

VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY



SORTING ALGORITHMS: LAB 3 REPORT DATA STRUCTURES AND ALGORITHMS

Lecturer: Dr. Nguyen Thanh Phuong

Teaching Assistant: Mr. Bui Huy Thong

Project Instructor: Mr. Bui Huy Thong

Class: 22TNT1TN

Student: 22120283 – Tran Huu Phuc

HO CHI MINH CITY, DECEMBER 2023

1 Information page

Name: Tran Huu Phuc

Student ID: 22120283

Class: 22TNT1TN

Subject: Data structures and algorithms

Lecturer: Doctor Nguyen Thanh Phuong

Teaching Assistant: Master Bui Huy Thong

Project Instructor: Master Bui Huy Thong

Topic: Sorting algorithms

2 Introduction page

I've successfully executed all 11 necessary algorithms, encompassing selection sort, insertion sort, bubble sort, shaker sort, shell sort, heap sort, merge sort, quick sort, counting sort, radix sort, and flash sort.

Regarding output requirements, I've fulfilled 5 out of 5 commands, comprising 3 for algorithm mode and 2 for comparison mode.

Following are the hardware specifications of the computer utilized for executing these algorithms:

Device name	LAPTOP-8FKCMAEB
Processor	11th Gen Intel(R) Core(TM) i5-11320H @ 3.20GHz 3.19 GHz
Installed RAM	16.0 GB (15.8 GB usable)
Device ID	1B87EE43-E014-433E-A70C-D3C0F0137FCA
Product ID	00356-24576-05711-AAOEM
System type	64-bit operating system, x64-based processor
Pen and touch	No pen or touch input is available for this display

Figure 1 – Hardware specifications

Contents

1	Information page	1
2	Introduction page	2
	tings	5
3	Algorithm presentation	7
3.1	Selection sort	7
3.2	Insertion sort	8
3.3	Bubble sort	8
3.4	Shaker sort (Cocktail sort)	9
3.5	Shell sort	10
3.6	Heap sort	10
3.6.1	Heap data structure	11
3.6.2	Build a min-heap	11
3.6.3	Build a max-heap	11
3.6.4	Pseudocodes	11
3.7	Merge sort	12
3.8	Quick sort	13
3.9	Counting sort	15
3.10	Radix sort	15
3.11	Flash sort	16
3.11.1	Stage 1: Classification of elements of the array	16
3.11.2	Stage 2: Partition of elements	17
3.11.3	Stage 3: Sort the elements in each partition	17
3.11.4	Complexity	18
4	Experimental results and comments	19
4.1	Tables of running time and comparisons count	19
4.2	Line graphs of running time	23
4.3	Bar charts of comparisons	25
4.4	Comments	27
5	Project organization and Programming notes	28
5.1	Project organization	28
5.2	Programming notes	28

Figures

1	Hardware specifications	2
2	Max-heap and min-heap.	11
3	Line graph of running time for randomized input	23
4	Line graph of running time for sorted input	23
5	Line graph of running time for reversed input	24
6	Line graph of running time for nearly sorted input	24
7	Bar chart of comparisons for randomized input	25
8	Bar chart of comparisons for sorted input	25
9	Bar chart of comparisons for reversed input	26
10	Bar chart of comparisons for nearly sorted input	26
11	Files in project	28

Tables

1	Data order: Randomized - table 1	19
2	Data order: Randomized - table 2	19
3	Data order: Sorted - table 1	20
4	Data order: Sorted - table 2	20
5	Data order: Reversed - table 1	21
6	Data order: Reversed - table 2	21
7	Data order: Nearly sorted - table 1	22
8	Data order: Nearly sorted - table 2	22

List of Codes

1	Selection sort	7
2	Insertion sort	8
3	Bubble sort	8
4	Shaker sort	9
5	Shell sort	10
6	Heap sort	11
7	Merge sort	12
8	Quick sort	14
9	Counting sort	15
10	Radix sort	15
11	Flash sort - stage 1	16
12	Flash sort - stage 2	17
13	Flash sort - stage 3	17

3 Algorithm presentation

In this section, I will present the algorithms implemented in the project: ideas, step-by-step descriptions, and complexity evaluations. Variants/improvements of an algorithm, if there is any, will be also mentioned.

In this project, sorting algorithms are only used to sort the array in ascending order. Sorting in descending order will be similar.

Most of pseudocodes in this section will be presented in Pascal, with the 1-base array.

3.1 Selection sort

Selection sort stands as one of the most straightforward sorting techniques. Its fundamental principle operates as follows:

Ideas:

- Initially, in the first iteration, the algorithm identifies the minimum element within the range $a[1..n]$ and exchanges it with $a[1]$, effectively placing the minimum value at the start of the array.
- Subsequently, in the second iteration, the algorithm pinpoints the minimum element within the range $a[2..n]$ and swaps it with $a[2]$, positioning the second smallest value in the array's second position.
- This process continues sequentially: during the i -th iteration, the algorithm selects the minimum element within the range $a[i..n]$ and exchanges it with $a[i]$.
- In the $(n-1)$ -th iteration, the algorithm discerns the lesser of the elements between $a[n-1]$ and $a[n]$, then swaps it with $a[n-1]$, arranging the second-to-last element appropriately within the sorted sequence."

Pseudocodes: [1]

Code 1 – Selection sort

```

for i from 1 to n - 1 do
    jmin = i
    for j from i + 1 to n do
        if a[j] < a[jmin] then
            jmin = j
        end if
    end for

    if jmin != i then
        swap(a[jmin], a[i])
    end if
end for

```

Time complexity: [3]

- Worst case: $O(n^2)$.
- Best case: $O(n^2)$.
- Average case: $O(n^2)$.

Space complexity: $O(1)$. [3]

3.2 Insertion sort

Ideas: Consider the array $a[1..n]$.

We see that the subarray with only one element $a[1]$ can be seen as sorted.

Consider $a[2]$, we compare it with $a[1]$, if $a[2] \leq a[1]$, we insert it before $a[1]$.

With $a[3]$, we compare it with the sorted subarray $a[1..2]$, find the position to insert $a[3]$ to that subarray to have an ascending order.

In a general speech, we will sort the array $a[1..k]$ if the array $a[1..k-1]$ is already sorted by inserting $a[k]$ to the appropriate position.

Pseudocodes: [1]

Code 2 – Insertion sort

```
for i from 2 to n do
    temp = a[i]
    j = i - 1
    while j > 0 and temp < a[j] do
        a[j + 1] = a[j]
        j = j - 1
    end while
    a[j + 1] = temp
end for
```

Time complexity: [4]

- Worst case: $O(n^2)$.
- Best case: $O(n)$, in case the array is already sorted.
- Average case: $O(n^2)$.

Space complexity: $O(1)$. [4]

Improvements:

- Binary insertion sort – find the position to insert using binary search, which reduces the number of comparisons. Details at link: [5].
- Another improvement of insertion sort is shell sort, which will be presented in section 3.5

3.3 Bubble sort

Ideas: Bubble sort is the simplest sorting algorithm, which swaps the adjacent elements if they are in wrong order, repeatedly n times.

After the i -th turn, the i -th smallest element will be swapped to position i .

Pseudocodes: [1]

Code 3 – Bubble sort

```
for i from 2 to n do
    for j from n downto i do
        if a[j - 1] > a[j] then
            swap(a[j - 1], a[j])
        end if
    end for
end for
```

Time complexity: $O(n^2)$, not mentioned how the input data is. [1]

Space complexity: $O(1)$. [4]

Variations: There are some variations in the implementation.

- Instead of top-down with j , we can iterate from the bottom up, from $i + 1$ to n .
- Another variation is j iterates from 1 to $n - i$. This is the version that I choose in my project.

Improvements: An improvement of bubble sort is shaker sort, which we will research in section 3.4.

3.4 Shaker sort (Cocktail sort)

Ideas: Shaker sort, known as cocktail sort or bi-directional bubble sort, enhances the traditional bubble sort method. While bubble sort operates by traversing elements exclusively from left to right, shaker sort differs by traversing in both directions—first from left to right and then from right to left—alternating between the two directions.

