UNIVERSITY OF SCIENCE - VIETNAM NATIONAL UNIVERSITY
FACULTY OF INFORMATION TECHNOLOGY

Quang - Thang Nguyen

ANALYZING SORTING ALGORITHMS AND COMPARE THEM

REPORT FOR LAB 3
DATA STRUCTURE AND ALGORITHM - 22TNT1TN

Ho Chi Minh City, 11/2023

UNIVERSITY OF SCIENCE - VIETNAM NATIONAL UNIVERSITY
FACULTY OF INFORMATION TECHNOLOGY

Quang - Thang Nguyen

ANALYZING SORTING ALGORITHMS AND COMPARE THEM

REPORT FOR LAB 3
DATA STRUCTURE AND ALGORITHM - 22TNT1TN

Lecturer
Mr. Huy-Thong Bui

Ho Chi Minh City, 11/2023

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In practical contexts, data sorting holds significant importance. Through sorting, information is organized systematically, enhancing the ease of search and management.

In tandem with the evolution of computer science, sorting algorithms have developed over time. Presently, there exists a multitude of sorting algorithms, each with its own merits and drawbacks. Within the scope of this report, I will conduct an investigation and implementation of 11 classical sorting algorithms, commonly taught to current computer science students.

Chapter 2

Information

Table 2.1: Computer information

Student name	Quang - Thang Nguyen
Processor	Intel(R) Core(TM) i7-10750H CPU
System type	64-bit operating system, x64-based processor
Installed RAM	8.00 GB (7.83 GB usable)
Main OS	Windows 11 Home Single Language
Main OS OS version	22H2
WSL version	2.0.9.0
WSL OS	Ubuntu 22.04.3 LTS
WSL compiler	g++ 11.4.0

Chapter 3

Algorithm presentation

3.1 Selection sort

3.1.1 Idea of algorithm

- This algorithm operates through the partitioning of an array into two segments: a sorted region and an unsorted region.
- Typically, the sorted region evolves incrementally from the left side of the array.
- The algorithm iterates through $n-1$ steps, during each of which it identifies the minimum value within the unsorted region and exchanges it with the leftmost unsorted element.
- Consequently, the demarcation between the sorted and unsorted regions is shifted one element to the right after each step.
- The algorithm concludes upon completing $n-1$ steps.

3.1.2 Pseudo code

```
Function selectionSort(arr)
    n ← length(arr) ;
    One by one, move boundary of unsorted region
    for i ← 0 to n - 2 do
        m ← 0 ;
        Find minimum of unsorted region
        for j ← i + 1 to n do
            if arr[j] < arr[m] then  m ← j ;
        end
        Bring minimum element to leftmost position of unsorted region
        if m != i then
            swap(arr[i], arr[m]) ;
        end
    end
end
```

Algorithm 1: Selection sort

3.1.3 Illustration

Assume this array needs to be sorted

$$arr = \{6, 3, 0, 5\}$$

The table shows how the algorithm sorts array

Table 3.1: Illustration for Selection sort

step	array	explain
0	{ | 6, 3, 1, 5 }	At first, sorted region has no element.
1	{ | 6, 3, **1**, 5 }	After first step, minimum element, **1**, is chosen. It will be brought to leftmost position of unsorted region, position 0.
2	{ 1, | 6, **3**, 5 }	Now the smallest unsorted element is **3**, who will be brought to position 1 on next step.
3	{ 1, 3, | 6, **5** }	Now the smallest unsorted element is **5**, who will be brought to position 2, leftmost position of unsorted region, on next step.
4	{ 1, 3, 5, | 6 }	Now the array is sorted. The algorithm does not work with last element since it is already at right position.

3.1.4 Complexity Analysis

Space

Since the algorithm does not requires extra arrays, space complexity is $\Theta(1)$ for all cases.

Time

- The for loop in line 4 has to execute $n-1$ times.
- The for loop in line 7 has to execute $n-1-i$ times.
- Therefore, the number of comparison operations needed is

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

- Both loops are independent from data distribution, so time complexity is always $\Theta(n^2)$.

3.1.5 Optimize

At each step, the minimum and maximum can be determined at the same time. Then the loop will not have to be executed $n-1$ times, but can be decreased approximately two times. Therefore running time will decrease about two times.

3.2 Insertion sort

3.2.1 Idea of algorithm

- Like Selection sort (3.1), insertion sort operates through the partitioning of an array into 2 segments: sorted region and an unsorted region.
- Typically, the sorted region evolves incrementally from the left side of the array
- The algorithm individually considers each element in an unsorted array and accurately places it within its appropriate position in the sorted region.
- This entails inserting the selected element into the sorted region by displacing all elements greater than the current one, creating a void that is subsequently filled by the chosen element.

3.2.2 Pseudo code

```
Function insertionSort(arr)
    n ← length(arr) ;
    for i ← 1 to n - 1 do
        selected ← arr[i] ;
        j ← i - 1 ;
        Find position to add selected element
        while j >= 0 && arr[j] > selected do
            arr[j + 1] ← arr[j] ;
            j - - ;
        end
        arr[j + 1] ← selected ;
    end
end
```

Algorithm 2: Insertion sort

3.2.3 Illustration

Assume this array needs to be sorted

$$arr = \{12, 11, 13, 5, 6\}$$

The table shows how algorithm sorts array

Table 3.2: Illustration for Insertion sort

step	**array**	**explain**
0	{ 12, | 11, 13, 5, 6 }	At first, sorted region has 1 element.
1	{ *, 12, | **11**, 13, 5, 6 }	**11** is chosen, and the suitable position for it is marked by *
2	{ 11, 12, *, | **13**, 5, 6 }	**13** is chosen, and the suitable position for it is marked by *
3	{ *, 11, 12, 13, | **5**, 6 }	**5** is chosen, and the suitable position for it is marked by *
4	{ 5, *, 11, 12, 13, | **6** }	**6** is chosen, and the suitable position for it is marked by *
5	{ 5, 6, 11, 12, 13, | }	The array is now sorted.

3.2.4 Complexity Analysis

Space

Since the algorithm does not requires extra arrays, space complexity is $\Theta(1)$ for all cases.

Time

- The for loop in line 3 has to execute $n - 1$ times.
- In best case, the while loop does not have to execute (in case of original array is already sorted). Then, the number of comparison operations needed is

$$(n - 1) * 1 = 1$$

- In worse case, the $while$ loop has to execute i times (when array is reversely sorted). Then, the number of comparison operations needed is

$$\sum_{1} n - 1 \,(i) = \frac{n(n-1)}{2}$$

- Therefore, in best case, time complexity is $\Omega(n)$, and in worse case, it is $O(n^2)$.

- Average time complexity is $O(n^2)$.

3.2.5 Optimize

In Insertion sort, finding the location to insert is sequential. The searching speed can be increased by using binary search.

3.3 Binart insertion sort

3.3.1 Pseudo code

```
Function binaryInsertionSort(arr)
    n ← length(arr) ;
    for i ← 1 to n - 1 do
        selected ← arr[i] ;
        j ← i - 1 ;
        Find position to insert using binary search
        loc ← binarySearch(arr, selected, 0, j) ;
        Move all elements after location to create space
        while j >= loc do
            arr[j + 1] ← arr[j] ;
            j - - ;
        end
        arr[j + 1] ← selected ;
    end
end
Function binarySearch(arr, key, low, high)
    if high <= low then
        return (key > arr[low]) ? (low + 1) : low ;
    end
    mid ← (low + high) / 2 ;
    if key == arr[mid] then
        return mid + 1 ;
    end
    if key > arr[mid] then
        return binarySearch(arr, key, mid + 1, high) ;
    end
    return binarySearch(arr, key, low, mid - 1) ;
end
```

Algorithm 3: Binary Insertion sort

If an element has to be moved far (in case of reversed sorted array), many movements are involved. Shell sort was invented to fix this problem.

3.4 Shell sort

3.4.1 Idea of algorithm

- 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.
- h is reduced until it becomes 1.
- An array is said to be h-sorted if all sub-lists of every h'th element are sorted.

3.4.2 Pseudo code

```
Function shellSort(arr)
    n ← length(arr) ;
    Rearrange elements at each n/2, n/4, n/8, ... intervals
    interval ← n / 2 ;
    while interval > 0 do
        Insertion sort for interval
        for i ← interval to n - 1 do
            selected ← arr[i] ;
            j ← i ;
            while j >= interval && arr[j - interval] > selected do
                arr[j] ← arr[j - interval] ;
                j ← j - interval ;
            end
            arr[j] ← selected ;
        end
        interval ← ⌊ interval / 2 ⌋ ;
    end
end
```

Algorithm 4: Shell sort

3.4.3 Illustration

Assume this array needs to be sorted

$$arr = \{12, 11, 13, 5, 6\}$$

Table 3.3: Illustration for Shell sort

inverval	**i**	**array**	**explain**
2	2	{ **12**, 11, **13**, 5, 6 }	Insertion for **13** and **12**.
2	3	{ 12, **11**, 13, **5**, 6 }	Insertion for **5** and **11**.
2	4	{ 12, 5, **13**, 11, **6** }	Insertion for **5** and **11**.
2	4	{ 12, 5, 6, 11, 13 }	Finish with $interval = 2$.
1	1	{ **12**, 5, 6, 11, 13 }	Insertion for **12**
1	2	{ **12**, **5**, 6, 11, 13 }	Insertion for **5** and **12**.
1	3	{ **5**, **12**, **6**, 11, 13 }	Insertion for **6**, **12** and **5**.
1	4	{ **5**, **6**, **12**, **11**, 13 }	Insertion for **11**, **12**, **6** and **5**.
1	5	{ **5**, **6**, **11**, **12**, **13** }	Insertion for **13** **12**, **11**, **6** and **5**.
1	6	{ 5,6,11,12,13 }	The array is sorted.

3.4.4 Complexity Analysis

Space

Since the algorithm does not requires extra arrays, space complexity is $\Theta(1)$.

Time

Time complexity of shell sort depends on the intervals the programmer chose. In the pseudo code implemented in 3.4.2, the original algorithm by Shell (1959) is selected.

- In worst case, time complexity is $O(n^2)$, because worst case for insertion sort in line 6 is $O(n^2)$. The reduction of intervals alone doesn't create a logarithmic time complexity in the overall algorithm.
- In best case, time complexity is $\Omega(n \log n)$.

An experiment is made to compare Insertion sort and Shell sort. In worst cases, Insertion sort's running time increases very fast when n increases.

Table 3.4: Compare running time between Insertion sort and Shell sort

Data distribution	**Randomized**		**Reversed sorted**	
n	**10 000**	**100 000**	**10 000**	**100 000**
Insertion sort	53.9181	5013.86	104.36	10165.9
Shell sort	1.555	24.0713	0.5074	6.0847

3.5 Bubble sort

3.5.1 Idea of algorithm

- The algorithm iterates through $n-1$ steps, examining pairs of adjacent elements in each iteration. Upon detecting an inversion, the algorithm swaps the elements and proceeds to evaluate the next pair.
- Following the initial step, the largest element is positioned at the rightmost of the array. Subsequently, the second largest element is situated at the second rightmost position. This process is repeated for a total of $n - 1$ steps, ultimately resulting in a sorted array.

