already sorted portion of the array, thereby reducing the total number of comparisons needed.

- Binary Insertion Sort reduces the number of operations compared to Insertion Sort by half, achieved through more efficient element insertion position finding, O($\log n$) instead of O(n).
- The three algorithms with the fewest comparisons are Counting Sort, followed by Flash Sort and Radix Sort. However, both Flash Sort and Radix Sort are sensitive to the values present in the array. If the array has an uneven distribution of values or many identical elements, flash sort can require a high number of comparisons. Similarly, Radix Sort's comparison count increases significantly with more digits in the elements of the array. We see that as the data size increases, the ratio of Counting Sort and Radix Sort increases.
- Of the four sort functions with an average number of comparisons: Heap Sort, Shell Sort, Quick Sort, and Merge Sort, Quick Sort has the least number of comparisons. Due to Merge Sort and Heap Sort, multiple comparisons are used to merge and create the heap

4.2 Sorted data

4.2.1 Time

Table 5: Performance on sorted data ranging from 10,000 to 50,000 elements

Resulting Statistics	Running Time (microseconds)		
	Data Size 10,000	**Data Size 30,000**	**Data Size 50,000**
Selection Sort	99,871	917,382	2,548,577
Insertion Sort	32	94	156
Shell Sort	393	1,386	2,443
Bubble Sort	19	56	92
Heap Sort	1,420	3,952	6,515
Merge Sort	979	3,064	5,236
Quick Sort	679	1,325	2,345
Counting Sort	155	487	710
Binary Insertion Sort	508	1,733	2,450
Flash Sort	268	784	1,376
Shaker Sort	30	60	107
Radix Sort	476	1,763	2,565

Table 6: Performance on sorted data ranging from 100,000 to 500,000 elements

Resulting Statistics	Running Time (microseconds)		
	Data Size 100,000	Data Size 300,000	Data Size 500,000
Selection Sort	9,940,800	116,976,442	248,214,363
Insertion Sort	465	945	1,689
Shell Sort	5,708	17,922	29,928
Bubble Sort	278	565	2,072
Heap Sort	13,932	46,968	99,356
Merge Sort	11,367	35,851	57,775
Quick Sort	5,360	15,715	26,061
Counting Sort	1,469	3,224	6,535
Binary Insertion Sort	6,429	18,793	34,209
Flash Sort	2,536	7,535	11,005
Shaker Sort	198	603	1,090
Radix Sort	5,181	18,454	30,571

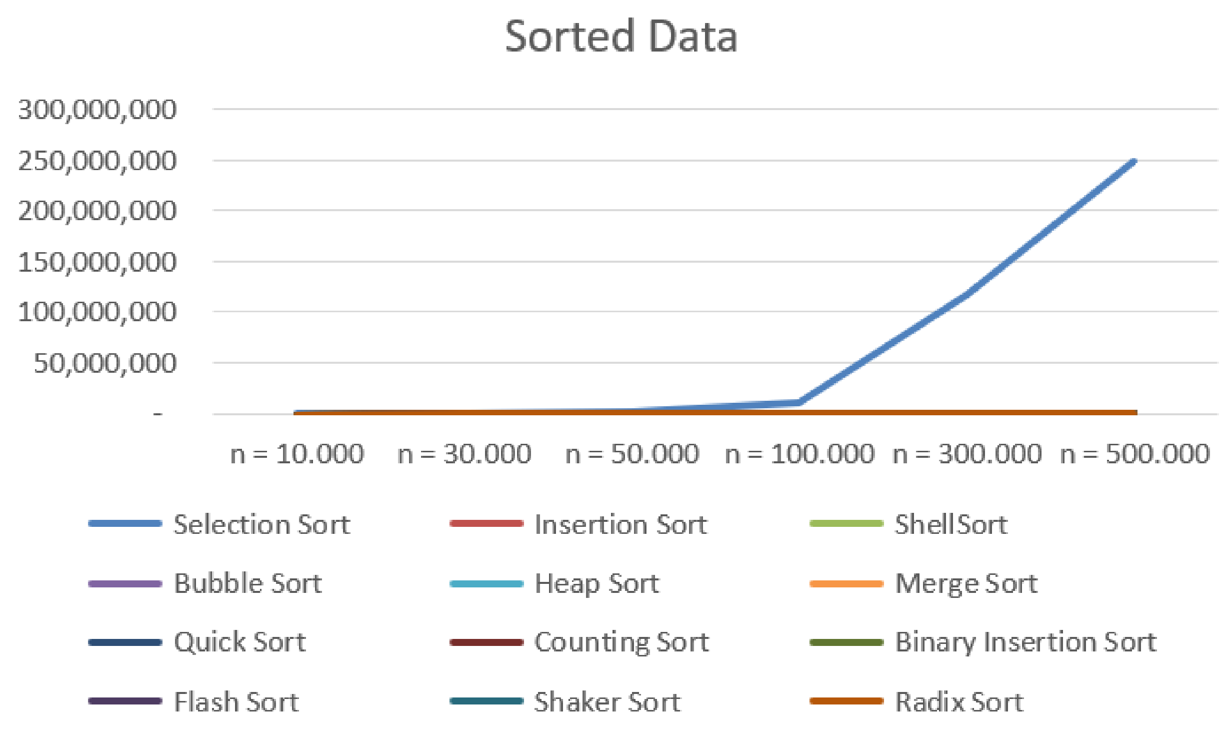

Figure 5: Running time analysis on sorted data ranging from 10,000 to 500,000 elements