Pseudocode: [2]

Code 4 – Shaker sort

```

left = 2
right = n
k = n
repeat
    for j from right downto left do
        if a[j - 1] > a[j] then
            swap(a[j - 1], a[j])
            k = j
        end if
    end for
    left = k + 1
    for j from left to right do
        if a[j - 1] > a[j] then
            swap(a[j - 1], a[j])
            k = j
        end if
    end for
    right = k - 1
until left > right

```

Time complexity: [7]

- Worst case: $O(n^2)$.
- Best case: $O(n)$, in case the array is already sorted.
- Average case: $O(n^2)$.

Space complexity: $O(1)$. [7]

3.5 Shell sort

A drawback of insertion sort is that we always have to insert an element to a position near the beginning of the array. In that case, we use shell sort.

Ideas: Consider an array $a[1..n]$. For an integer $h : 1 \leq h \leq n$, we can divide the array into h subarrays:

- Subarray 1: $a[1], a[1 + h], a[1 + 2h] \dots$
- Subarray 2: $a[2], a[2 + h], a[2 + 2h] \dots$
- ...
- Subarray h : $a[h], a[2h], a[3h] \dots$

Those subarrays are called subarrays with step h . With a step h , shell sort will use insertion sort for independent subarrays, then similarly with $\frac{h}{2}, \frac{h}{4}, \dots$ until $h = 1$.

Pseudocodes:

Code 5 – Shell sort

```
gap = n div 2
while gap > 0 do
  for i from gap to n do
    j = i - gap
    k = a[i]
    while j >= 0 and a[j] > k do
      a[j + gap] = a[j]
      j = j - gap
    end while
    a[j + gap] = k
  end for
  gap = gap div 2
end while
```

Time complexity: [8]

- Worst case: $O(n^2)$.
- Best case: $O(n \log n)$.
- Average case: depends on the gap sequence.

Space complexity: $O(1)$. [8]

3.6 Heap sort

In 1981, J. W. J. Williams introduced the heap sort algorithm, which not only presented an efficient sorting method but also established a crucial data structure for priority queues: the heap data structure. This algorithm not only aids in swiftly sorting arrays but also opens up a broad spectrum for managing and utilizing priorities in programming.

3.6.1 Heap data structure

A heap represents a distinctive form of binary tree. For a binary tree to adhere to the heap data structure:

- It must embody the characteristics of a complete binary tree.
- All nodes within the tree must adhere to a specific rule where they are either greater than their children, creating a max-heap with the largest element as the root, or smaller than their children, forming a min-heap.[9]

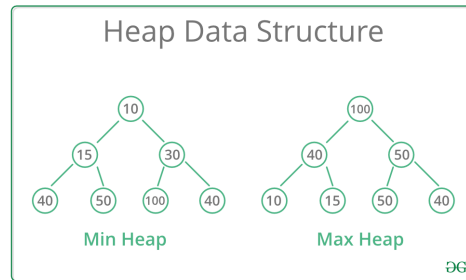


Figure 2 – Max-heap and min-heap.

3.6.2 Build a min-heap

To build a min heap, we: [10]

- Create a new child node at the end of the heap (last level).
- Add the new key to that node (append it to the array).
- Move the child up until we reach the root node and the heap property is satisfied.

To remove/delete a root node in a min heap, we: [10]

- Delete the root node.
- Move the key of last child to root.
- Compare the parent node with its children.
- If the value of the parent is greater than its children, swap them, and repeat until the heap property is satisfied.

3.6.3 Build a max-heap

Building a max-heap is similar to building a min-heap.

3.6.4 Pseudocodes

Code 6 – Heap sort

```
procedure heapify(a[], n, i)
    max = i
    left = 2 * i
```

```

    right = 2 * i + 1

    if left <= n and a[left] > a[max] then
        max = left
    if right <= n and a[right] > a[max] then
        max = right

    if max != i then
        swap(a[i], a[max])
        heapify(a, n, max)

procedure heapsort(a[], n)
    for i from n div 2 - 1 downto 1 do
        heapify(a, n, i)

    for i from n downto 1 do
        swap(a[1], a[i])
        heapify(a, i - 1, 1)

```

Time complexity: [9]

- Worst case: $O(n \log n)$.
- Best case: $O(n \log n)$.
- Average case: $O(n \log n)$.

Space complexity: $O(1)$. [9]

3.7 Merge sort

Merge sort is a divide-and-conquer algorithm that was invented by John von Neumann in 1945. This is one of the most popular sorting algorithms.

Ideas:

- Split the array into two subarrays at its midpoint.
- Attempt to sort both subarrays. If the base case hasn't been reached, proceed to further divide these subarrays..
- Merge the sorted subarrays.

Pseudocodes:

Code 7 – Merge sort

```

procedure mergeSort(a[], n)
    if n <= 1 then
        return

    mid = n div 2
    left[1..mid]
    right[1..n - mid]

    for i from 1 to mid do
        left[i] = a[i]

```

```

end for
for j from 1 to n - mid do
    right[j] = a[mid + j]
end for

mergeSort(left, mid)
mergeSort(right, n - mid)

i = 1
j = 1
k = 1
while i <= mid and j <= n - mid do
    if left[i] < right[j] then
        a[k] = left[i]
        k = k + 1
        i = i + 1
    else
        a[k] = right[j]
        k = k + 1
        j = j + 1
    end if
end while

while i <= mid do
    a[k] = left[i]
    k = k + 1
    i = i + 1
end while

while j <= n - mid do
    a[k] = right[j]
    k = k + 1
    j = j + 1
end while
end

```

Time complexity: [11]

- Worst case: $O(n \log n)$.
- Best case: $O(n \log n)$.
- Average case: $O(n \log n)$.

Space complexity: $O(n)$. [11]

3.8 Quick sort

Quicksort is a divide-and-conquer algorithm, introduced by C. A. R. Hoare, an English computer scientist, in 1960. It has become widely used due to its efficient, and is now one of the most popular sorting algorithms.

Ideas:

- Sorting the array $a[1..n]$ can be seen as sorting the segment from index 1 to index n of that array.

- To sort a segment, if that segment has less than 2 elements, then we have to do nothing, else we choose a random element to be the "pivot". All elements that are less than pivot will be arranged to a position before pivot, and all ones that are greater than pivot will be arranged to a position after pivot.
- After that, the segment is divided into two segments, all elements in the first segment are less than pivot, and all elements in the second segment are greater than pivot. And now we have to sort two new segments, which have lengths smaller than the length of the initial segment.

I will choose the middle elements to be the pivot to implement the algorithm.

Pseudocodes: [2].

Code 8 – Quick sort

```

procedure partition(a[], l, r)
    mid = (l + r) div 2
    pivot = a[mid]
    i = l - 1
    j = r + 1

    repeat
        repeat
            inc(i)
        until a[i] >= pivot

        repeat
            dec(j)
        until a[j] <= pivot

        if i < j then
            swap(a[i], a[j])
        end if
    until i >= j

    swap(a[mid], a[j])
    return j
end

procedure quicksort(a[], l, r)
    if l < r then
        s = partition(a, l, r)
        quicksort(a, l, s - 1)
        quicksort(a, s + 1, r)
    end if
end

```

Time complexity: [12]

- Worst case: $O(n^2)$.
- Best case: $O(n \log n)$.
- Average case: $O(n \log n)$.

Space complexity: $O(1)$. [12]

Variations: Above is the implementation of quicksort using recursion. There is also an iterative algorithms: [13]

3.9 Counting sort

Counting sort is a sorting algorithm working by counting the number of objects having distinct key values (a kind of hashing). [14]

Ideas: Iterate through the input, count the number of times each item occurs, then use those results to calculate an item's index in the sorted array. [15]

Pseudocodes: [2]

Code 9 – Counting sort

```

procedure countingsort(a[], n)
    u = maximum_value_in(a)
    f[0..u] := {0}
    b[1..n]

    for i from 1 to n do
        inc(f[a[i]])

    for i from 1 to u do
        f[i] := f[i - 1] + f[i]

    for i from n downto 1 do
        b[f[a[i]]] := a[i]
        dec(f[a[i]])

    for i from 1 to n do
        a[i] := b[i]
end

```

Counting sort works well when $n \approx u$, but it will be "disastrous" if $u \gg n$. [2]

Time complexity: $O(n + u)$. [14]

Space complexity: $O(n + u)$. [14]

3.10 Radix sort

Like counting sort mentioned in section 3.9, radix sort only works with integer.