3.5.2 Illustration

Assume this array needs to be sorted

$$arr = \{6, 0, 3, 5\}$$

The table shows how algorithm sorts array

Table 3.5: Illustration for Bubble sort

step	**array**	**explain**
0	{ 6, 0, 3, 5 | }	At first, sorted region has 0 element.
1	{ **6**, **0**, 3, 5 | }	**6** and **0** form an inversion, swap them.
1	{ 0, **6**, **3**, 5 | }	**6** and **3** form an inversion, swap them.
1	{ 0, 3, **6**, **5** | }	**6** and **5** form an inversion, swap them.
1	{ 0, 3, 5, | 6 }	First loop ends here. **6** is now at right position.
2	{ **0**, **3**, 5, | 6 }	**0** and **3** does not form an inversion. Continue.
2	{ 0, **3**, **5**, | 6 }	**3** and **5** does not form an inversion. Continue.
2	{ 0, 3, |5, 6 }	Second loop ends here. **5** is now at right position.
3	{ **0**, **3**, | 5, 6 }	**0** and **3** does not form an inversion. Continue.
3	{ | 0, 3, 5, 6 }	Third loop ends here. The array is now sorted.

3.5.3 Pseudo code

```
1  Function bubbleSort(arr)
2      n ← length(arr) ;
3      // loop through the array
4      for i ← 0 to n - 1 do
5          Bring the largest element to the end of array
6          for j ← 0 to n - i - 1 do
7              if arr[j] > arr[j + 1] then
8                  swap(arr[j], arr[j + 1]) ;
9              end
10         end
11     end
12 end
```

Algorithm 5: Bubble sort

3.5.4 Complexity Analysis

Space

Since the algorithm does not requires extra arrays, space complexity is $\Theta(1)$ for all cases.

Time

- Base operator is comparisons.
- The for loop in line 4 has to execute n times.
- The for loop in line 7 has to execute $n - i$ times.
- Therefore, the number of comparison operations needed is

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

- Both loops are independent from data distribution, so time complexity is always $\Theta(n^2)$.

3.5.5 Optimize

When looping to a certain step, if the array is already sorted, further looping is pointless and time-consuming. Therefore, it is possible to improve by placing a flag in that turn whether or not a change of position is performed.

```
Function bubbleSort(arr)
    n ← length(arr) ;
    // loop through the array
    for i ← 0 to n - 1 do
        swapped: if no swap is made in a loop, then array is sorted
        swapped ← false ;
        Bring the largest element to the end of array
        for j ← 0 to n - i - 1 do
            if arr[j] > arr[j + 1] then
                swap(arr[j], arr[j + 1]) ;
                swapped ← true ;
            end
        end
        If no two elements were swapped by inner loop, then break
        if swapped == false then break ;
        ;
    end
end
```

Algorithm 6: Bubble sort optimized

In the remaining of this report, the optimized version of bubble sort will be used when "bubble sort" mentioned.

The aforementioned enhancement proves notably efficient when sorting an array that is already ordered or has undergone substantial reduction in sorting time. However, in instances where a low-value element is positioned at the end of the array with the remaining elements in ascending order, the algorithm must traverse the entire array to relocate that element to the begin. This is due to each traversal merely shifting the element back by one position.

To address this issue, Shaker Sort adopts a strategy of reversing the traversal direction after each pass, thereby mitigating the mentioned drawback.

3.6 Shaker sort

3.6.1 Idea of algorithm

- Shaker sort is a variant of bubble sort. Bubble sort is one - way travel, shaker sort is two - way travel, forward and backward.

3.6.2 Pseudo code

```
Function shakerSort(arr)
    n ← length(arr) ;
    start ← 0 ;
    end ← n - 1 ;
    swapped ← true ;
    while swapped do
        swapped ← false ;
        Bubble sort from left to right
        for i ← start to end - 1 do
            if arr[i] > arr[i + 1] then
                swap(arr[i], arr[i + 1]) ;
                swapped ← true ;
            end
        end
        If nothing moved, then array is sorted
        if !swapped then
            break ;
        end
        Otherwise, reset the swapped flag
        swapped ← false ;
        Bubble sort from right to left
        for i ← end - 1 to start do
            if arr[i] < arr[i - 1] then
                swap(arr[i], arr[i - 1]) ;
                swapped ← true ;
            end
        end
    end
end
```

Algorithm 7: Shaker sort

3.6.3 Illustration

Assume this array needs to be sorted

$$arr = \{6, 0, 3, 5\}$$

The table shows how algorithm sorts array

Table 3.6: Illustration for Shaker sort

step	array	explain
0	{ | 6, 0, 3, 5 | }	At first, both sorted regions have 0 element.
1	{ | **6**, **0**, 3, 5 | }	**6** and **0** form an inversion, swap them.
1	{ | 0, **6**, **3**, 5 | }	**6** and **3** form an inversion, swap them.
1	{ | 0, 3, **6**, **5** | }	**6** and **5** form an inversion, swap them.
1	{ | 0, 3, 5, | 6 }	First loop ends here. **6** is now at right position.
2	{ | 0, **3**, **5**, | 6 }	**3** and **5** does not form an inversion. Continue.
2	{ | **0**, **3**, 5, | 6 }	**0** and **3** does not form an inversion. Continue.
2	{ 0, | 3, 5, | 6 }	Second loop ends here. **0** is now at right position.
3	{ 0, | **3**, **5**, | 6 }	**3** and **5** does not form an inversion. Continue.
3	{ 0, | 3, | 5, 6 }	The array is sorted.

3.6.4 Complexity Analysis

Space

Since the algorithm does not requires extra arrays, space complexity is $\Theta(1)$ for all cases.

Time

- Since shaker sort is variant of bubble sort, time complexity is still $\Theta(n^2)$.
- But running time is faster because the number of swapping operations is reduced.

An experiment is made to compare Bubble sort and Shaker sort. Due to the implementation of the random nearly sorted array function, a small and a large value are swapped. Bubble sort needs more time to bring the small values located at the end of array to their right position.

Table 3.7: Compare running time between Bubble sort and Shaker sort

Data distribution	Randomized		Nearly sorted	
n	**50 000**	**500 000**	**50 000**	**500 000**
Bubble sort	6334.1	643998	2100.23	213153
Shaker sort	4660.84	477125	2.0739	27.4674

3.7 Counting sort

3.7.1 Idea of algorithm

- Counting sort is a non - comparison - based algorithm.
- The algorithm stores the count of each unique element of the input array at their respective indices in a frequency array.
- Store the prefix sum of elements of frequency array. This will help in placing the elements of the input array at the correct index in the output array.
- From the prefix sum array, the sorted array will be formed.
- Counting sort works with integers only.

3.7.2 Pseudo code

```
Function countingSort(arr)
    n ← length(arr) ;
    Find the maximum element of array
    h ← max(arr) ;
    Create frequency array
    f ← array[n] ;
    Set all elements of frequency array to 0
    for i ← 0 to n - 1 do f[i] = 0;
    Count frequency of each element appeared in array
    for i ← 0 to n - 1 do f[arr[i]]++;
    Prefix sum to get its last position in sorted array
    for i ← 1 to h do f[i] ← f[i] + f[i - 1];
    Temporary array to save sorted array
    tmp ← array[n] ;
    for i ← 0 to n - 1 do
        tmp[ f [ arr[ i ] ] - 1] ← arr[i] ;
        f[arr[i]]- - ;
    end
    Copy data from temp array back to origin array
    arr ← tmp ;
end
```

Algorithm 8: Counting sort

3.7.3 Illustration

Assume this array needs to be sorted

$$arr = \{3, 4, 3, 5, 4, 3, 2, 2, 1, 3, 1\}$$

The frequency array is

$$f = \{0, 2, 2, 4, 2, 1\}$$

- $f[\mathbf{0}] = 0$ means there are no elements have value of **0** in the array.
- $f[\mathbf{1}] = 2$ means there are 2 elements have value of **1** in the array.
- $f[\mathbf{2}] = 2$ means there are 2 elements have value of **2** in the array.
- $f[\mathbf{3}] = 4$ means there are 4 elements have value of **3** in the array.
- $f[\mathbf{4}] = 2$ means there are 2 elements have value of **4** in the array.
- $f[\mathbf{5}] = 1$ means there are 1 elements have value of **5** in the array.

The sorted array is

$$arr = \{1, 1, 2, 2, 3, 3, 3, 3, 4, 4, 5\}$$

3.7.4 Complexity Analysis

Space

Since the algorithm requires a frequency array of length $\max(arr) - \min(arr)$ (denote: k), therefore space complexity is $O(k)$.

Time

Time complexity is $O(n + k)$ in all cases.

3.8 Heap sort

3.8.1 Definition of data structure heap

- A heap is a binary tree satisfies any node's key value is not smaller than its children's key value in a relationship.
- In max - heap, any node's key value is not smaller than its children's key value.
- In min - heap, any node's key value is not greater than its children's key value.

- In ascending - order sorting algorithm, max - heap is used, an its elements is stored in an array, where left-child of node at position i, if exists, is at position $2 * i + 1$, and right-child of node at position i, if exists, is at position $2 * i + 2$.

- So max - heap now is an array satisfies

$$\begin{cases} arr[i] \geq arr[2 * i + 1] \\ arr[i] \geq arr[2 * i + 2] \end{cases}, \forall i \in \left[0, \left\lfloor \frac{n-2}{2} \right\rfloor\right]$$

3.8.2 Idea of algorithm

- Heap sort uses data structure heap to sort an array.

- The algorithm first turns whole array into a max - heap (in case of ascending sort).

  - From left to right, the node and its children, if one or both of them exist, are considered. The father node will be swapped with the greater of its children.

  - This process is repeated until a first leaf node is considered.

- After building a heap, the array will be sorted

  - The root is taken out, and stored at the backward of the array. The heap's boundary is reduced by 1. The heap is rebuilt to have new root.

  - This process is repeated until the heap is empty. The array is sorted.

3.8.3 Pseudo code

```
Function heapify(arr, l, r)
    i ← l ;
    j ← 2 * i + 1 ;
    x ← arr[i] ;
    while j <= i do
        If node pointing by j has two children
        if j < r then
            j point to smaller node
            if arr[j] < arr[j + 1] then
                j++
            end
        end
        If true, then the heap is built.
        if x >= arr[j] then
            break ;
        end
        swap(arr[j], arr[i]) ;
        Prepare for new loop
        i ← j ;
        j ← 2 * i + 1 ;
    end
end
Function heapSort(arr)
    n ← length(arr) ;
    Prepare heap
    From right to left, push each element to heap
    for l ← ⌊n/2⌋ − 1 to 0 do heapify(arr, l, n - 1);
    Heap sort
    r ← n - 1 ;
    while r > 0 do
        Swap top element of heap to last of array
        swap(arr[0], arr[r]) ;
        r - - ;
        Push swapped element to heap again
        heapify(arr, 0, r) ;
    end
end
```

Algorithm 9: Heap sort

3.8.4 Complexity Analysis

Space

Since the algorithm does not requires extra arrays, space complexity is $\Theta(1)$ for all cases.