Comments on running times

- The Bubble Sort algorithm benefits from already sorted data. It recognizes when the array is sorted and stops, leading to a best-case time complexity of O(n)
- For small data sets (n $\leq$ 30,000), the runtime of the algorithms does not differ significantly.
- For pre-sorted structures, the runtime for Selection Sort and Bubble Sort remains the largest. This reveals a drawback of these two algorithms: they do not recognize already sorted elements to stop the algorithm early. Although this is the best case, the complexity is still O(n^2).
- Insertion Sort, Binary Insertion Sort, and Shaker Sort have mechanisms to stop the algorithm early in the best case. The complexity in this case is O(n) for Insertion Sort and Shaker Sort, while Binary Insertion Sort has a complexity of O($n \log n$) due to the inefficiency of binary search (increasing from 1 to log n). This can be improved by adding a comparison check between a[i] and a[i-1] before performing the binary search for the insertion position of a[i].
- For the O(n) and O($n \log n$) group of algorithms:
  . Counting Sort is still the fastest algorithm in this group, as it optimally uses space to execute the algorithm and has relatively impressive runtime.
  . Merge Sort is quite slow on pre-sorted data because it still takes time and space to split and copy elements into other arrays and then merge them back.
  . Quick Sort and Flash Sort run very stably and are almost as fast as Counting Sort. Flash Sort is slightly faster than Quick Sort, possibly due to the choice of pivot. If the pivot in Quick Sort is the median element, the runtime is shorter, making it the best case and nearly as fast as Counting Sort.

4.2.2 Comparison

Table 7: Performance on sorted data ranging from 10,000 to 50,000 elements

Resulting Statistics	**Number of Comparisons**		
	Data Size 10,000	**Data Size 30,000**	**Data Size 50,000**
Selection Sort	100,019,998	900,059,998	2,500,099,998
Insertion Sort	29,998	89,998	149,998
Shell Sort	360,042	1,170,050	2,100,049
Bubble Sort	20,001	60,001	100,001
Heap Sort	655,992	2,197,817	3,859,218
Merge Sort	728,075	2,416,155	4,222,315
Quick Sort	149,056	485,547	881,084
Counting Sort	60,003	180,003	3,000
Binary Insertion Sort	370,852	1,251,700	2,203,396
Flash Sort	118,993	356,993	594,993
Shaker Sort	20,001	60,001	100,001
Radix Sort	100,054	360,067	600,067

Table 8: Performance on sorted data ranging from 100,000 to 500,000 elements

Resulting Statistics	**Number of Comparisons**		
	Data Size 100,000	**Data Size 300,000**	**Data Size 500,000**
Selection Sort	10,000,199,998	90,000,599,998	250,000,999,998
Insertion Sort	299,998	899,998	1,499,998
Shell Sort	4,500,051	15,300,061	25,500,058
Bubble Sort	200,001	600,001	1,000,001
Heap Sort	8,226,253	27,022,363	46,789,063
Merge Sort	8,944,635	29,178,555	50,378,555
Quick Sort	1,862,157	5,889,301	10,048,591
Counting Sort	600,003	1,800,003	3,000,003
Binary Insertion Sort	4,706,788	15,527,140	26,927,140
Flash Sort	1,189,993	3,569,993	5,949,993
Shaker Sort	200,001	600,001	100,000
Radix Sort	1,200,067	4,200,080	7,000,080