Ideas: sort the array using counting sort (or any stable algorithms) according to the i -th digit. [16]

Let d be the maximum number of digits of elements in the array, and b be the base used to represent array, for example, for decimal system, $b = 10$.

Pseudocodes: [2]

Code 10 – Radix sort

```

procedure sort(a[], k)
    b = maximum_value_in(a)
    f[0..b - 1] := {0}
    n = length_of(a)
    b[1..n]

```



```

    for i from 1 to n do
        inc(f[digit(a[i], k)])

    for i from 1 to b - 1 do
        f[i] := f[i] + f[i - 1]

    for i from n downto 1 do
        j = digit(a[i], k)
        b[f[j]] = a[i]
        dec(f[j])

    a = b

procedure LSDradixsort(a[], d)
    for k from 0 to d do
        sort(a, k)

```

Time complexity: $O(d(n + b))$. [16]

Space complexity: $O(n)$. [16]

3.11 Flash sort

Flash sort is a distribution sorting algorithm, which has the time complexity approximately linear complexity. [17] Flash sort was invented by Dr. Neubert in 1997. He named the algorithm "flash" sort because he was confident that this algorithm is very fast.

Ideas: The algorithm is divided into three stages. [2] [18]

- Stage 1: Classification of elements of the array.
- Stage 2: Partition of elements.
- Stage 3: Sort the elements in each partition.

3.11.1 Stage 1: Classification of elements of the array

Let m be the number of classes. The element a_i will be in the k -th class with:

$$k_{a_i} = \left\lfloor \frac{(m - 1)(a_i - \min_a)}{\max_a - \min_a} \right\rfloor + 1.$$

Pseudocodes: [2]

Code 11 – Flash sort - stage 1

```

L[1..m] := {0}
for i from 1 to n do
    k = (m - 1) * (a[i] - min) div (max - min)
    inc(L[k])

for k from 2 to m do
    L[k] := L[k] + L[k - 1]

```

After this stage, $L[k]$ will point to the right boundary of the k -th class.

3.11.2 Stage 2: Partition of elements

The elements are sorted by *in-situ permutation*. During the permutation, the $L[k]$ are decremented by a unit step at each new placement of an element of class k . A crucial aspect of this algorithm is identifying new cycle leaders. A cycle ends, if the vector $L[k]$ points to the position of an element below boundary of class k . The new cycle leader is the element situated in the lowest position complying to the complimentary condition, i.e. for which $L[k]$ points to a position with $i \leq L_{k_{a_i}}$. [18]

Pseudocodes: [2]

Code 12 – Flash sort - stage 2

```
count := 1
i := 1
k := m

while (count <= n) do
  while (i > L[k]) do
    inc(i)
    k := (m - 1) * (a[i] - min) div (max - min) + 1

  x := a[i]

  while (i <= L[k]) do
    k := (m - 1) * (x - min) div (max - min) + 1
    y := a[L[k]]
    a[L[k]] := x
    x := y
    dec(L[k])
  inc(count)
```

3.11.3 Stage 3: Sort the elements in each partition

A small number of partially distinguishable elements are sorted locally within their classes either by recursion or by a simple conventional sort algorithm. [18]

In this project, I will choose insertion sort for this stage.

Pseudocodes: [2]

Code 13 – Flash sort - stage 3

```
for k := 2 to m do
begin
  for i := L[k] - 1 to L[k - 1] do
  begin
    if (a[i] > a[i + 1]) then
    begin
      t := a[i];
      j := i;
      while (t > a[j + 1]) do
      begin
        a[j] := a[j + 1];
        inc(j);
      end
      a[j] := t;
```

```
    end  
  end  
end
```

This code is written correctly because the last class only contains of maximum element of the array, therefore it has been already sorted.

3.11.4 Complexity

Time complexity: $O\left(\frac{n^2}{m}\right)$.

Experiments has shown that $m \approx 0.43n$ will be the best for this algorithm. In that case, time complexity of the algorithm is linear. [2]

Space complexity: $O(m)$.

4 Experimental results and comments

4.1 Tables of running time and comparisons count

Data order: Randomized						
Data size	10000		30000		50000	
Resulting Statics	Run time	Comparisons	Run time	Comparisons	Run time	Comparisons
Selection Sort	0.0560	100009999	0.4900	900029999	1.3120	2500049999
Insertion Sort	0.0410	50218472	0.3260	450097717	0.9240	1247555741
Bubble Sort	0.1170	100009999	1.6030	900029999	4.5260	2500049999
Shaker Sort	0.1250	100005001	1.3100	900015001	3.9570	2500025001
Shell Sort	0.0010	664515	0.0050	2273811	0.0110	4488190
Heap Sort	0.0000	637900	0.0040	2150296	0.0070	3772581
Merge Sort	0.0010	582598	0.0060	1935819	0.0120	3377488
Quick Sort	0.0010	275718	0.0030	926644	0.0060	1571349
Counting Sort	0.0010	70002	0.0010	209999	0.0000	298306
Radix Sort	0.0000	140056	0.0020	510070	0.0020	850070
Flash Sort	0.0010	92093	0.0010	264060	0.0010	463640

Table 1 – Data order: Randomized - table 1

Data order: Randomized						
Data size	100000		300000		500000	
Resulting Statics	Run Time	Comparisons	Run Time	Comparisons	Run Time	Comparisons
Selection Sort	5.6780	10000099999	47.8400	90000299999	135.2290	250000499999
Insertion Sort	4.1240	5007613355	31.8300	45014934014	87.3680	125116378803
Bubble Sort	19.6470	10000099999	182.0200	90000299999	518.9700	250000499999
Shaker Sort	15.1760	10000050001	141.3240	90000150001	402.7320	250000250001
Shell Sort	0.0180	10163942	0.0620	34112628	0.1100	65477662
Heap Sort	0.0160	8042832	0.0560	26487198	0.1000	45970601
Merge Sort	0.0210	7154775	0.0640	23360316	0.1090	40368749
Quick Sort	0.0010	3307775	0.0280	10587994	0.0460	18433094
Counting Sort	0.0010	498306	0.0010	1298306	0.0020	2098306
Radix Sort	0.0040	1700070	0.0140	5100070	0.0230	8500070
Flash Sort	0.0030	836196	0.0090	2742244	0.0200	4367656

Table 2 – Data order: Randomized - table 2

Data order: Sorted						
Data size	10000		30000		50000	
Resulting Statics	Run Time	Comparisons	Run Time	Comparisons	Run Time	Comparisons
Selection Sort	0.0540	100009999	0.4750	900029999	1.3200	2500049999
Insertion Sort	0.0000	29998	0.0000	89998	0.0000	149998
Bubble Sort	0.0600	100009999	0.5220	900029999	1.4750	2500049999
Shaker Sort	0.0530	100005001	0.4840	900015001	1.3440	2500025001
Shell Sort	0.0000	360042	0.0010	1170050	0.0020	2100049
Heap Sort	0.0010	670329	0.0030	2236648	0.0060	3925351
Merge Sort	0.0010	466442	0.0050	1543466	0.0070	2683946
Quick Sort	0.0000	154959	0.0010	501929	0.0010	913850
Counting Sort	0.0000	70002	0.0000	210002	0.0000	350002
Radix Sort	0.0000	140056	0.0010	510070	0.0020	850070
Flash Sort	0.0000	112902	0.0010	338702	0.0010	564502