Time

- Building a heap costs $O(n)$.
- For sorting, it costs $O(n)$ to sort. In best case, it costs $O(1)$ to push new root back to heap, and in worst case, it costs $O(\log n)$.

Totally, it costs $O(n)$ in best case, and $O(n \log n)$ in worse case and average.

3.8.5 Illustration

To have another view, the min - heap and descending - order sorting algorithm is used for illustration.

Assume this array needs to be sorted

$$arr = \{27, 9, 1, 6, 3, 13, 14, 7, 28\}$$

Table 3.8: Illustration for Heap sort

position	**array**	**explain**
3	{ 27, 9, 1, **6**, 3, 13, 14, *7*, *28* }	First position is $\left\lfloor \frac{n-2}{2} = \frac{9-2}{2} = 3 \right\rfloor$. **6** is at right position.
2	{ 27, 9, **1**, 6, 3, *13*, *14*, 7, 28 }	**1** is at right position.
1	{ 27, **9**, 1, *6*, *3*, 13, 14, 7, 28 }	**9** will be swapped with *3*.
1	{ 27, 3, 1, 6, **9**, 13, 14, 7, 28 }	**9** does not have any children. It stays there.
0	{ **27**, *3*, *1*, 6, 9, 13, 14, 7, 28 }	**27** will be swapped with *1*.
0	{ 1, 3, **27**, 6, 9, *13*, *14*, 7, 28 }	**27** will be swapped with *13*.
0	{ 1, 3, 13, 6, 9, **27**, 14, 7, 28 }	**27** does not have any children. It stays there

The min - heap is built. Now the array will be sorted.

step	**array**	**explain**
0	{ **1**, 3, 13, 6, 9, 27, 14, 7, *28* | }	Firstly, sorted region have no elements. The root of a heap is selected and brought to the right of array.
0	{ **28**, 3, 13, 6, 9, 27, 14, 7, | 1 }	**28** will be pushed back to heap.
0	{ 3, 6, 13, 7, 9, 27, 14, **28**, | 1 }	**28** is at right position.
1	{ **3**, 6, 13, 7, 9, 27, 14, *28*, | 1 }	The root of a heap, **3**, is selected and brought to the right of array.
1	{ **28**, 6, 13, 7, 9, 27, 14, | 3, 1 }	**28** will be pushed back to heap.
1	{ 6, 7, 13, **28**, 9, 27, 14, | 3, 1 }	**28** is at right position.
2	{ **6**, 7, 13, 28, 9, 27, *14*, | 3, 1 }	**6**, is selected and brought to the right of array.
2	{ **14**, 7, 13, 28, 9, 27, | 6, 3, 1 }	**14** will be pushed back to heap.
2	{ 7, 9, 13, 28, **14**, 27, | 6, 3, 1 }	**14** is at right position.
3	{ **7**, 9, 13, 28, 14, *27*, | 6, 3, 1 }	**7**, is selected and brought to the right of array.
3	{ **27**, 9, 13, 28, 14, | 7, 6, 3, 1 }	**27** will be pushed back to heap.
3	{ 9, 14, 13, 28, **27**, | 7, 6, 3, 1 }	**27** is at right position.
4	{ **9**, 14, 13, 28, *27*, | 7, 6, 3, 1 }	**9** is selected and brought to the right of array.
4	{ **27**, 14, 13, 28, | 9, 7, 6, 3, 1 }	**27** will be pushed back to heap.
4	{ 13, 14, **27**, 28, | 9, 7, 6, 3, 1 }	**27** is at right position.
5	{ **13**, 14, 27, *28*, | 9, 7, 6, 3, 1 }	**13** is selected and brought to the right of array.
5	{ **28**, 14, 27, | 13, 9, 7, 6, 3, 1 }	**28** will be pushed back to heap.
5	{ 14, **28**, 27, | 13, 9, 7, 6, 3, 1 }	**28** is at right position
6	{ **14**, 28, *27*, | 13, 9, 7, 6, 3, 1 }	**28** is at right position
6	{ **27**, 28, | 14, 13, 9, 7, 6, 3, 1 }	**27** will be pushed back to heap.
6	{ **27**, 28, | 14, 13, 9, 7, 6, 3, 1 }	**27** is at right position.
7	{ **27**, *28*, | 14, 13, 9, 7, 6, 3, 1 }	**27** is selected and brought to the right of array.
7	{ 28, | 27, 14, 13, 9, 7, 6, 3, 1 }	The array is sorted.

3.9 Merge sort

3.9.1 Idea of algorithm

Merge sort is a recursive algorithm works by following steps:

- If array contains less than two elements, do nothing and return.
- Else, divide array into two halves, sort them and merge them back into one array.

3.9.2 Illustration

Assume this array needs to be sorted

$$arr = [4, 2, 3, 1, 8, 6, 7, 5]$$

The table shows how algorithm sorts array

Table 3.9: Illustration for Merge sort

array	**explain**
[4, 2, 3, 1, 8, 6, 7, 5]	The array will be divided into two halves.
[[4, 2, 3, 1], [8, 6, 7, 5]]	Each half will be divided into two halves.
[[[4, 2], [3, 1]], [[8, 6], [7, 5]]]	Each half will be divided into two halves.
[[[[4], [2]], [[3], [1]]], [[[8], [6]], [[7], [5]]]]	Each half will be divided into two halves.
[[[2, 4], [1, 3]], [[6, 8], [5, 7]]]	Merge two halves.
[[1, 2, 3, 4], [5, 6, 7, 8]]	Merge two halves.
[1, 2, 3, 4, 5, 6, 7, 8]	Merge two halves.

3.9.3 Pseudo code

```
Function mergeSort(arr, l, r)
    if l < r then
        mid ← l + ⌊(r − l)/2⌋ ;
        Sort first and second halves
        mergeSort(arr, l, mid) ;
        mergeSort(arr, mid + 1, r) ;
        Merge the sorted halves
        merge(arr, l, mid, r) ;
    end
end
Function merge(arr, l, m, r)
    Create temporary arrays
    leftArr ← arr[l..m] ;
    rightArr ← arr[m+1..r] ;
    Merge the temp arrays back into arr[l..r]
    leftID ← 0 ;
    rightID ← 0 ;
    mergedId ← l ;
    while leftID < length(leftArr) && rightID < length(rightArr) do
        if leftArr[leftID] <= rightArr[rightID] then
            arr[mergedId] ← leftArr[leftID] ;
            leftID++ ;
        end
        else
            arr[mergedId] ← rightArr[rightID] ;
            rightID++ ;
        end
        mergedId++ ;
    end
    Copy remaining elements of leftArr[] if any
    while leftID < length(leftArr) do
        arr[mergedId] ← leftArr[leftID] ;
        leftID++ ;
        mergedId++ ;
    end
    Copy remaining elements of rightArr[] if any
    while rightID < length(rightArr) do
        arr[mergedId] ← rightArr[rightID] ;
        rightID++ ;
        mergedId++ ;
    end
end
```

Algorithm 10: Merge sort

3.9.4 Complexity Analysis

Space

Each recursive step requires extra array to copy data, space complexity is $\Theta(n)$ for all cases.

Time

- It takes $O(n)$ to merge two halves.
- The original is divided $\log_2 n$ times.
- So time complexity is $O(n \log n)$ in all cases, since above steps are executed independently of data distribution.

3.10 Quick sort

3.10.1 Idea of algorithm

- One random element is chosen as a pivot.
- The array will be split into to segments: one segment contains elements smaller than pivot, another contains elements greater than pivot.
- The algorithm call itself to partition two segments.

3.10.2 Illustration

Assume this array needs to be sorted, an pivot is the rightmost element of the array

$$arr = [4, 1, 3, 2, 7, 6, 8, 5]$$

The table shows how algorithm sorts array

Table 3.10: Illustration for Quick sort

array	**explain**
[4, 1, 3, 2, 7, 6, 8, **5**]	The array will be divided into two segments.
[[4, 1, 3, 2], **5**, [6, 8, 7]]	The array is partitioned.
[[4, 1, 3, **2**], 5, [6, 8, **7**]]	Recursive sort to sub-array.
[[[1], **2**, [3, 4]], 5, [[6], **7**, [8]]]	Finish next stage.
[[[1], 2, [3, **4**]], 5, [[6], 7, [8]]]	Recursive sort to sub-array.
[[[1], 2, [3, **4**]], 5, [[6], 7, [8]]]	The array is sorted.

3.10.3 Pseudo code

```
Function quickSort(arr, l, r)
    if l < r then
        pi is partitioning index, arr[pi] is now at right place
        pi ← partition(arr, l, r) ;
        Separately sort elements before partition and after partition
        quickSort(arr, l, pi - 1) ;
        quickSort(arr, pi + 1, r) ;
    end
end
Function partition(arr, l, r)
    Choose the pivot
    swap(arr[random(l, r)], arr[r]) ;
    pivot ← arr[r] ;
    Index of smaller element and indicate the right position of pivot
    found so far
    i ← (l - 1) ;
    for j ← l to r do
        If current element is smaller than the pivot
        if arr[j] < pivot then
            Increment index of smaller element
            i++ ;
            swap(arr[i], arr[j]) ;
        end
    end
    swap(arr[i+1], arr[r]) ;
    return (i+1) ;
end
```

Algorithm 11: Quick sort

3.10.4 Complexity Analysis

Space

Since the algorithm does not requires extra arrays, space complexity is $\Theta(1)$ for all cases.

Time

- Best cases: $\Omega(n \log n)$. This case occurs when the pivot chosen at each step divides the array into two halves.
- Worst csae: $O(n^2)$. This occurs when the pivot chosen at each step is always the smallest or largest element.
- Average: $\Theta(n \log n)$.

3.11 Radix sort

3.11.1 Idea of algorithm

- Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value.
- There are two variations: LSD and MSD. In number 1234, if digit **1** is checked first, it is MSD (Most Significant Digit), if **4** is checked first, it is LSD (Least Significant Digit).
- In this report, the chosen radix is 10, and LSD.