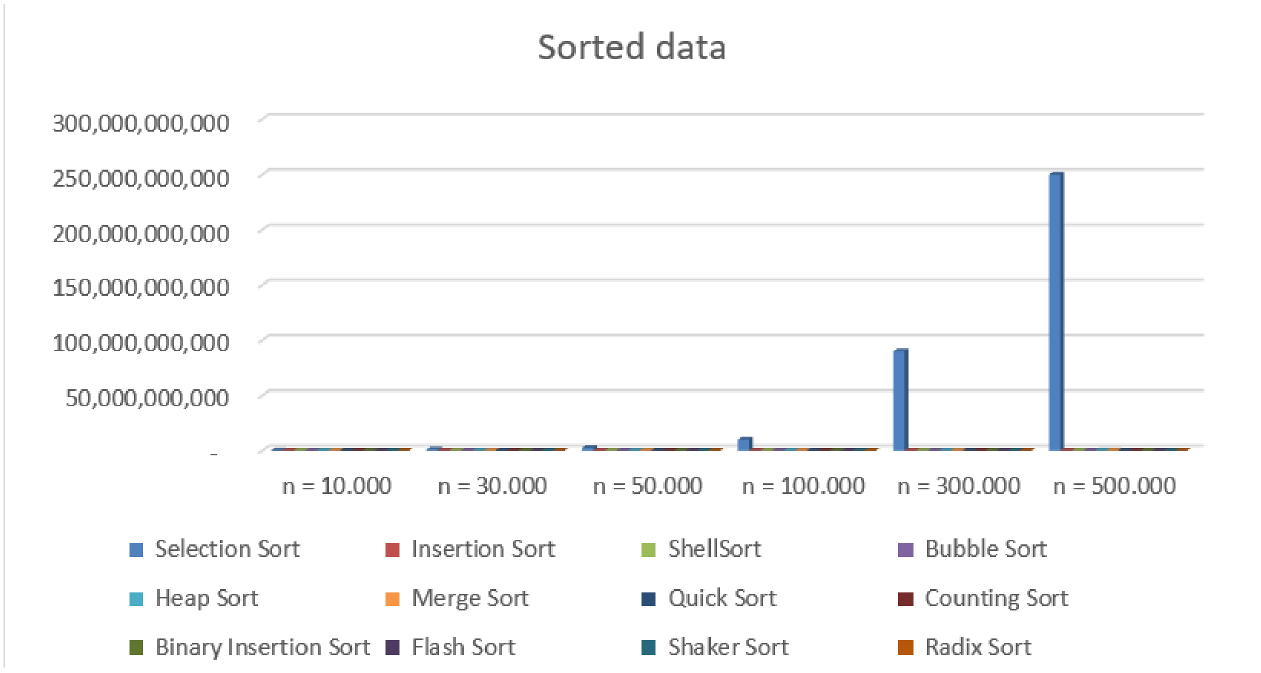

Figure 6: Comparison analysis on sorted data ranging from 10,000 to 500,000 elements

- In an already sorted array, Selection Sort has the highest number of comparisons, as it always scans the entire array, resulting in $n \cdot \frac{n-1}{2}$ comparisons regardless of the data's initial order.
- Bubble Sort, Shaker Sort, and Insertion Sort each have at least O(n) comparisons in their best case, which is an already sorted array. However, Insertion Sort has slightly more comparisons due to the inner loop placing a guard at the array's end.
- Counting Sort, though still O(n), doesn't have the lowest number of comparisons due to its multiple loops for initializing a new array, increasing the comparisons.
- When the pivot is chosen as the median, Quick Sort performs exceptionally well on sorted data, outperforming both Merge Sort and Heap Sort.
- In a sorted array, Binary Insertion Sort has more comparisons than Insertion Sort because it performs a log(n) binary search for each element, whereas Insertion Sort only needs one comparison per correctly placed element.
- Shell sort performs well on sorted data, running faster than merge and Heap Sort, because its best case comparisons are $O(n \log n)$. In contrast, Heap Sort and Merge Sort take time to merge and create the heap.

4.3 Nearly Sorted data

4.3.1 Time

Table 9: Performance on nearly sorted data ranging from 10,000 to 50,000 elements

Resulting Statistics	**Running Time (microseconds)**		
	Data Size 10,000	**Data Size 30,000**	**Data Size 50,000**
Selection Sort	106,162	1,082,569	2,525,505
Insertion Sort	298	591	741
Shell Sort	634	1,920	3,076
Bubble Sort	85,855	823,511	1,694,552
Heap Sort	1,528	4,306	6,965
Merge Sort	1,155	3,639	5,535
Quick Sort	2,260	21,292	41,522
Counting Sort	191	451	705
Binary Insertion Sort	918	2,296	3,471
Flash Sort	295	735	1,213
Shaker Sort	641	1,369	2,218
Radix Sort	815	1,535	2,624

Table 10: Performance on nearly sorted data for size from 100,000 to 500,000 elements

Resulting Statistics	**Running Time (microseconds)**		
	Data Size 100,000	**Data Size 300,000**	**Data Size 500,000**
Selection Sort	10,171,150	90,121,396	252,510,248
Insertion Sort	995	1,562	2,619
Shell Sort	5,625	17,786	30,670
Bubble Sort	3,388,569	14,923,653	26,772,562
Heap Sort	15,828	44,797	84,221
Merge Sort	12,435	37,742	67,252
Quick Sort	72,911	39,751	56,774
Counting Sort	1,548	3,838	7,384
Binary Insertion Sort	7,511	22,114	36,603
Flash Sort	2,579	7,202	11,932
Shaker Sort	3,948	7,537	13,366
Radix Sort	5,089	19,476	32,618