Table 3 – Data order: Sorted - table 1

Data order: Sorted						
Data size	100000		300000		500000	
Resulting Statics	Run Time	Comparisons	Run Time	Comparisons	Run Time	Comparisons
Selection Sort	5.2900	10000099999	49.1590	90000299999	135.4670	250000499999
Insertion Sort	0.0000	299998	0.0000	899998	0.0010	1499998
Bubble Sort	6.2680	10000099999	54.2010	90000299999	152.9830	250000499999
Shaker Sort	5.5610	10000050001	49.3240	90000150001	145.7010	250000250001
Shell Sort	0.0030	4500051	0.0120	15300061	0.0200	25500058
Heap Sort	0.0120	8365080	0.0440	27413230	0.0750	47404886
Merge Sort	0.0150	5667898	0.0440	18408314	0.0760	31836410
Quick Sort	0.0020	1927691	0.0050	6058228	0.0090	10310733
Counting Sort	0.0010	700002	0.0020	2100002	0.0040	3500002
Radix Sort	0.0050	1700070	0.0190	6000084	0.0340	10000084
Flash Sort	0.0020	1129002	0.0060	3387002	0.0110	5645002

Table 4 – Data order: Sorted - table 2

Data order: Reversed						
Data size	10000		30000		50000	
Resulting Statics	Run Time	Comparisons	Run Time	Comparisons	Run Time	Comparisons
Selection Sort	0.0580	100009999	0.5180	900029999	1.4260	2500049999
Insertion Sort	0.0720	100009999	0.6800	900029999	1.8370	2500049999
Bubble Sort	0.1480	100009999	1.4030	900029999	3.8440	2500049999
Shaker Sort	0.1430	100005001	1.3080	900015001	3.5960	2500025001
Shell Sort	0.0000	475175	0.0010	1554051	0.0030	2844628
Heap Sort	0.0010	606771	0.0040	2063324	0.0050	3612724
Merge Sort	0.0010	485241	0.0030	1589913	0.0080	2772825
Quick Sort	0.0000	164975	0.0000	531939	0.0010	963861
Counting Sort	0.0000	70002	0.0000	210002	0.0000	350002
Radix Sort	0.0000	140056	0.0020	510070	0.0030	850070
Flash Sort	0.0000	97855	0.0010	293555	0.0010	489255

Table 5 – Data order: Reversed - table 1

Data order: Reversed						
Data size	100000		300000		500000	
Resulting Statics	Run Time	Comparisons	Run Time	Comparisons	Run Time	Comparisons
Selection Sort	5.8960	10000099999	51.6470	90000299999	148.1630	250000499999
Insertion Sort	7.1940	10000099999	66.6410	90000299999	179.3650	250000499999
Bubble Sort	15.2130	10000099999	139.5000	90000299999	380.6030	250000499999
Shaker Sort	14.5210	10000050001	132.7560	90000150001	357.0370	250000250001
Shell Sort	0.0060	6089190	0.0170	20001852	0.0300	33857581
Heap Sort	0.0110	7718943	0.0420	25569379	0.0690	44483348
Merge Sort	0.0160	5845657	0.0430	18945945	0.0760	32517849
Quick Sort	0.0020	2027703	0.0060	6358249	0.0100	10810747
Counting Sort	0.0010	700002	0.0020	2100002	0.0030	3500002
Radix Sort	0.0060	1700070	0.0210	6000084	0.0310	10000084
Flash Sort	0.0010	978505	0.0060	2935505	0.0100	4892505

Table 6 – Data order: Reversed - table 2

Data order: Nearly sorted						
Data size	10000		30000		50000	
Resulting Statics	Run Time	Comparisons	Run Time	Comparisons	Run Time	Comparisons
Selection Sort	0.0580	100009999	0.4770	900029999	1.4450	2500049999
Insertion Sort	0.0000	189566	0.0000	445094	0.0010	465754
Bubble Sort	0.0590	100009999	0.5330	900029999	1.4300	2500049999
Shaker Sort	0.0570	100005001	0.5180	900015001	1.3420	2500025001
Shell Sort	0.0000	409331	0.0020	1312748	0.0000	2231457
Heap Sort	0.0160	669847	0.0030	2236609	0.0060	3925543
Merge Sort	0.0010	496049	0.0070	1646576	0.0070	2787968
Quick Sort	0.0000	155010	0.0020	501973	0.0000	913882
Counting Sort	0.0000	70002	0.0000	210002	0.0000	350002
Radix Sort	0.0000	140056	0.0020	510070	0.0060	850070
Flash Sort	0.0000	112875	0.0000	338675	0.0040	564477

Table 7 – Data order: Nearly sorted - table 1

Data order: Nearly sorted						
Data size	100000		300000		500000	
Resulting Statics	Run Time	Comparisons	Run Time	Comparisons	Run Time	Comparisons
Selection Sort	5.3020	10000099999	46.9550	90000299999	135.3920	250000499999
Insertion Sort	0.0000	883142	0.0010	1321790	0.0010	1896770
Bubble Sort	6.1740	10000099999	51.8050	90000299999	147.7110	250000499999
Shaker Sort	5.3000	10000050001	48.0920	90000150001	136.8880	250000250001
Shell Sort	0.0000	4656753	0.0120	15451527	0.0210	25667234
Heap Sort	0.0120	8364623	0.0410	27413042	0.0730	47405018
Merge Sort	0.0160	5761629	0.0430	18504184	0.0720	31928266
Quick Sort	0.0010	1927715	0.0060	6058272	0.0090	10310773
Counting Sort	0.0010	700002	0.0020	2100002	0.0030	3500002
Radix Sort	0.0050	1700070	0.0180	6000084	0.0320	10000084
Flash Sort	0.0020	1128980	0.0060	3386979	0.0100	5644981