3.11.2 Illustration

Assume this array needs to be sorted

$$arr = [72, 34, 52, 44, 76, 56]$$

The table shows how algorithm sorts array

Table 3.11: Illustration for Radix sort

digit	**array**	**explain**
0	[72, 34, 52, 44, **76**, 56]	The maximum element has two digits.
1	[7**2**, 3**4**, 5**2**, 4**4**, 7**6**, 5**6**]	The array will be divided into groups based on their last significant digit.
1	{ {7**2**, 5**2**}, {3**4**, 4**4**}, {7**6**, 5**6**} }	The array is divided into three groups.
1	[72, 52, 34, 44, 76, 56]	Merged back.
2	[**7**2, **5**2, **3**4, **4**4, **7**6, **5**6]	The array will be divided into groups based on their second last significant digit
2	{ { **3**4 }, { **4**4 }, { **5**2, **5**6 }, { **7**2, **7**6 } }	The array is divided into three groups.
2	[34, 44, 56, 56, 72, 76]	Merged back.

3.11.3 Pseudo code

```
Function radixSort(arr)
    n ← length(arr) ;
    Find the maximum number to know the number of digits
    maxNum ← max(arr) ;
    Do counting sort for every digit
    for exp ← 1; maxNum / exp > 0; exp *= 10 do
        countSort(arr, n, exp) ;
    end
end
Function countSort(arr, n, exp)
    Counting sort of arr[] according to the digit represented by exp
    k ← 10 ;
    output ← array[n] ;
    count ← array[k] with all elements 0 ;
    Count occurrences of each digit in the input array
    for i ← 0 to n - 1 do
        count[(arr[i] / exp) % 10]++ ;
    end
    Update count[i] to store the position of the next occurrence
    for i ← 1 to k - 1 do
        count[i] += count[i - 1] ;
    end
    Build the output array
    for i ← n - 1 to 0 do
        output[count[(arr[i] / exp) % 10] - 1] ← arr[i] ;
        count[(arr[i] / exp) % 10]– ;
    end
    Copy the output array to arr[] so that arr[] contains sorted
    numbers based on current digit
    arr[i] ← output[i] ;
end
```

Algorithm 12: Radix sort

3.11.4 Complexity Analysis

Space

Since the algorithm does not requires extra arrays, space complexity is $\Theta(1)$.

Time

- Best cases: $\Omega(n \log n)$. This case occurs when the pivot chosen at each step divides the array into two halves.
- Worst csae: $O(n^2)$. This occurs when the pivot chosen at each step is always the smallest or largest element.
- Average: $\Theta(n \log n)$.

3.12 Flash sort

3.12.1 Idea of algorithm

- The algorithm divides elements into m segments.
- Element x belongs to segment $k[x] = \left\lfloor (m-1) * \frac{x - min(arr)}{max(arr) - min(arr)} \right\rfloor$.
- After partitioning, use insertion sort for each partitions.

3.12.2 Illustration

Assume this array needs to be sorted

$$arr = [4, 1, 0, 3, 2]$$

With $m = 3$, the partitioning table is

element	**segment**
4	3
1	1
0	1
3	3
2	2

After partition, array becomes

$$arr = [1, 0, |3, 2, |4]$$

3.12.3 Pseudo code

```
Function flashSort(arr, n)
    Step 0: Find min and max
    min ← min(arr) ;
    max ← max(arr) ;
    If arr[i] == arr[j], for all i, j
    if max == min then return;
    ;
    Step 1: Determine the size of partitions
    m ← n * 0.45 ;
    if m <= 2 then m ← 2 ;
    ;
    L ← array[m] with all elements 0 ;
    for i ← 0 to n - 1 do
        k ← (m − 1) ∗ ⌊(arr[i] − min) / (max − min)⌋ ;
        L[k]++ ;
    end
    for k ← 1 to m − 1 do L[k] ← L[k] + L[k - 1];
    Step 2: Partition
    i ← 0 ;
    count ← 0 ;
    k ← m - 1 ;
    while count < n do
        while i >= L[k] do
            i++ ;
            k ← (m − 1) ∗ ⌊(arr[i] − min) / (max − min)⌋ ;
        end
        flash ← arr[i] ;
        while i != L[k] do
            k ← (m − 1) ∗ ⌊(flash − min) / (max − min)⌋ ;
            swap(arr[L[k] - 1], flash) ;
            L[k] - - ;
            count++ ;
        end
    end
end
```

Algorithm 13: Flash sort

The array is partitioned into m segments. Use insertion sort 3.2 for sorting each segment. Insertion sort is chosen because its speed when sorting array with small number of elements.

3.12.4 Complexity Analysis

Space

Space complexity is $O(m)$, for array L in line 13.

Time

- Time complexity for partition step is $O(n)$, sice each element is considered once.
- In average, each segment after partitioning has $\frac{n}{m}$ elements. Therefore, due to the time complexity of insertion sort, time complexity for sorting each segment is $O(\frac{n^2}{m^2})$. There are m segments, so time complexity for sorting array is $O(\frac{n^2}{m^2} * m) = O(\frac{n^2}{m})$.
- After experiments, $m = 0.43n$ returns best complexity. (Neubert, 1998).
- In worst case, time complexity can reach $O(n^2)$, when data is unevenly distributed.

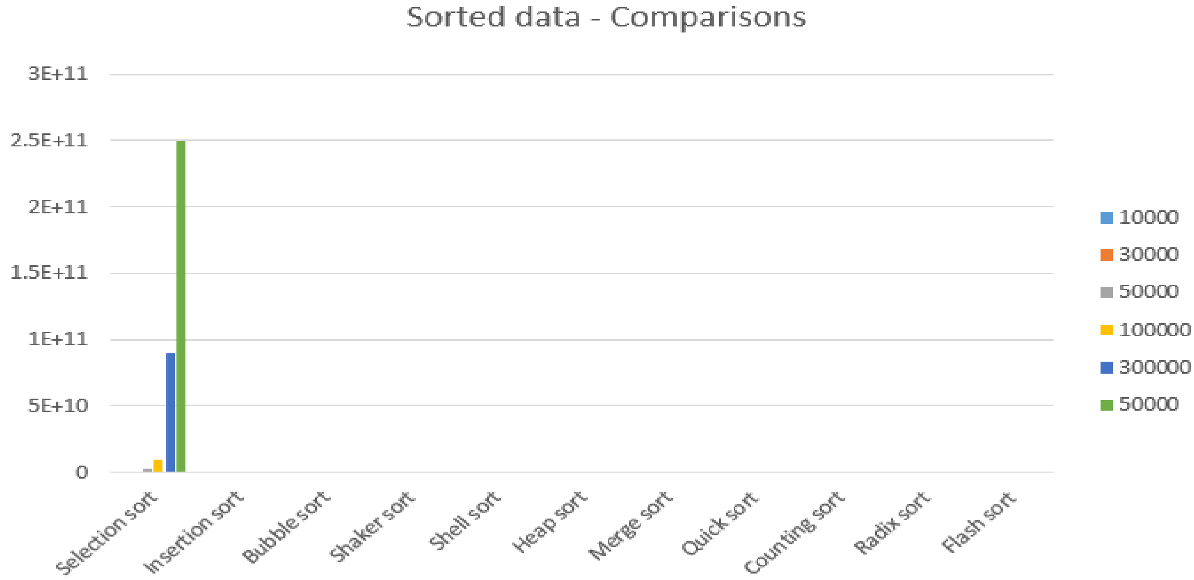
Chapter 4

Experimental results and comments

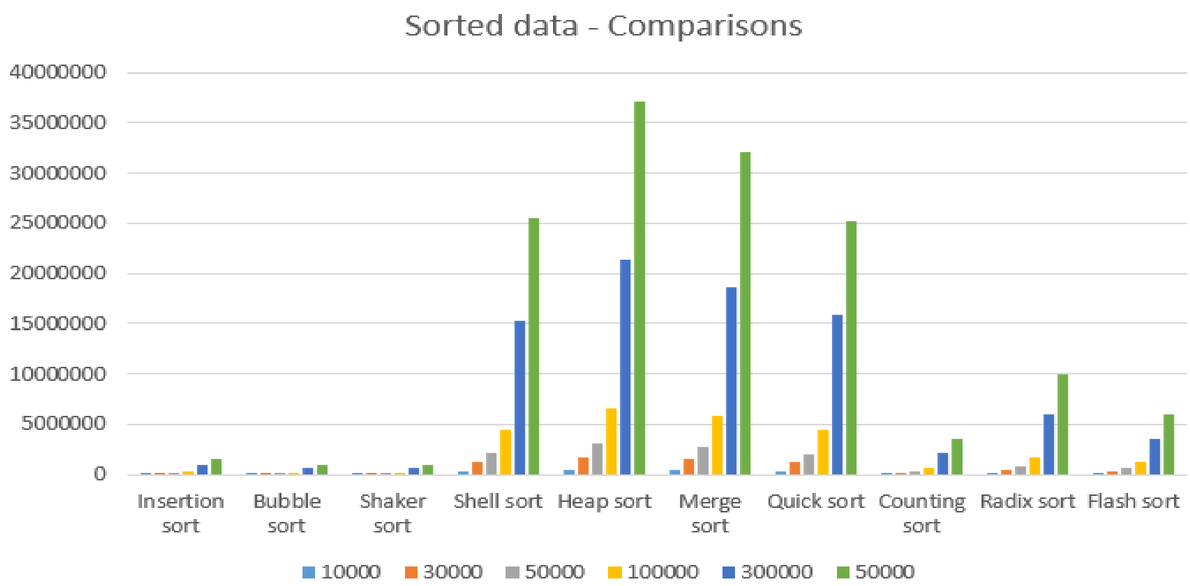
4.1 Sorted data

These charts show number of comparisons when the algorithms sort the sorted array