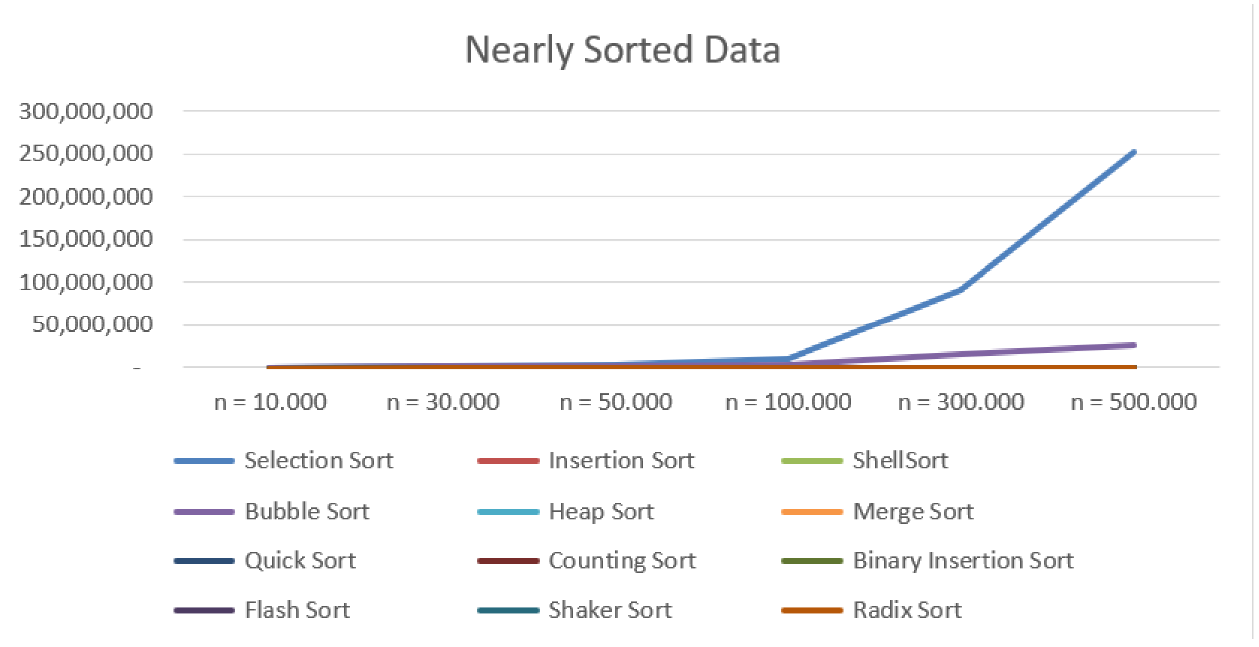

Figure 7: Running time analysis on nearly sorted data ranging from 10,000 to 500,000 elements

Comments on running times

- Based on the chart, the two slowest algorithms are Bubble Sort and Shaker Sort. Part of this is due to the algorithms not recognizing that the array is almost sorted, requiring them to run from start to finish, which consumes a significant amount of time.
- Although Counting Sort has a complexity of O($n \log n$), it is still the best-performing algorithm for this type of data, as explained in the previous sections.
- In general, aside from Bubble Sort, Shaker Sort, and Counting Sort, the remaining algorithms have relatively similar run times, with negligible differences.
- For nearly sorted data, some algorithms benefit from this type of data: Insertion Sort, Binary Insertion Sort, Shaker Sort (Cocktail Sort), and optimized Bubble Sort.
- Algorithms that have certain improvements with nearly sorted data include Shell Sort and Quick Sort.
- Algorithms like Heap Sort and Merge Sort benefit little or not at all from nearly sorted data.

4.3.2 Comparison

Table 11: Performance on nearly sorted data ranging from 10,000 to 50,000 elements

Sorting Algorithm	**Number of Comparisons**		
	Data Size 10,000	**Data Size 30,000**	**Data Size 50,000**
Selection Sort	100,019,998	90,005,999	2,500,099,998
Insertion Sort	179,406	359,034	523,294
Shell Sort	397,017	1,288,350	2,229,824
Bubble Sort	100,010,001	90,003,000	2,500,050,001
Heap Sort	655,703	2,197,842	385,937
Merge Sort	727,971	2,416,023	4,222,177
Quick Sort	1,546,741	16,772,680	26,966,250
Counting Sort	60,003	180,003	300,003
Binary Insertion Sort	446,448	1,404,272	240,082
Flash Sort	118,966	356,967	594,970
Shaker Sort	299,791	899,791	1,499,791
Radix Sort	100,054	360,067	600,067

Table 12: Performance on reverse data ranging from 100,000 to 500,000 elements

Sorting Algorithm	**Number of Comparisons**		
	Data Size 100,000	**Data Size 300,000**	**Data Size 500,000**
Selection Sort	10,000,199,998	90,000,599,998	250,000,999,998
Insertion Sort	726,542	1,203,434	2,026,110
Shell Sort	4,682,533	15,428,932	25,683,026
Bubble Sort	10,000,100,001	90,000,300,001	250,000,500,001
Heap Sort	8,226,128	27,022,601	46,789,040
Merge Sort	8,944,507	29,178,431	50,378,433
Quick Sort	38,140,212	42,539,247	51,870,746
Counting Sort	600,003	1,800,003	3,000,003
Binary Insertion Sort	4,972,294	15,688,208	27,102,319
Flash Sort	1,189,968	3,569,972	5,949,969
Shaker Sort	2,999,791	6,599,891	10,999,891
Radix Sort	1,200,067	4,200,080	7,000,080