Table 8 – Data order: Nearly sorted - table 2

4.2 Line graphs of running time

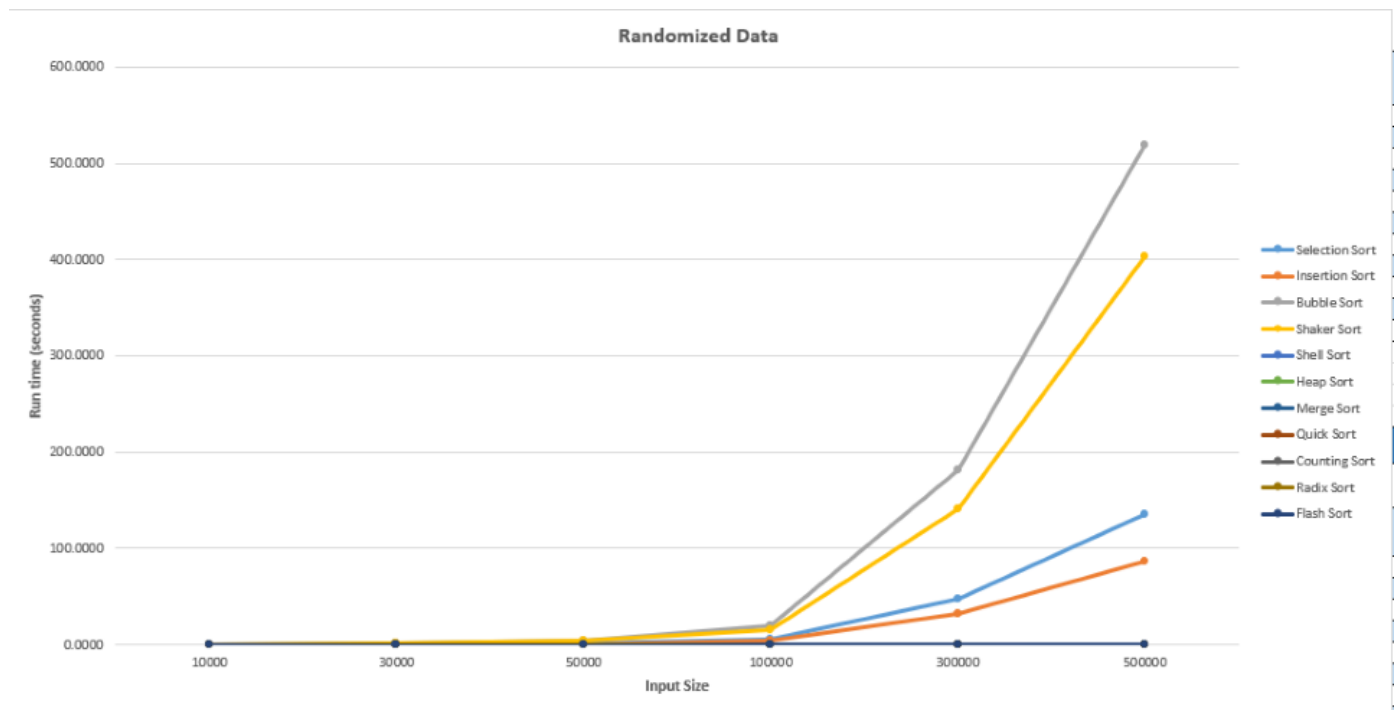


Figure 3 – Line graph of running time for randomized input

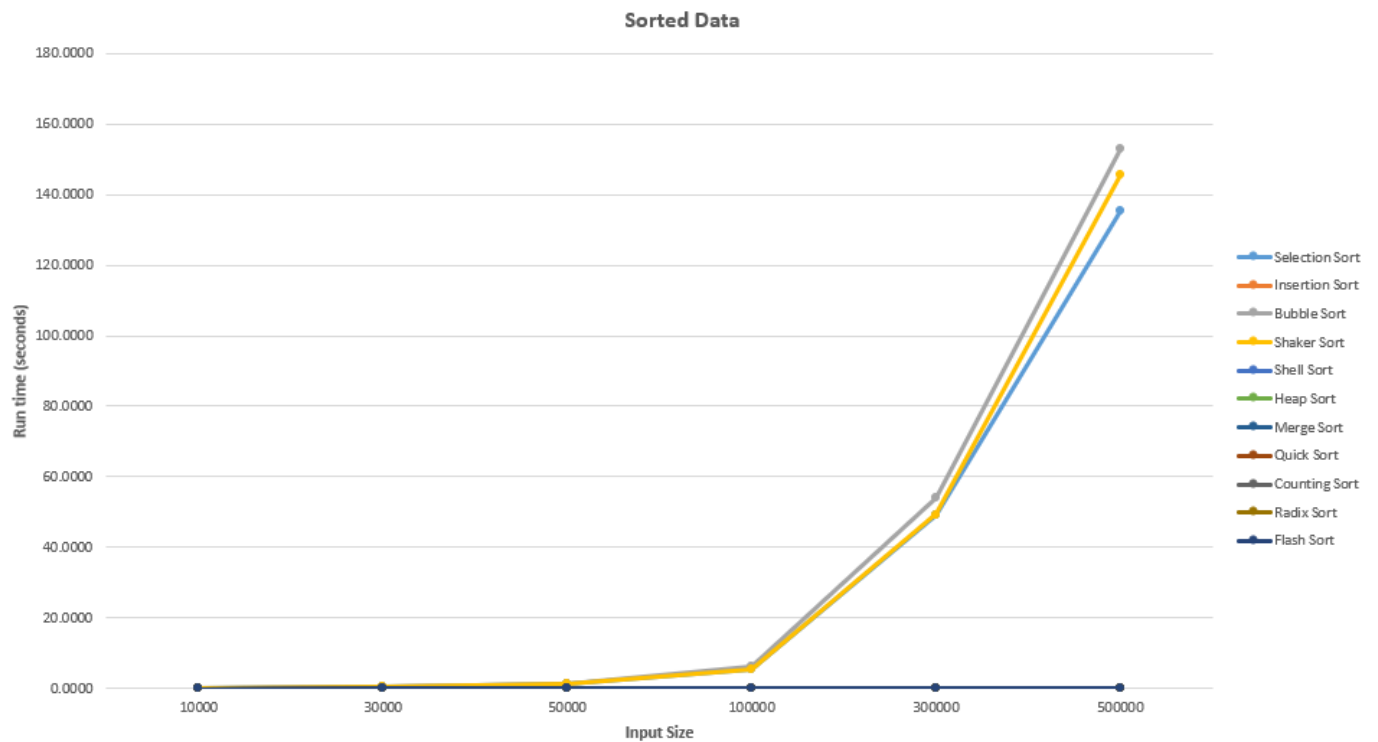


Figure 4 – Line graph of running time for sorted input

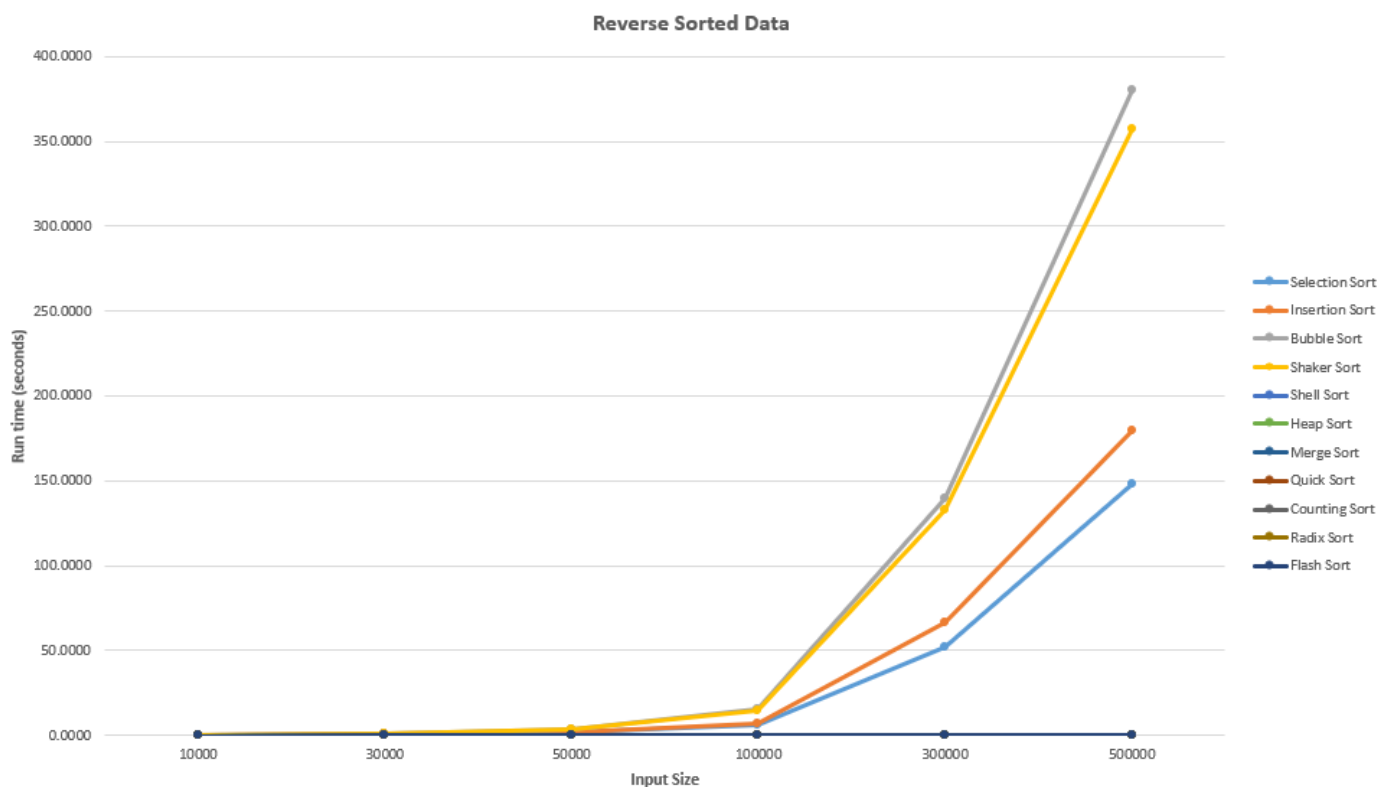


Figure 5 – Line graph of running time for reversed input

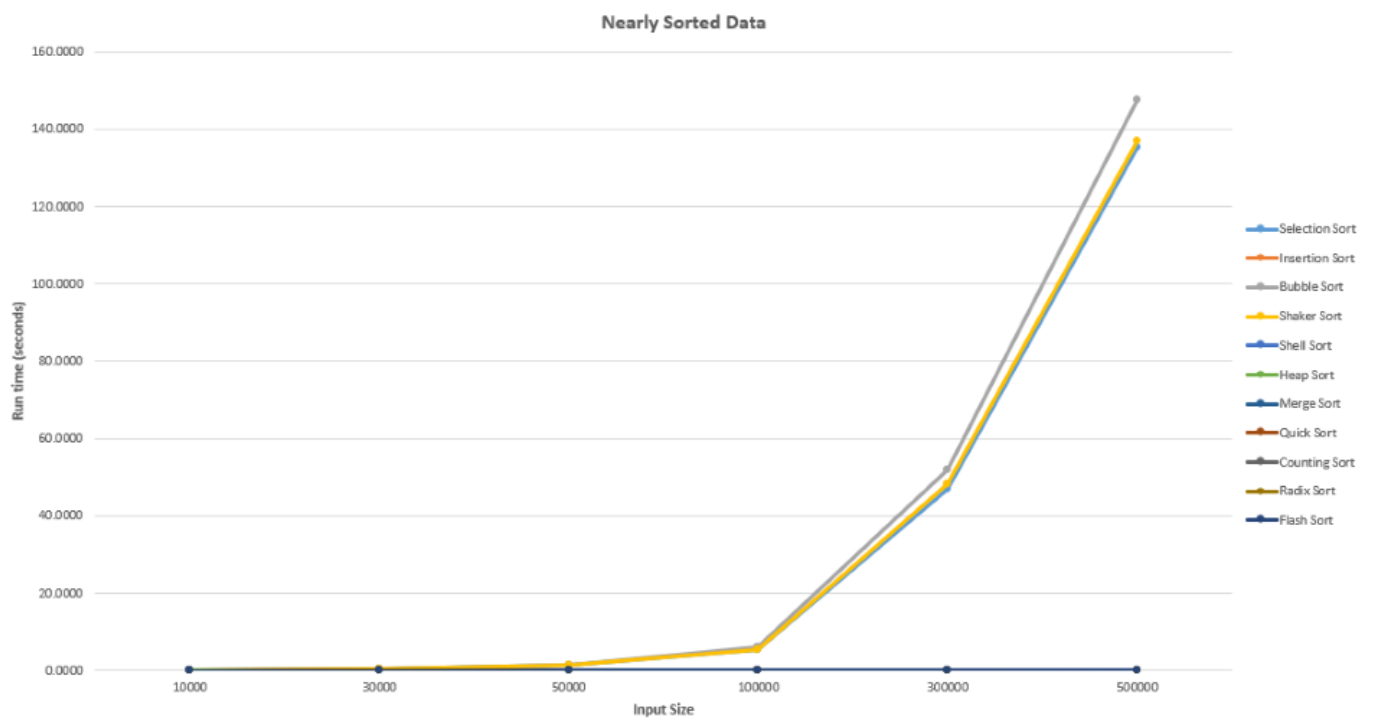