Figure 4.1: Sorted data - Comparisons



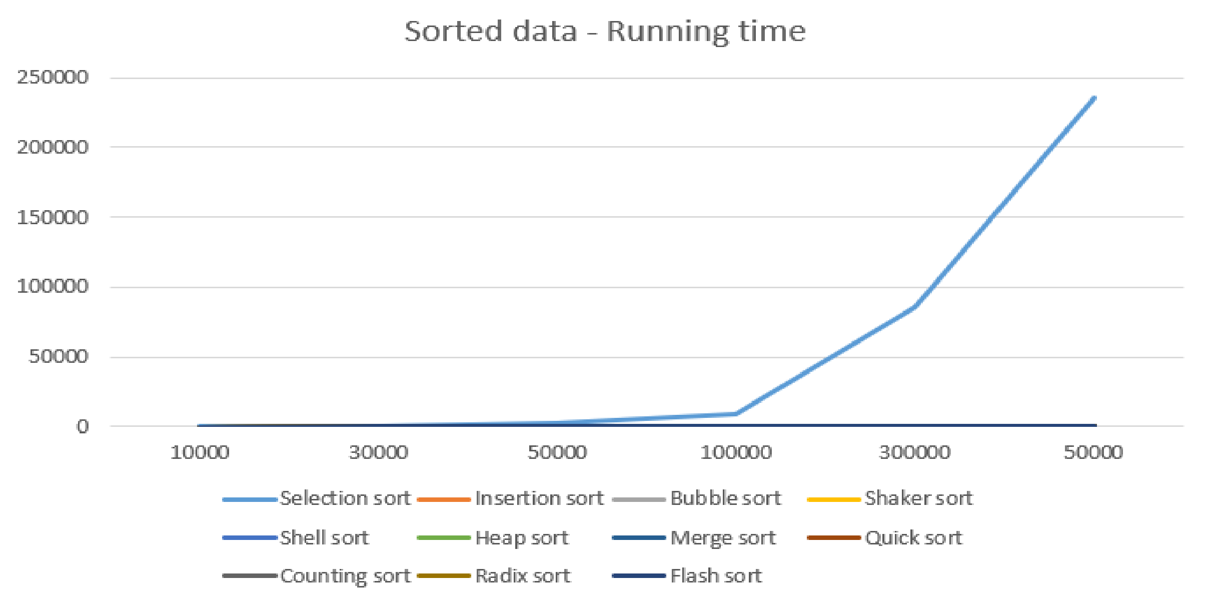
Remove Selection sort from a chart



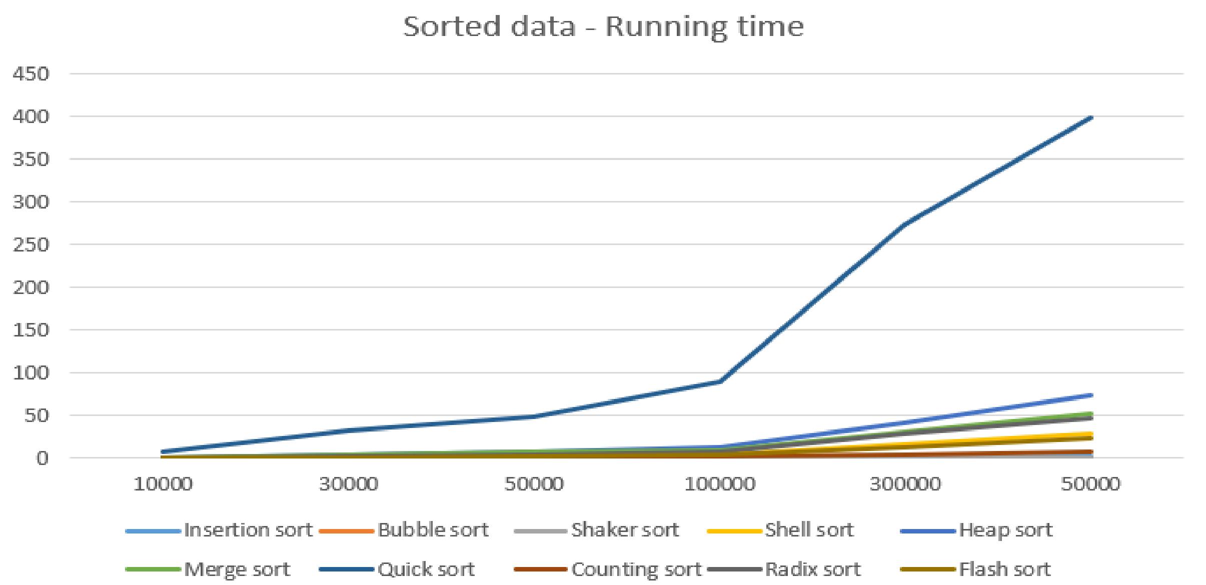
The number of comparisons made by Selection sort is very high, since Insertion sort works very fast when sorting the sorted arrays, Bubble sort and Shaker sort both have a mechanism allowed them to break when meeting a sorted array, and the other algorithms are much more efficient than Selection sort.

These line graph show running time when the algorithms sort the sorted array

Figure 4.2: Sorted data - Running time



The reason why Selection sort costs lots of time to run it explained above. Remove Selection sort from a graph



Only one line is not normal here, Quick sort's running time is much higher than the others. This happens because the pivot chosen is random, and because the array is already sorted, the probability of the pivot chosen separate the array into two halves is low. That is why its running time is much higher than other algorithms.

Table 4.1: Data - order: sorted data

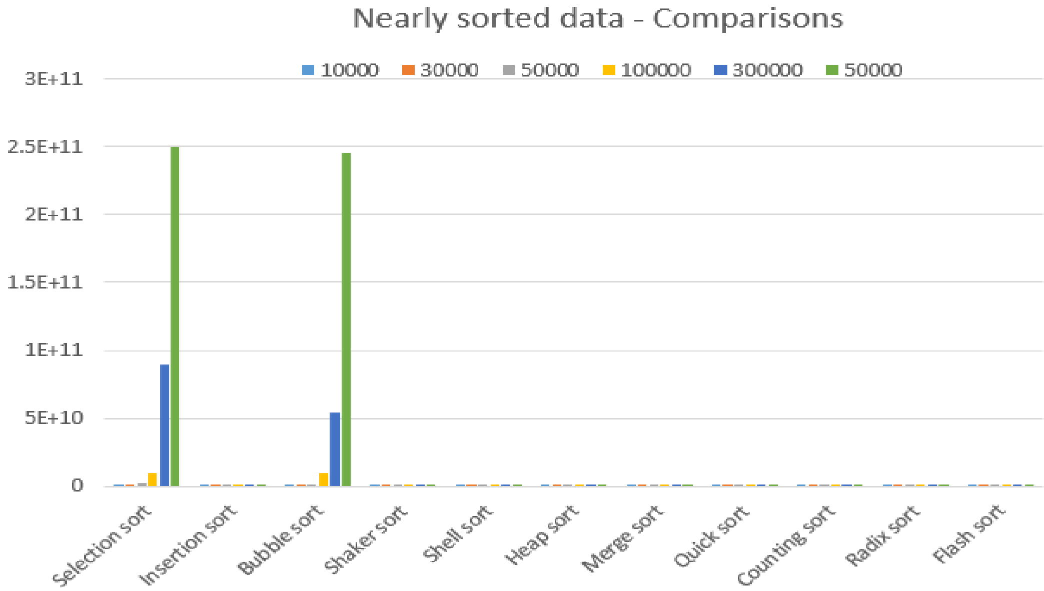
Data order Sorted data						
Datasize	**10000**		**30000**		**50000**	
Resulting statics	**Comparisons**	**Running time**	**Comparisons**	**Running time**	**Comparisons**	**Running time**
Selection sort	100019998	96.2864	900059998	805.218	2500099998	2294.9
Insertion sort	29998	0.0272	89998	0.0841	149998	0.1647
Bubble sort	20001	0.0168	60001	0.0522	100001	0.1163
Shaker sort	19999	0.0182	59999	0.0562	99999	0.0879
Shell sort	360042	0.3254	1170050	1.1348	2100049	2.9733
Heap sort	518705	1.0428	1739633	3.7386	3056481	7.9947
Merge sort	475242	0.8436	1559914	3.5848	2722826	6.8961
Quick sort	356967	7.8724	1276873	33.0438	2005123	49.3276
Counting sort	70003	0.3501	210003	0.4068	350003	0.7236
Radix sort	140056	0.897	510070	2.4651	850070	3.8429
Flash sort	119000	0.404	357000	1.269	595000	2.0921

Datasize	**100000**		**300000**		**50000**	
Resulting statics	**Comparisons**	**Running time**	**Comparisons**	**Running time**	**Comparisons**	**Running time**
Selection sort	10000199998	9285.45	90000599998	85347.3	250000999998	234893
Insertion sort	299998	0.2788	899998	0.7971	1499998	1.7079
Bubble sort	200001	0.1744	600001	0.5113	1000001	1.0486
Shaker sort	199999	0.1874	599999	0.5489	999999	0.9716
Shell sort	4500051	4.4216	15300061	16.0825	25500058	29.5994
Heap sort	6519813	13.8387	21431637	40.9354	37116275	73.1254
Merge sort	5745658	9.8761	18645946	31.4051	32017850	52.1767
Quick sort	4490896	89.9941	15934253	272.911	25195263	399.325
Counting sort	700003	1.5004	2100003	4.48189	3500003	7.7054
Radix sort	1700070	8.1059	6000084	29.3186	10000084	47.417
Flash sort	1190000	4.3061	3570000	12.4245	5950000	23.3434

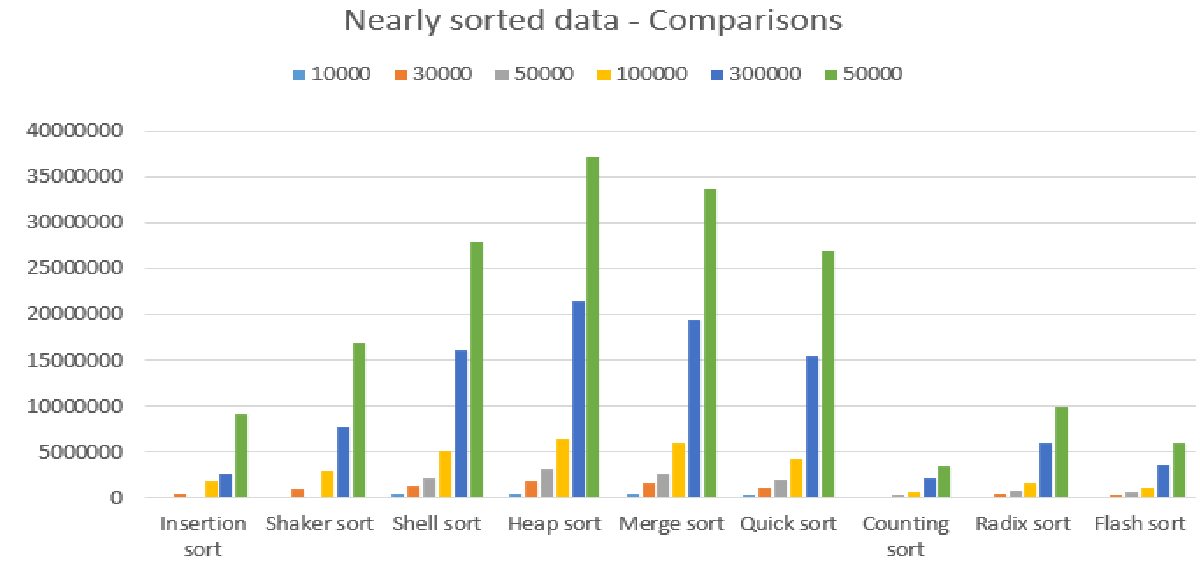
4.2 Nearly sorted data

These charts show number of comparisons when the algorithms sort the nearly sorted array