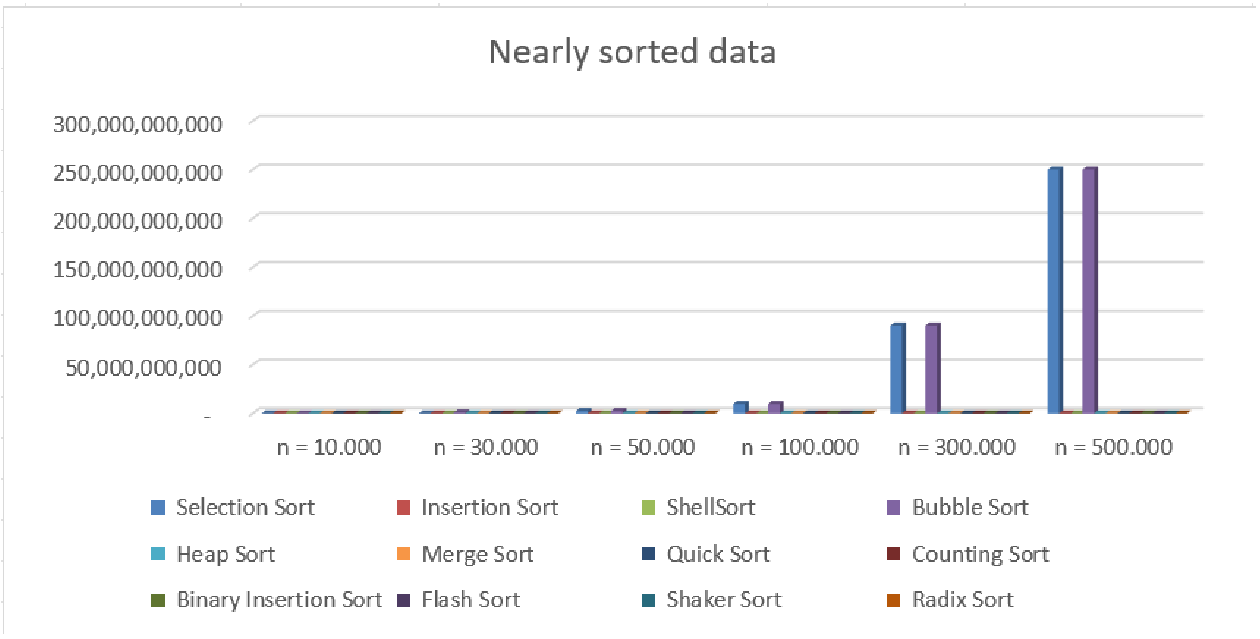

Figure 8: Comparison analysis on nearly sorted data ranging from 10,000 to 500,000 elements

Comments on comparison

- In nearly sorted data, Bubble Sort and Selection Sort show the worst performance, resulting in the largest number of comparisons for larger data sizes. This inefficiency demonstrates that these two algorithms are not suitable for nearly sorted data.
- The algorithms with the fewest comparisons are Counting Sort, Radix Sort, and Flash Sort.
- Heap Sort, Shell Sort, Merge Sort, and Quick Sort are among the 12 sorting functions analyzed, each with an average number of comparisons. Quick Sort stands out with fewer comparisons due to its median pivot selection strategy, which helps maintain efficiency in nearly sorted data. However, as data size increases, Merge Sort and Heap Sort demonstrate more stable performance characteristics. Shell Sort has the advantage of minimizing the number of comparisons due to its gap sequence strategy, which effectively reduces the distance between elements in the initial passes, leading to an almost sorted state as the gaps decrease.

4.4 Reverse data

4.4.1 Time

Table 13: Performance on reverse data ranging from 10,000 to 50,000 elements

Resulting Statistics	**Running Time (microseconds)**		
	Data Size 10,000	**Data Size 30,000**	**Data Size 50,000**
Selection Sort	98,511	909,141	2,550,334
Insertion Sort	115,874	1,168,127	2,959,511
Shell Sort	886	2,165	4,045
Bubble Sort	224,124	1,985,740	5,728,134
Heap Sort	1,222	3,741	6,247
Merge Sort	853	3,278	4,476
Quick Sort	735	2,568	4,306
Counting Sort	170	448	758
Binary Insertion Sort	66,890	668,357	1,688,383
Flash Sort	252	755	1,236
Shaker Sort	221,203	975,112	5,391,704
Radix Sort	619	1,588	2,366

Table 14: Performance on reverse data ranging from 100,000 to 500,000 elements

Resulting Statistics	**Running Time (microseconds)**		
	Data Size 100,000	**Data Size 300,000**	**Data Size 500,000**
Selection Sort	10,369,623	91,021,040	255,822,494
Insertion Sort	12,001,763	109,018,229	309,278,293
Shell Sort	7,629	23,662	45,706
Bubble Sort	24,733,522	219,473,736	615,799,750
Heap Sort	13,453	44,022	77,639
Merge Sort	9,407	28,597	48,975
Quick Sort	9,387	30,588	53,487
Counting Sort	1,298	3,065	6,101
Binary Insertion Sort	6,806,304	68,605,091	247,994,333
Flash Sort	2,250	7,170	1,120
Shaker Sort	22,577,121	212,500,174	566,291,392
Radix Sort	7,187	19,378	29,125