Figure 6 – Line graph of running time for nearly sorted input

4.3 Bar charts of comparisons

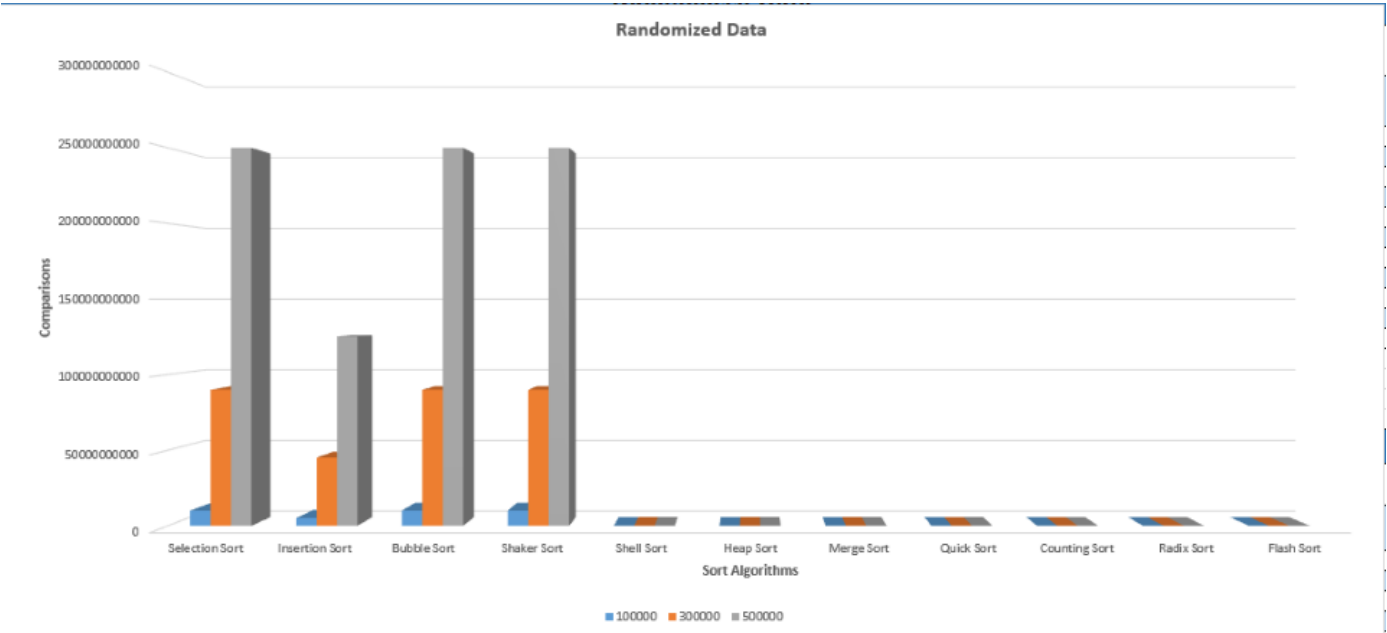


Figure 7 – Bar chart of comparisons for randomized input

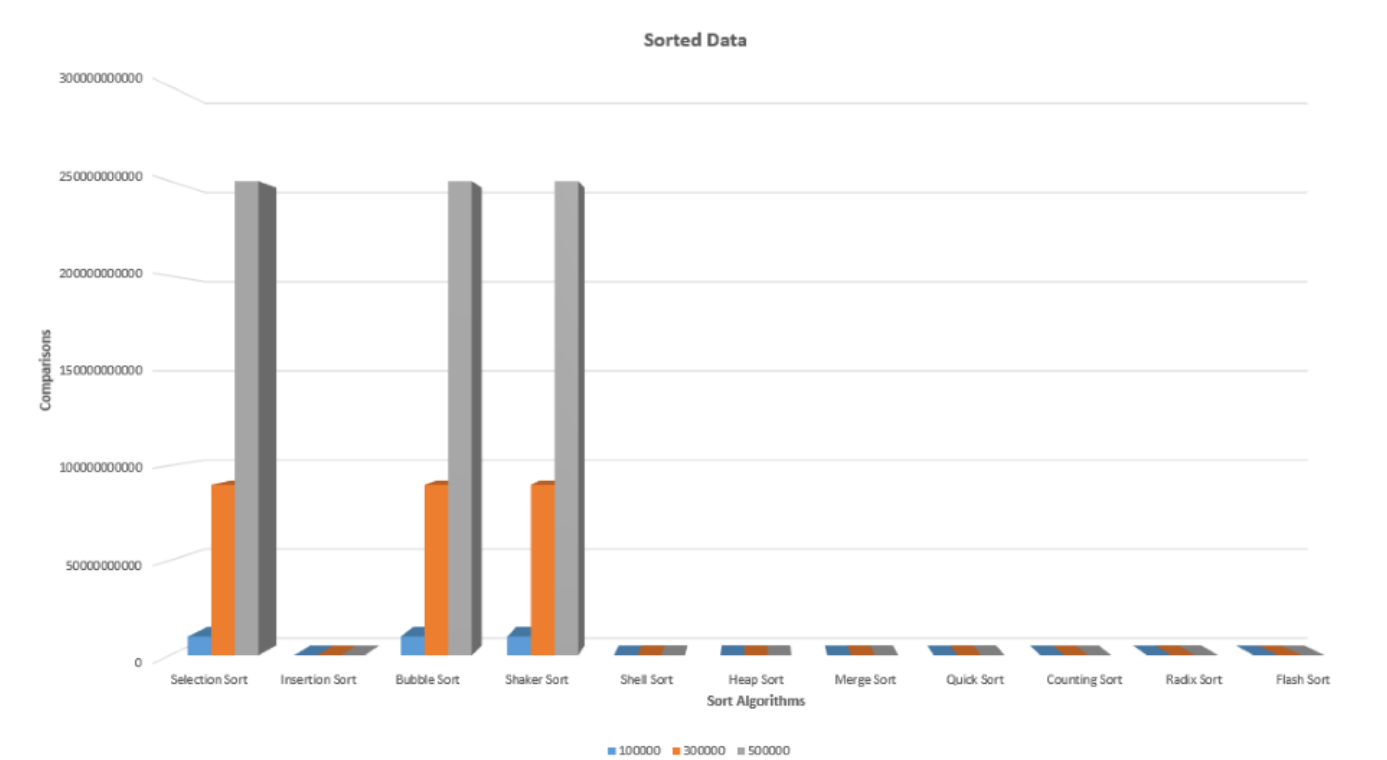


Figure 8 – Bar chart of comparisons for sorted input

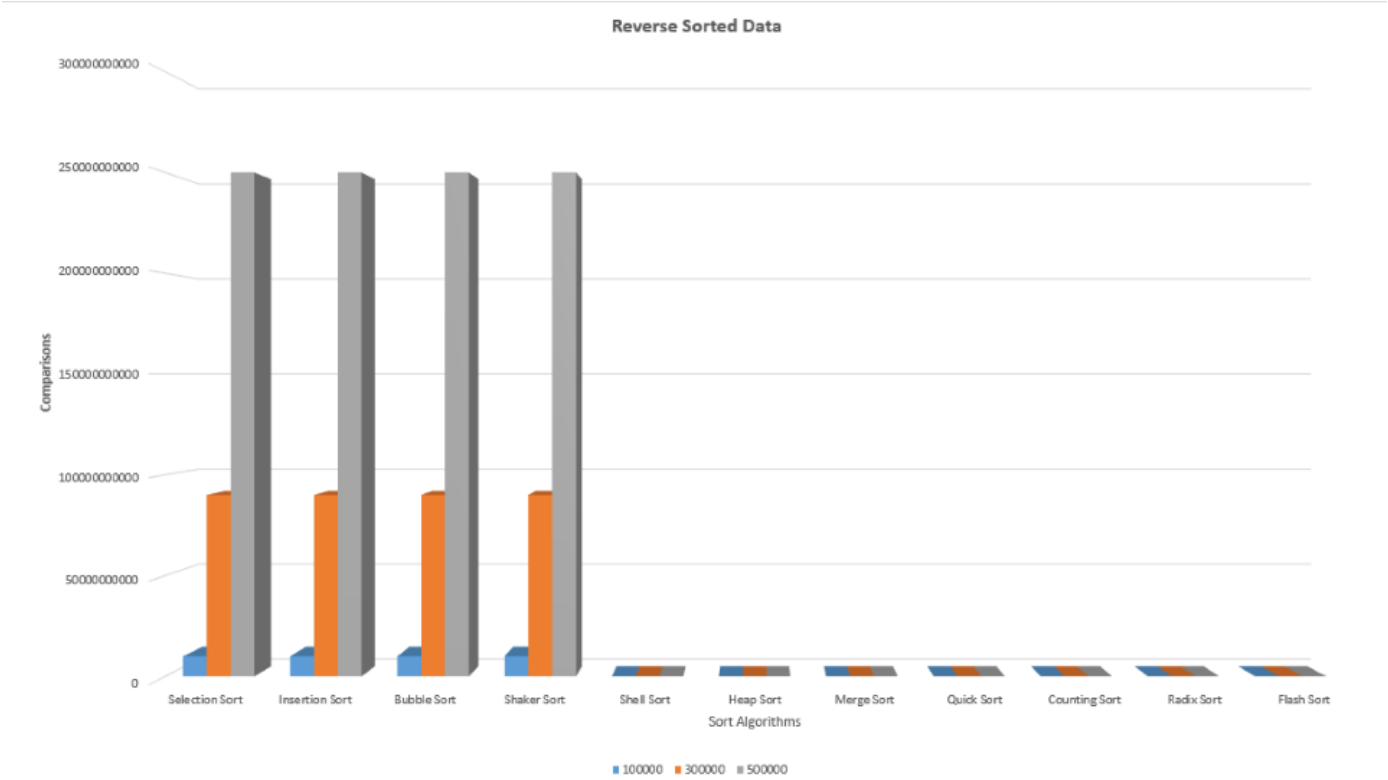


Figure 9 – Bar chart of comparisons for reversed input

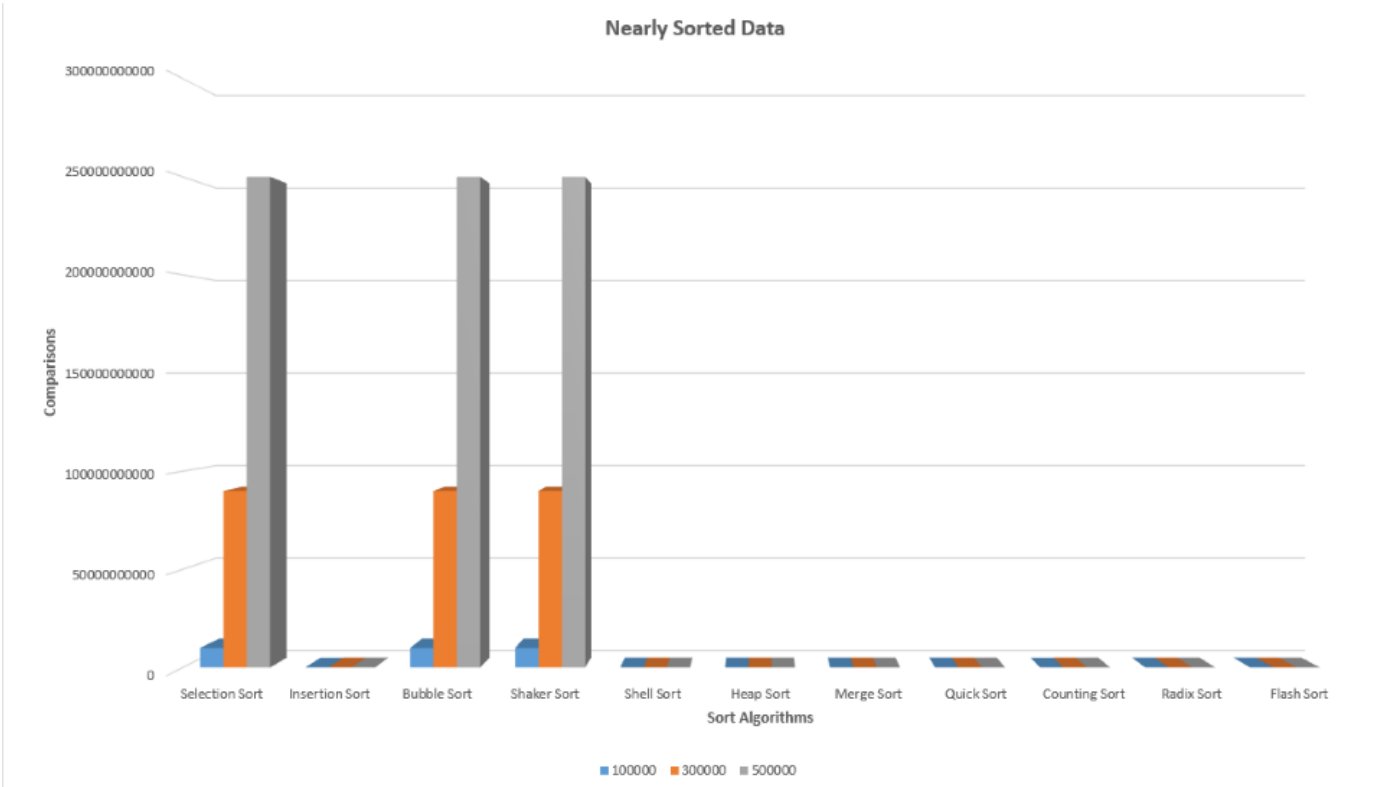


Figure 10 – Bar chart of comparisons for nearly sorted input

4.4 Comments

Due to my own observations on the graphs and charts above, I can claim that flash sort is the fastest and the most effective algorithm, because I chose $m \approx 0.43n$, to have the best performance. Moreover, bubble sort is the slowest and the least effective in almost all cases of input because of its giant number of comparisons.