Figure 4.3: Nearly sorted data - Comparisons



Remove Selection sort and Bubble sort from a chart



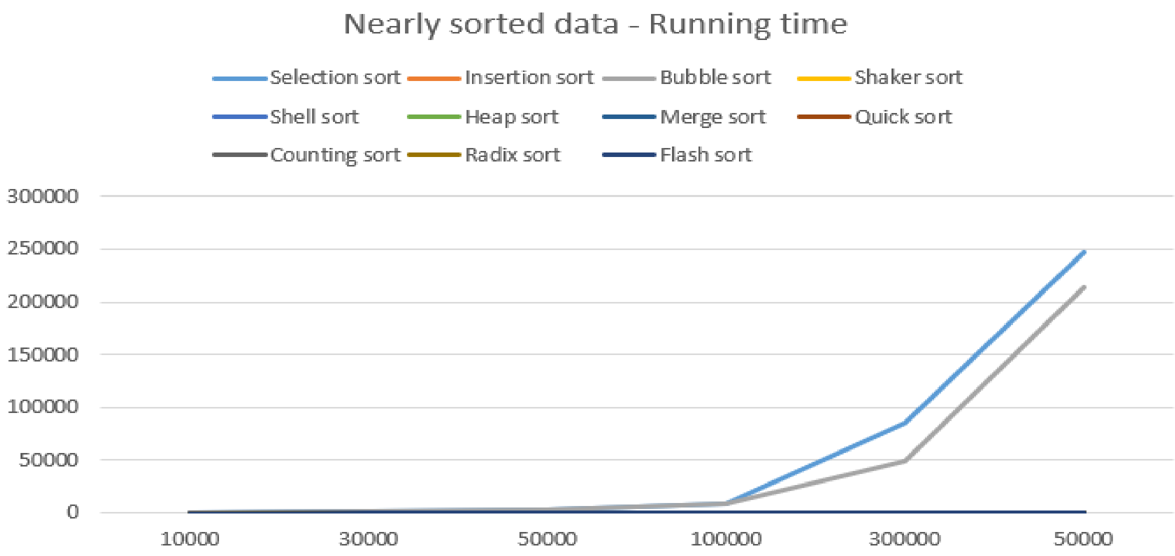
The reason why the number of comparisons of Bubble sort gets high is explained in 3.5.5. Insertion sort keeps its fast speed because this experiment uses nearly sorted data.

Table 4.2: Data - order: nearly sorted data

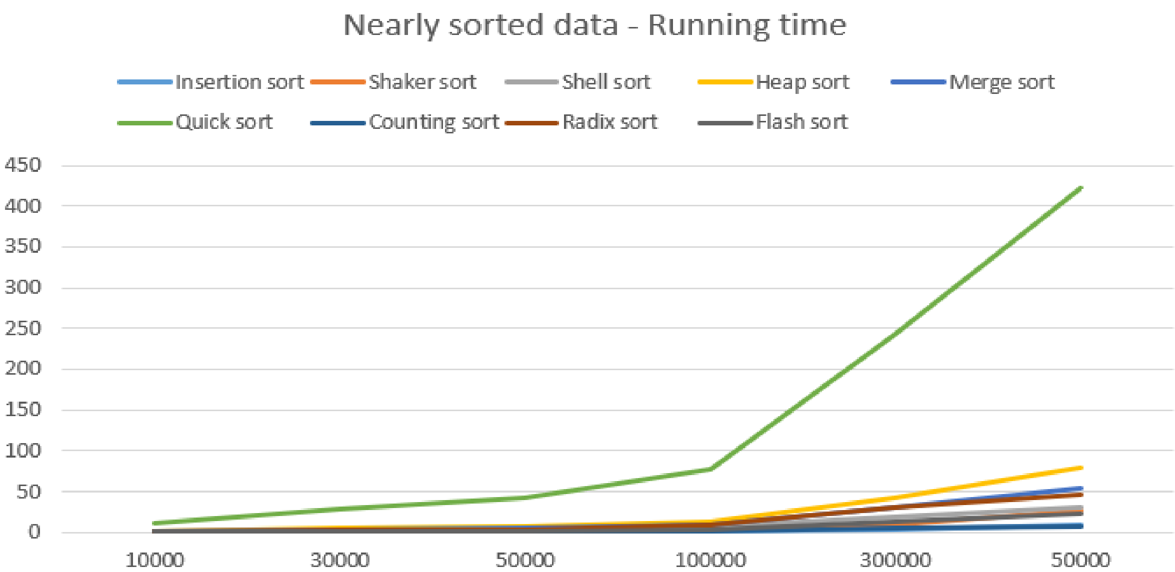
Data order Nearly sorted data						
Datasize	**10000**		**30000**		**50000**	
Counting sort	**Comparisons**	**Running time**	**Comparisons**	**Running time**	**Comparisons**	**Running time**
Selection sort	100019998	95.8791	900059998	800.118	2500099998	2235.68
Insertion sort	126638	0.1222	450282	0.4489	778254	0.8097
Bubble sort	93924040	84.3582	760536657	655.726	2461312017	2100.23
Shaker sort	219879	0.4084	899775	1.2233	1299831	2.0739
Shell sort	400902	0.433799	1296018	1.5364	2386473	3.1536
Heap sort	518548	1.0452	1739672	4.4733	3056461	6.5843
Merge sort	502717	0.856	1638335	4.0481	2894240	4.648
Quick sort	346580	10.5369	1168928	28.7507	2104137	42.4138
Counting sort	70003	0.5179	210003	0.4138	350003	0.7268
Radix sort	140056	0.6279	510070	2.4048	850070	4.1443
Flash sort	118977	0.614	356973	1.2148	594977	2.0378

Datasize	**100000**		**300000**		**50000**	
Counting sort	**Comparisons**	**Running time**	**Comparisons**	**Running time**	**Comparisons**	**Running time**
Selection sort	10000199998	9163.6	90000599998	85308.5	250000999998	247640
Insertion sort	1814974	2.106	2616150	2.8135	9103482	9.8182
Bubble sort	9697196352	8490.01	54459301425	48587.6	245213328752	213153
Shaker sort	2999775	4.3468	7799831	9.3559	16999711	27.4674
Shell sort	5131771	6.8392	16122798	18.5211	27814150	31.1973
Heap sort	6519805	12.9089	21431691	42.0096	37111283	78.8615
Merge sort	6035245	9.6227	19360056	31.4401	33634038	54.6727
Quick sort	4221055	77.3767	15481165	242.769	26830065	421.998
Counting sort	700003	1.5704	2100003	4.6373	3500003	6.9059
Radix sort	1700070	8.4764	6000084	30.8014	10000084	46.5434
Flash sort	1189975	4.0161	3569975	13.0936	5949976	21.9048

Figure 4.4: Nearly sorted data - Running time



Remove Selection sort and Bubble sort from a chart

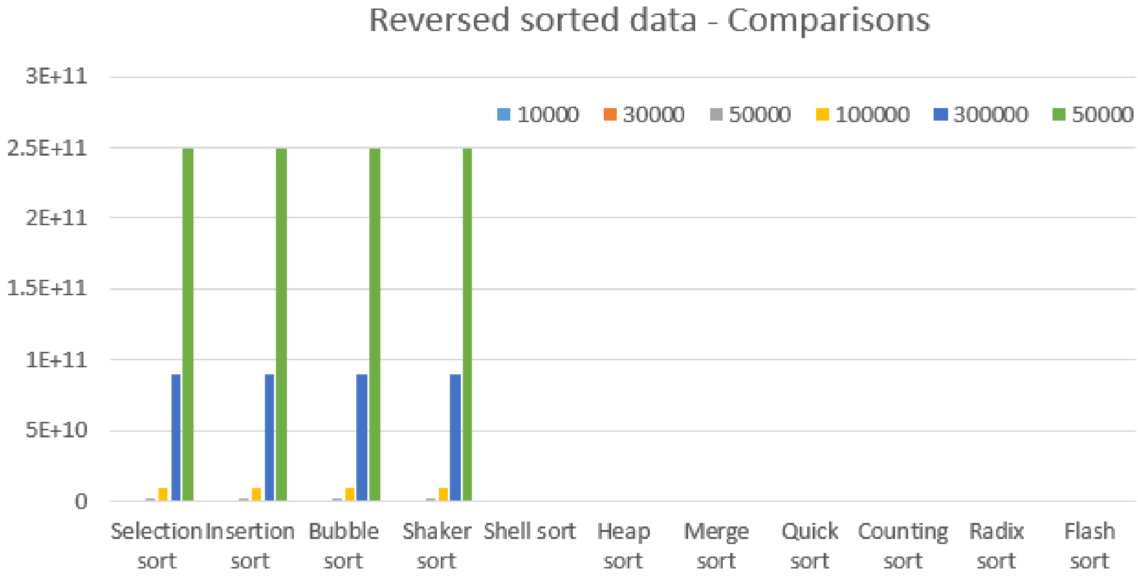


The weakness of Quick sort when sorting a nearly sorted array is as same as when sorting sorted array. Bubble sort runs faster than Selection sort because it has mechanism to realize when the array is sorted. The mechanism is explained in Algorithm 6

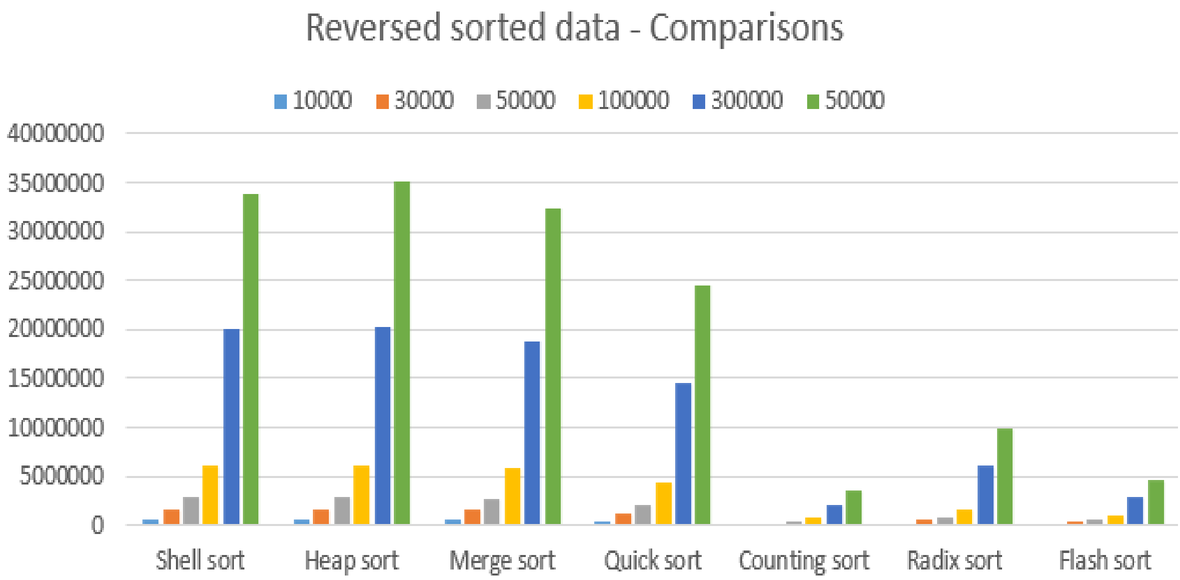
4.3 Reversed sorted data

These charts show number of comparisons when the algorithms sort the reversed sorted array

Figure 4.5: Reversed sorted data - Comparisons



Remove Selection sort, Insertion sort, Bubble sort, Shaker sort from a chart



This experiment uses reversed sorted array, therefore the advantages of Insertion sort and Shaker sort are gone. Selection sort, Insertion sort, Bubble sort and Shaker sort now are same, they are all $O(n^2)$ algorithms.