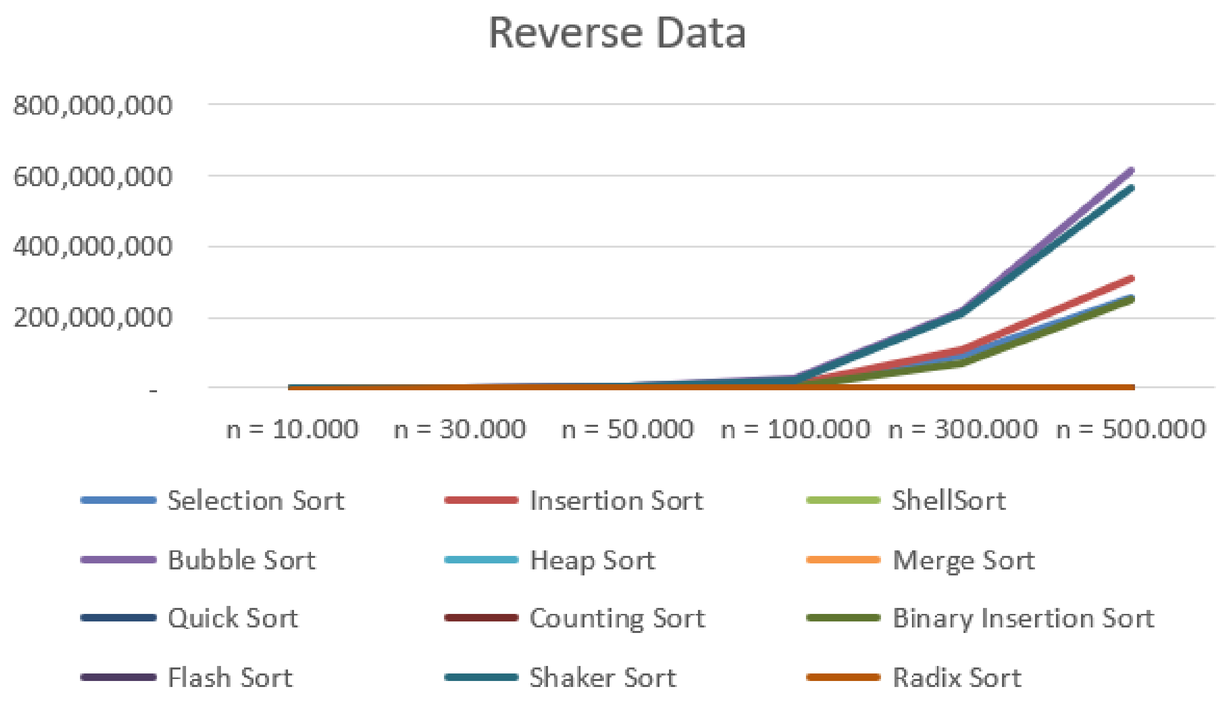

Figure 9: Running time analaysis on reverse data ranging from 10,000 to 500,000 elements

Comments on running times

Analysis of Sorting Algorithms with Reverse Ordered Data

For data that has been sorted in reverse order, there are some changes in the relative speeds of the algorithms:

Slow Algorithms

- **Bubble Sort**: This algorithm remains slow.
- **Shaker Sort**: Shaker Sort becomes the slowest algorithm due to the inefficiency of checking and marking the last swapped position, making it slower than Bubble Sort.

O(n^2) Algorithms

- **Binary Insertion Sort**: Among the O(n^2) algorithms, Binary Insertion Sort has the shortest runtime. The binary search operation proves to be quite effective when the data is in reverse order, making the runtime of Binary Insertion Sort significantly better than the original Insertion Sort.

Algorithms with Average Complexity O(n) and O($n \log n$)

- **Fastest Algorithms**: Counting Sort, Quick Sort, and Flash Sort are the three fastest algorithms in this category.

- **Counting Sort**: This algorithm is always fast with a stable O(n) time complexity for integer data types and when the range of values is not too large.
- **Quick Sort**: For data sorted in ascending or descending order, Quick Sort encounters the best case scenario, selecting a median pivot element for subarrays, thus having a faster runtime compared to Flash Sort.
- **Merge Sort**: Merge Sort is relatively slow due to the top-down recursive calls and the process of copying memory regions back and forth during the execution of the algorithm.

4.4.2 Comparison

Table 15: Performance on reverse data ranging from 10,000 to 50,000 elemen

Sorting Algorithm	**Number of Comparisons**		
	Data Size 10,000	**Data Size 30,000**	**Data Size 50,000**
Selection Sort	100,019,998	900,059,998	2,500,099,998
Insertion Sort	100,009,999	900,029,999	2,500,049,999
Shell Sort	475,175	1,554,051	2,844,628
Bubble Sort	100,019,998	900,059,998	2,500,099,998
Heap Sort	605,098	2,055,681	3,606,091
Merge Sort	590,059	1,960,699	3,418,411
Quick Sort	251,262	853,937	1,516,333
Counting Sort	60,003	180,003	300,003
Binary Insertion Sort	50,348,179	451,187,593	1,252,080,140
Flash Sort	103,751	311,251	51,875
Shaker Sort	100,010,001	900,030,001	2,500,050,001
Radix Sort	100,054	360,067	600,067

Table 16: Performance on reverse data ranging from 100,000 to 500,000 elements

Sorting Algorithm	**Number of Comparisons**		
	Data Size 100,000	**Data Size 300,000**	**Data Size 500,000**
Selection Sort	10,000,199,998	90,000,599,998	250,000,999,998
Insertion Sort	10,000,099,999	90,000,299,999	250,000,499,999
Shell Sort	6,089,190	20,001,852	33,857,581
Bubble Sort	10,000,199,998	90,000,599,998	250,009,999,998
Heap Sort	7,700,828	25,507,884	44,375,011
Merge Sort	7,236,827	23,584,027	40,812,123
Quick Sort	3,273,652	10,945,538	19,183,503
Counting Sort	600,003	1,800,003	3,000,003
Binary Insertion Sort	5,004,460,231	45,014,870,410	125,025,890,765
Flash Sort	1,037,501	3,112,501	5,187,501
Shaker Sort	10,000,100,001	90,000,300,001	250,000,500,001
Radix Sort	1,200,067	4,200,080	7,000,080