Besides, sorting algorithms that do not base on comparisons like counting sort, radix sort or flash sort have used less comparisons than others, significantly less.

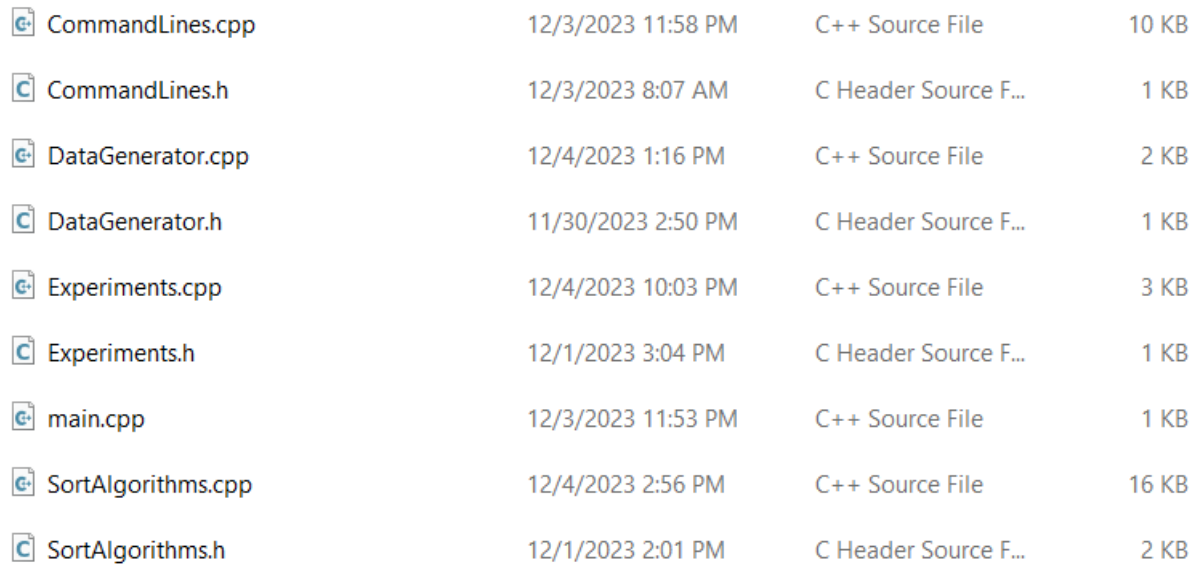
It can be seen obviously that in most algorithms, sorted input will cause to least running time. Here I will group algorithms due to its stability:

- Stable algorithms:
 - Selection sort. It uses approximate running times with different input orders.
 - Shell sort. I think it has shown its stability for various input orders.
 - Heap sort. The most important part of heap sort is building a heap, which requires the same cost although there are many of input orders.
 - Merge sort. Like heap sort, merge does the same thing for all input orders.
 - Radix sort. On integers, I think radix sort is fine, as the experimental results have shown.
 - Flash sort. With the complexity is approximate linear complexity, I think flash sort is an effective and stable algorithm.
- Unstable algorithms:
 - Insertion sort. It has shown unefficiency when the data is reversed or randomized, meanwhile it is very fast when the data is already sorted or nearly sorted.
 - Bubble sort. However it is always slow when the size of input is large, it has noticeably different running times for different input orders.
 - Shaker sort. Like insertion sort, it works extremely fast when the data is sorted or nearly sorted, but has bad performance when the data is random or reversed.
 - Quick sort. Although the graphs have shown that quicksort uses small number of comparisons and runs in a very short period of time, I will say that it is unstable, because the efficiency of this algorithm varies with the way we select the pivot, for example, if the input data is already sorted and we choose the leftmost elements of each segments to be pivots, it will be "disastrous".
 - Counting sort. Results on these input have shown that counting sort works fast, but as I mentioned in 3.9, it will be very ineffective when $u \gg n$.

5 Project organization and Programming notes

5.1 Project organization

Figure below shows files in my project.












 CommandLines.cpp	12/3/2023 11:58 PM	C++ Source File	10 KB
 CommandLines.h	12/3/2023 8:07 AM	C Header Source F...	1 KB
 DataGenerator.cpp	12/4/2023 1:16 PM	C++ Source File	2 KB
 DataGenerator.h	11/30/2023 2:50 PM	C Header Source F...	1 KB
 Experiments.cpp	12/4/2023 10:03 PM	C++ Source File	3 KB
 Experiments.h	12/1/2023 3:04 PM	C Header Source F...	1 KB
 main.cpp	12/3/2023 11:53 PM	C++ Source File	1 KB
 SortAlgorithms.cpp	12/4/2023 2:56 PM	C++ Source File	16 KB
 SortAlgorithms.h	12/1/2023 2:01 PM	C Header Source F...	2 KB

Figure 11 – Files in project

- CommandLines files include Command line arguments functions.
- DataGenerator files include Input data generator functions.
- Experiments files include Experiments functions to take note for statistics.
- Sort Algorithms files include all Sort functions used in this project.

5.2 Programming notes

My project does not use any special libraries or data structures. All are included in basic C++ 20.

References

- [1] Le Minh Hoang (2002) *Giai thuat va lap trinh*, Ha Noi University of Education Press
- [2] Lectures from Dr. Nguyen Thanh Phuong
- [3] <https://deviot.vn/tutorials/c-co-ban.78025672/thuat-toan-sap-xep-lua-chon.80502462>
- [4] <https://www.geeksforgeeks.org/analysis-of-different-sorting-techniques>
- [5] <https://www.geeksforgeeks.org/binary-insertion-sort/>
- [6] <https://viblo.asia/p/thuat-toan-sap-xep-noi-bot-bubble-sort-m68Z0exQlkG>
- [7] <https://daynhauhoc.com/t/cau-truc-du-lieu-va-giai-thuat-giai-thuat-sap-xep-rung-lac-126359>
- [8] <https://www.tutorialspoint.com/Shell-Sort>
- [9] <https://www.geeksforgeeks.org/heap-sort/>
- [10] <https://www.educative.io/blog/data-structure-heaps-guide>
- [11] <https://www.geeksforgeeks.org/merge-sort/>
- [12] <https://freetuts.net/thuat-toan-sap-xep-nhanh-quick-sort-2940.html>
- [13] <https://www.geeksforgeeks.org/iterative-quick-sort/>
- [14] <https://viblo.asia/p/sap-xep-bang-dem-phan-phoi-counting-sort-Qbq5Q63LKD8>
- [15] <https://www.interviewcake.com/concept/java/counting-sort>
- [16] <https://viblo.asia/p/tim-hieu-ve-radix-sort-va-cach-implement-thuat-toan-nay-trong->
- [17] <https://www.w3resource.com/javascript-exercises/searching-and-sorting-algorithm/searching-and-sorting-algorithm-exercise-12.php>
- [18] <https://www.neubert.net/FS0Intro.html>