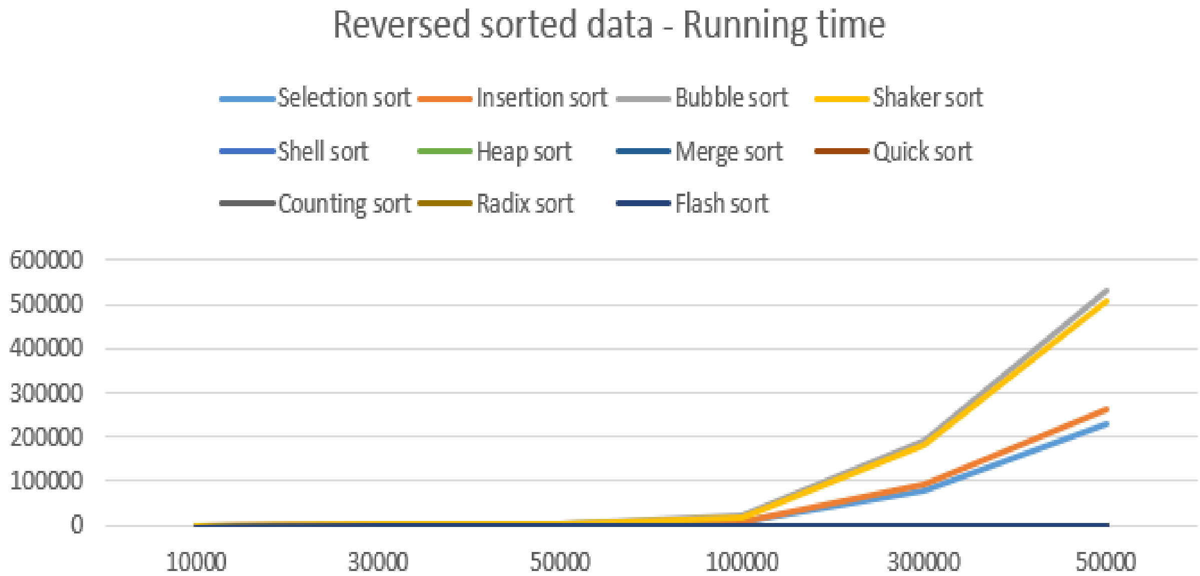
Table 4.3: Data - order: reversed sorted data

Data order Reverse sorted data						
Datasize	**10000**		**30000**		**50000**	
Resulting statics	**Comparisons**	**Running time**	**Comparisons**	**Running time**	**Comparisons**	**Running time**
Selection sort	100019998	96.3312	900059998	788.027	2500099998	2136.16
Insertion sort	100009999	104.36	900029999	907.416	2500049999	2486.1
Bubble sort	100019998	210.961	900059998	1872.56	2500099998	5514.29
Shaker sort	100000000	206.309	900000000	1822.39	2500000000	5050.89
Shell sort	475175	0.5074	1554051	1.8662	2844628	2.8571
Heap sort	476739	1.0432	1622791	5.3303	2848016	6.4839
Merge sort	476441	0.8646	1573465	4.0715	2733945	4.4712
Quick sort	338677	12.3484	1164547	28.0837	2038327	43.859
Counting sort	70003	0.2683	210003	0.6512	350003	0.7191
Radix sort	140056	1.0803	510070	2.4029	850070	3.7914
Flash sort	93754	0.5597	281254	1.2023	468754	1.8048

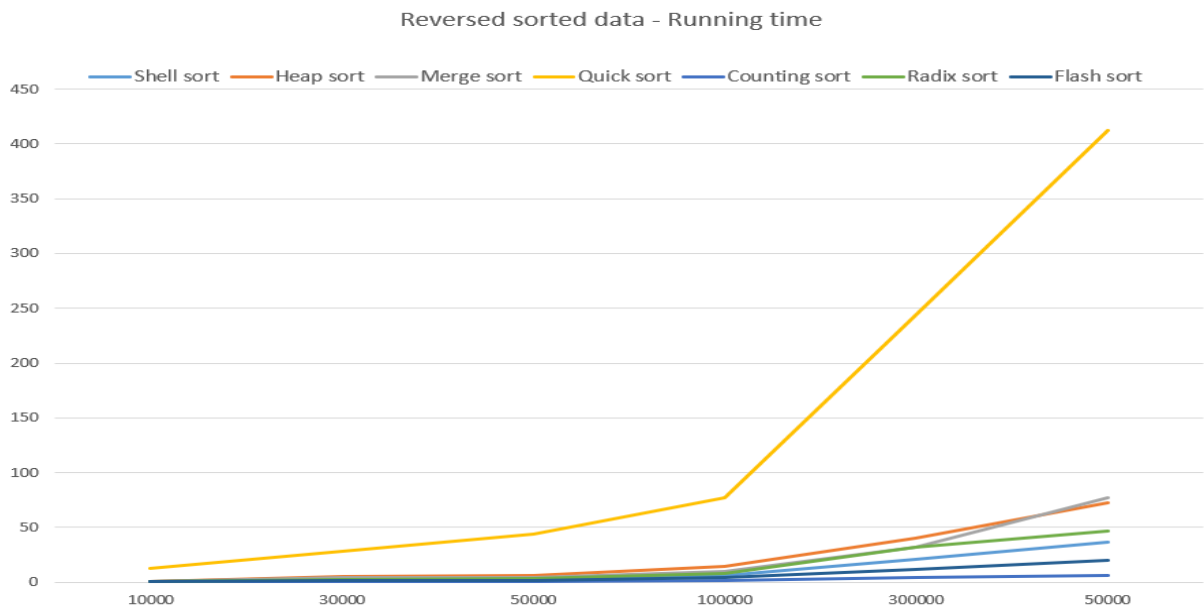
Datasize	**100000**		**300000**		**50000**	
Resulting statics	**Comparisons**	**Running time**	**Comparisons**	**Running time**	**Comparisons**	**Running time**
Selection sort	10000199998	8617.05	90000599998	76592.2	250000999998	229732
Insertion sort	10000099999	10165.9	90000299999	91942.3	250000499999	261026
Bubble sort	10000199998	21226	90000599998	190314	250000999998	529091
Shaker sort	10000000000	19656.4	90000000000	183563	250000000000	505338
Shell sort	6089190	6.0847	20001852	20.7879	33857581	36.2318
Heap sort	6087452	14.1803	20187386	40.4779	35135730	72.3901
Merge sort	5767897	10.1321	18708313	32.2165	32336409	77.1762
Quick sort	4424587	76.8702	14524899	243.568	24559163	412.199
Counting sort	700003	1.3295	2100003	4.0414	3500003	6.6226
Radix sort	1700070	7.865	6000084	32.1407	10000084	46.5935
Flash sort	937504	3.8868	2812504	11.3812	4687504	19.6628

These graph show running time when the algorithms sort the reversed sorted array

Figure 4.6: Reversed sorted data - Running time



Remove 4 $O(n^2)$ algorithms from a graph

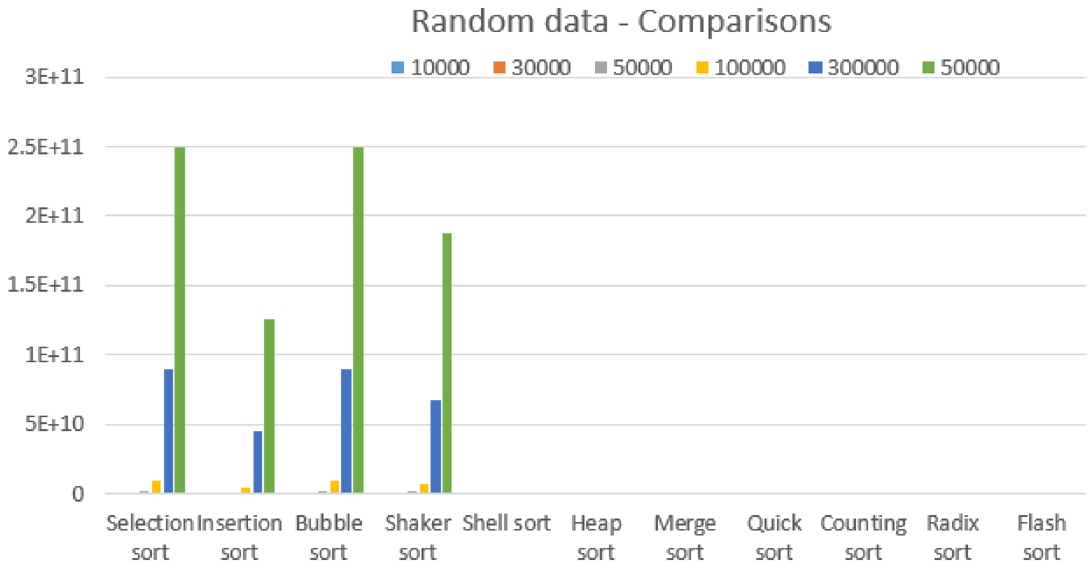


Bubble sort and its optimization algorithm, Shaker sort are the slowest algorithms. Sorting reversed data makes them very slow. Counting sort is the fastest algorithm, and right behind it is Flash sort.