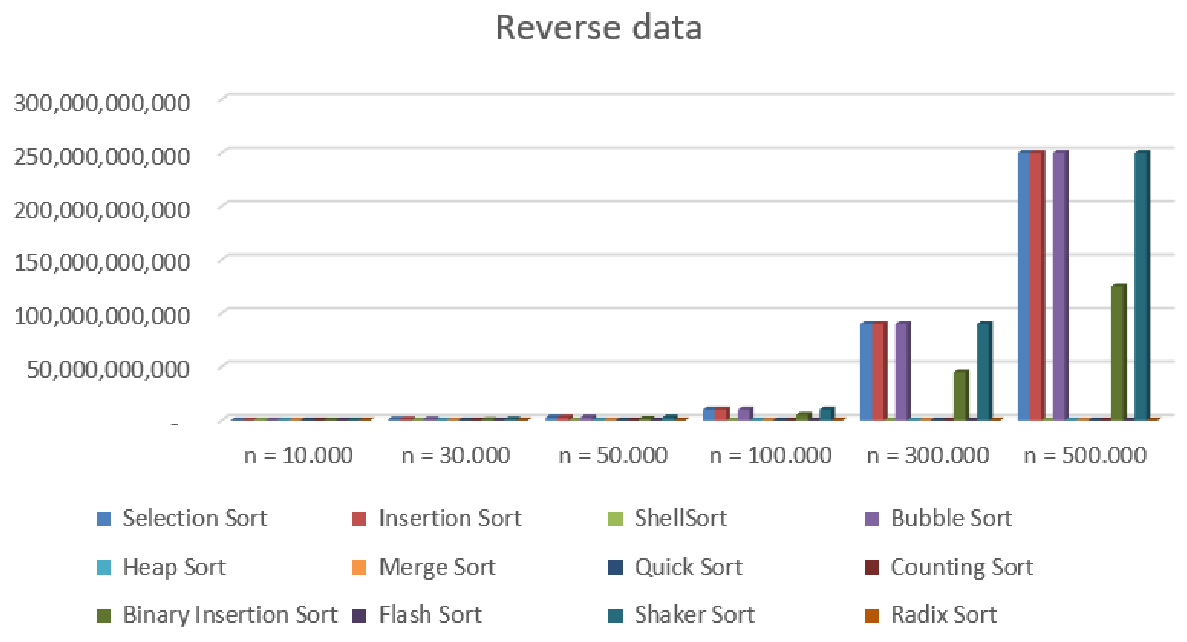

Figure 10: Comparison analysis on reverse data ranging from 10,000 to 500,000 elements

Comments on comparison

- In the reversed array data type, Insertion Sort, Selection Sort, Bubble Sort, and Shaker Sort algorithms all exhibit their worst-case performance, resulting in the largest number of comparisons, which is $n \cdot \frac{n-1}{n}$ for all data sizes. This inefficiency demonstrates that these four algorithms are not suitable for this type of data.

- The three algorithms with the fewest comparisons are Counting Sort, followed by Flash Sort and Radix Sort.

- Heap Sort, Shell Sort, Merge Sort, and Quick Sort are among the 12 sorting functions analyzed, each with an average number of comparisons. Quick Sort stands out with fewer comparisons due to its median pivot selection strategy, which helps mitigate issues with reversed arrays. However, as data size increases, Merge Sort and Heap Sort demonstrate more stable performance characteristics. Shell Sort has the advantage of minimizing the number of comparisons in 2 out of the 4 sorting functions. This efficiency stems from its initial passes where elements at far distances are positioned closer to their correct places, gradually approaching an almost sorted state as the gaps decrease.

5 Project Details

5.1 Project structures

- Main.cpp: Run the program
- Header.h: Declare all functions in ShowName.cpp, SortFunction.cpp and DataGenerator.cpp
- ShowName.cpp: Function to find the sorting algorithm and type of dataset of input
- SortFunction.cpp: Define all sorting function
- DataGenerator.cpp: Generate data for sorting.
- input.txt: Generated input for command 2 and command 5.
- input1.txt: Generated input of random order data for command 3.
- input2.txt: Generated input of nearly sorted data for command 3.
- input3.txt: Generated input of sorted data for command 3.
- input4.txt: Generated input of reversed data for command 3.

Note: File .txt format:

- First line: an integer n, number of elements in input.
- Second line: n integers number, separated by space.

5.2 Programming notes

Chrono Library

- The chrono library, part of the C++ Standard Library, facilitates time-related operations. It equips programmers with tools to measure code execution, introduce delays, and manage time units like seconds, minutes, hours, and days.
- The `std::chrono::high_resolution_clock::now()` function in C++ retrieves the current high-resolution time point from the system clock. It returns a `time_point` representing the current time with the highest possible system-specific precision, typically measured in nanoseconds or finer units depending on the system.
- This line of code calculates the elapsed time between two time points `end` and `start` in seconds with high precision.

Code implementations Some functions in file Main.cpp are adapted from this reference [11].

6 Conclusion

- Selection Sort, Bubble Sort, and Insertion Sort all have $O(n^2)$ time complexity, making them slow for large datasets. Insertion Sort performs relatively better for small arrays. Among these, Selection Sort is the slowest due to its $O(n^2)$ time complexity, which remains unchanged regardless of the input array's order.
- If the input array is in random order, the best algorithms to use are the ones with the time complexity of O($n \log n$) and O(n), like Counting Sort or Merge Sort.
- The Quick Sort and Merge Sort are efficient recursive sorting algorithms. Quick Sort is typically one of the fastest sorting algorithms, but its worst-case performance is slower than Merge Sort's, although this worst case is rare. Merge Sort offers consistently good performance but requires extra storage equal to the array size. On average, Quick Sort performs roughly $1.39.n \log n$ comparisons. Despite this, Quick Sort is often faster in practice due to less data movement. [21]
- When the order of the input array is nearly sorted, it's best to use Insertion Sort or Shaker Sort, because both algorithms stop immediately after the array is sorted.
- For an already sorted array, both shaker sort with Insertion Sort and Bubble Sort work well. Bubble sort, in particular, has a best-case time complexity of O(n), making it just as effective in this situation.
- If the array is sorted in reverse order, we should use algorithms that perform well on average, such as Counting Sort, Radix Sort, and Quick sort.