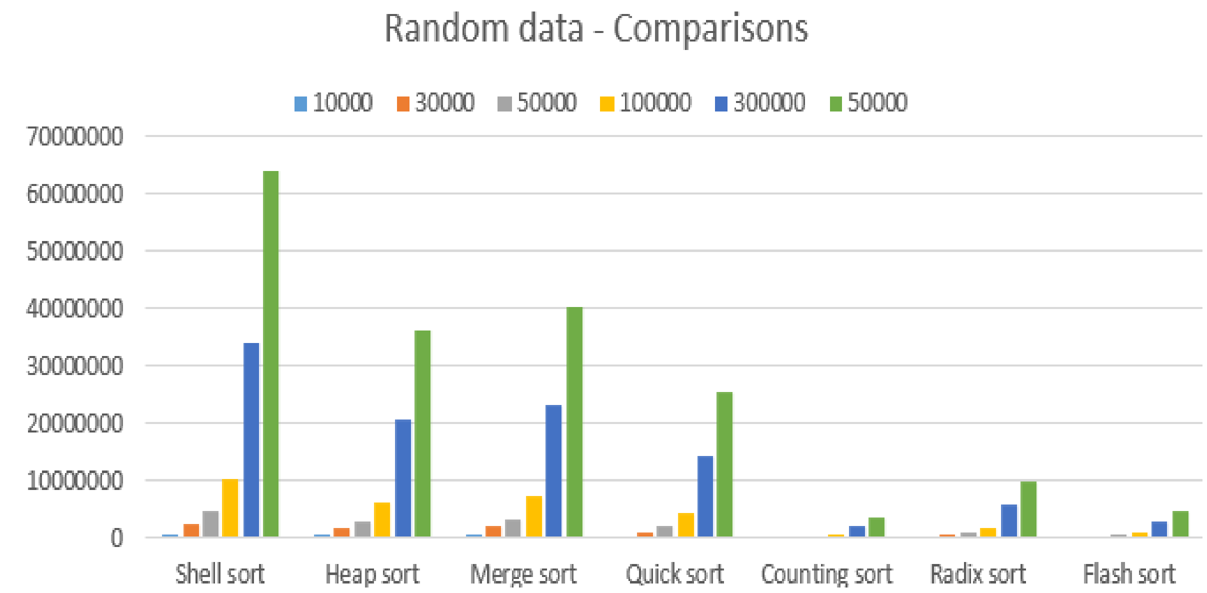
4.4 Random data

These charts show number of comparisons when the algorithms sort the random array

Figure 4.7: Random data - Comparisons



Remove Selection sort, Insertion sort, Bubble sort, Shaker sort from a chart



Shaker sort uses more comparisons than Bubble sort because Shaker sort is an optimization of Bubble sort, it travels less than Bubble sort. And they are both worse than Insertion sort. Selection sort uses a lot of comparisons while sorting array. But using most comparisons does not mean it is the slowest algorithm.

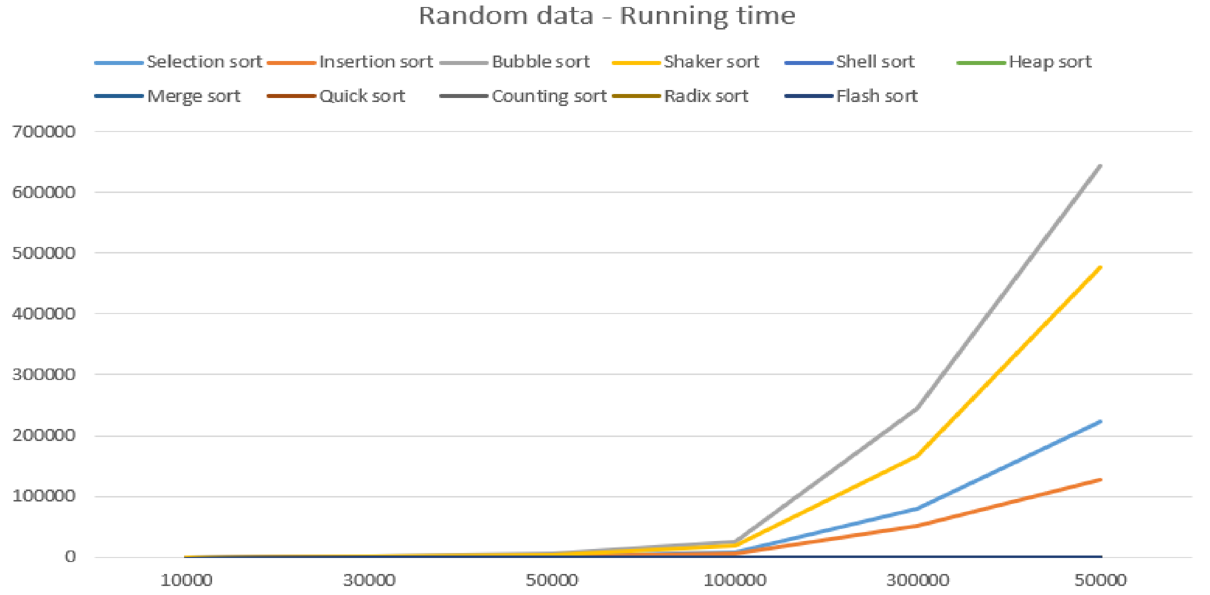
Table 4.4: Data - order: reversed sorted data

Data order Randomized data						
Datasize	**10000**		**30000**		**50000**	
Resulting statics	**Comparisons**	**Running time**	**Comparisons**	**Running time**	**Comparisons**	**Running time**
Selection sort	100019998	98.1629	900059998	811.446	2500099998	2251
Insertion sort	49974355	53.9181	449016555	466.717	1251895433	1301.13
Bubble sort	100018480	224.997	900056032	2233.54	2500091901	6334.1
Shaker sort	74769471	171.252	674399600	1628.38	1872046519	4660.84
Shell sort	646678	1.555	2302725	5.7788	4646259	10.2784
Heap sort	497302	1.3724	1680515	5.9896	2952195	11.2431
Merge sort	583780	1.4579	1937280	7.1702	3383011	8.9642
Quick sort	353748	11.3624	1181451	28.0247	2111668	46.1852
Counting sort	70003	0.1645	209999	0.4619	349997	0.867
Radix sort	140056	0.6038	510070	2.4963	850070	3.7654
Flash sort	97527	0.6006	306271	1.6157	494626	3.3008

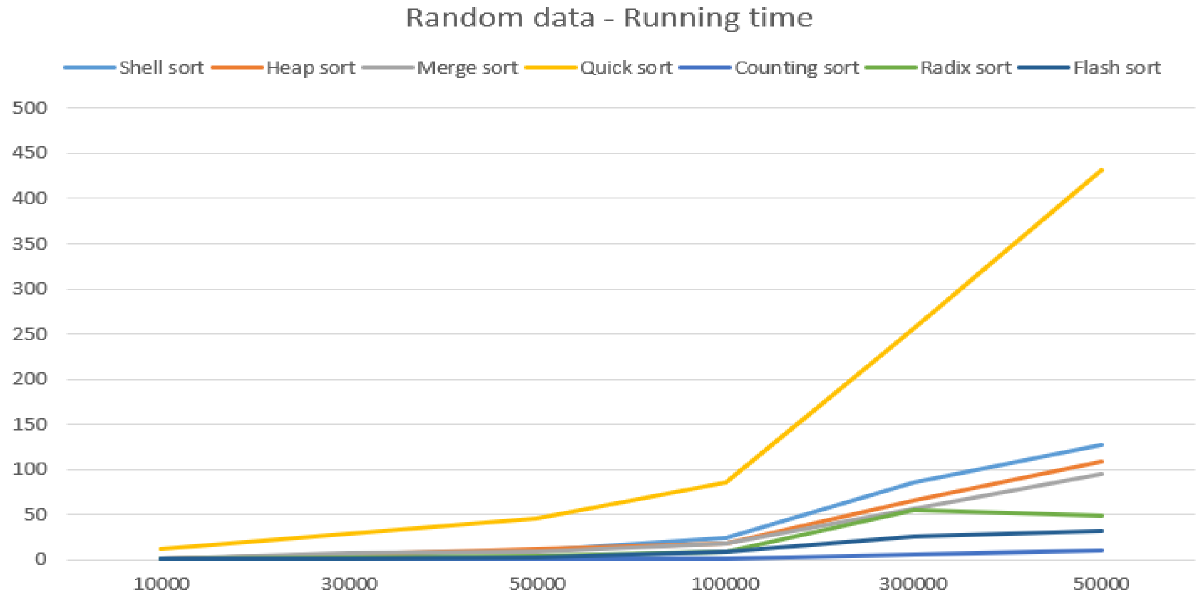
Datasize	**100000**		**300000**		**50000**	
Resulting statics	**Comparisons**	**Running time**	**Comparisons**	**Running time**	**Comparisons**	**Running time**
Selection sort	10000199998	8980.34	90000599998	79508.7	250000999998	222164
Insertion sort	5001866366	5013.86	44889583761	50845.5	125203211331	126346
Bubble sort	10000038397	25753.4	90000425277	245011	249999759005	643998
Shaker sort	7512085359	18813.2	67378937591	167260	187603956736	477125
Shell sort	10165918	24.0713	33927734	86.2957	63944956	126.51
Heap sort	6304095	18.2059	20798663	66.2753	36121793	108.54
Merge sort	7166164	17.2295	23382960	56.788	40382902	94.5375
Quick sort	4262666	85.0057	14183472	255.788	25438524	431.141
Counting sort	700003	1.5582	2100003	5.0146	3499999	9.5448
Radix sort	1700070	8.6169	6000084	54.1411	10000084	48.3469
Flash sort	954259	9.028	2845227	25.095	4688996	31.6843

These graph show number of comparisons when the algorithms sort the random array

Figure 4.8: Random data - Running time



Remove Selection sort, Insertion sort, Bubble sort, Shaker sort from a graph



Although using most comparisons, Selection sort is not the slowest. Because comparisons are executed in cache, and assignments are executed in RAM. Cache is much faster than RAM.

Counting sort and Flash sort run very fast. But these experiments use small data. For larger data, Counting sort cannot be used.

4.5 Overall

Counting sort and Flash sort are the fastest algorithms, in all data distribution cases. Counting sort is faster than Flash sort because its programming constant is less than Flash sort. But for large data, counting sort cannot be used.

Bubble sort, and Shaker sort should be mentioned too, are the slowest algorithms. Shaker sort is better than Bubble sort, but not much change. In some special cases, when the data is sorted or nearly sorted, they are fast because there is a mechanism allowed them to realize when the array is already sorted.

The experiments show that Quick sort is unstable algorithm in time.

Chapter 5

Project organization and Programming notes

All the source code files are saved in **SOURCE** directory.

- **SOURCE/algorithms** sub-directory saves 24 files, 22 algorithms' source code (.cpp and .hpp), and helpFunctions files contains some functions used in algorithms' source code.
- **SOURCE** directory contains the remain codes.
- No special library is used.
- Directory **script** contains scripts to make it easier to create files and build files. **They must be moved to root, mean outside SOURCE, to be used**.

There are two main files:

- **overview.cpp**: To run the experiments and give you an overview about algorithms.
  - Build: Stay in **SOURCE** directory and run **g++ algorithms/*.cpp DataGenerator.cpp helpler.cpp sort_execute.cpp overview.cpp -o overview.exe** (Do not include **sort.cpp** here).
  - Run: **./overview.exe** to run it. If you want to save the output to file *a.csv*, run **./overview.exe a.csv** instead.
- **overview.cpp**: To run the experiments and give you an overview about algorithms.
  - Build: Stay in **SOURCE** directory and run **g++ algorithms/*.cpp DataGenerator.cpp helpler.cpp sort_execute.cpp overview.cpp -o sort.exe** (Do not include **overview.cpp** here).

Chapter 6

References

1. Mr Nguyen Thanh Phuong's lectures.
2. https://www.geeksforgeeks.org/sorting-algorithms/
3. D. L. Shell. 1959. A high-speed sorting procedure. Commun. ACM 2, 7 (July 1959), 30–32. https://doi.org/10.1145/368370.368387
4. Neubert, Karl-Dietrich, "The Flashsort Algorithm," Dr. Dobb's Journal, p. 123, 1998.