References

[1] *Advanced Topics in Sorting*. Accessed: 2024-06-27, Slide. URL: https://www.cs.princeton.edu/courses/archive/spr07/cos226/lectures/05AdvancedSorts.pdf.

[2] *Applications, Advantages and Disadvantages of Radix Sort*. Accessed: 2024-06-29. URL: https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-radix-sort/.

[3] *Binary Insertion Sort*. Accessed: 2024-06-28. URL: https://www.geeksforgeeks.org/binary-insertion-sort/.

[4] *Binary Insertion Sort*. Accessed: 2024-06-29. URL: https://www.interviewkickstart.com/blogs/learn/binary-insertion-sort.

[5] *Bubble Sort Algorithm*. Accessed: 2024-06-27. URL: https://www.geeksforgeeks.org/bubble-sort-algorithm/.

[6] Frank M. Carrano. *Data-Abstraction-Problem-Solving-with-C-Walls-and-Mirrors*. Seventh Edition.

[7] *Cocktail Sort*. Accessed: 2024-06-28. URL: https://www.geeksforgeeks.org/cocktail-sort/.

[8] *Counting Sort – Data Structures and Algorithms Tutorials*. Accessed: 2024-06-28. URL: https://www.geeksforgeeks.org/counting-sort/.

[9] *Flash Sort*. Accessed: 2024-06-28. URL: https://www.neubert.net/FSOIntro.html.

[10] *Gap Sequences*. Accessed: 2024-06-28. URL: https://en.wikipedia.org/wiki/Shellsort#Gap_sequences/.

[11] HaiDuc0147. *sortingAlgorithm*. Accessed: 2024-06-30, Code. URL: https://github.com/HaiDuc0147/sortingAlgorithm/tree/main/reportSort?fbclid=IwZXh0bgNhZW0CMTEAAR2Bt
9gHOTGWhqzKrQcS0FD9qmBJnAk4tRLZPked2PMhiK9WMBeUKp2adE_aem_Nk85fYMHlyrp4hXbShdXAw.

[12] *Heap Sort – Data Structures and Algorithms Tutorials*. Accessed: 2024-06-28. URL: https://www.geeksforgeeks.org/heap-sort/?ref=shm.

[13] *Heap Sort Code in C: What is and Program*. Accessed: 2024-06-28. URL: https://www.shiksha.com/online-courses/articles/all-about-heap-sort-technique/#:~:text=Advantages%20of%20Heap%20Sort%3A,efficient%20for%20memory%2Dconstrained%20environments.

[14] *Insertion Sorting Algorithm*. Accessed: 2024-06-27. URL: https://www.geeksforgeeks.org/insertion-sort-algorithm/.

[15] *Introduction to Smooth Sort*. Accessed: 2024-06-29. URL: https://www.geeksforgeeks.org/introduction-to-smooth-sort/.

[16] *Merge Sort – Data Structure and Algorithms Tutorials*. Accessed: 2024-06-28. URL: https://www.geeksforgeeks.org/merge-sort/?ref=shm.

[17] *Merge Sort Algorithm (With Code)*. Accessed: 2024-06-28. URL: https://www.shiksha.com/online-courses/articles/merge-sort-algorithm-with-code/.

[18] *MSD(Most Significant Digit) Radix Sort*. Accessed: 2024-06-29. URL: https://www.geeksforgeeks.org/msd-most-significant-digit-radix-sort/.

[19] Van Nam Chi. *DSA-02-Sorting Algorithm-End*. Accessed: 2024-06-28, Slide. URL: https://drive.google.com/drive/folders/1Qu8QS_AqxZypG%201HP4W6vROqLhriXMGT.

[20] *Quick Sort Algorithm*. Accessed: 2024-06-28. URL: https://www.geeksforgeeks.org/quick-sort-algorithm/.

[21] *Quick Sort Analysis*. Accessed: 2024-06-28. URL: https://math.oxford.emory.edu/site/cs171/quickSortAnalysis/.

[22] *Radix Sort – Data Structures and Algorithms Tutorials*. Accessed: 2024-06-28. URL: Radix%20Sort%20%E2%80%93%20Data%20Structures%20and%20Algorithms%20Tutorials.

[23] *Selection Sort Algorithm*. Accessed: 2024-06-29. URL: https://www.geeksforgeeks.org/selection-sort-algorithm-2/.

[24] *Shell Sort: Advantages and Disadvantages*. Accessed: 2024-06-28. URL: https://www.shiksha.com/online-courses/articles/shell-sort-advantage%20and-disadvantages/.

[25] *ShellSort*. Accessed: 2024-06-27. URL: https://www.geeksforgeeks.org/shellsort/.

[26] *Sorting Algorithm*. Accessed: 2024-06-27. URL: https://www.geeksforgeeks.org/sorting-algorithms/.

[27] *Sorting algos in C programming|Understand basics and More.*. Accessed: 2024-06-27. URL: https://www.equestionanswers.com/c/c-sorting.php.

[28] *Stable Sorting Algorithms*. Accessed: 2024-06-27. URL: https://www.baeldung.com/cs/stable-sorting-algorithms.

[29] *TimSort – Data Structures and Algorithms Tutorials*. Accessed: 2024-06-29. URL: https://www.geeksforgeeks.org/timsort/.

[30] *What is Flash Sort?* Accessed: 2024-06-28. URL: https://www.educative.io/answers/what-is-flash-sort.