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

REPORT FOR LAB 3: SORTING

DATA STRUCTURES AND ALGORITHMS

TOPIC: SORTING ALGORITHMS

Lecturer: Doctor Nguyen Thanh Phuong
Teaching Assisstant: Master Bui Huy Thong
Project Instructor: Master Bui Huy Thong
Class: 20CTT1TN
Student: 20120131 – Nguyen Van Loc

HO CHI MINH CITY, NOVEMBER 2021

1 Information page

Name: Nguyen Van Loc
Student ID: 20120131
Class: 20CTT1TN
Subject: Data structures and algorithms
Lecturer: Doctor Nguyen Thanh Phuong
Teaching Assisstant: Master Bui Huy Thong
Project Instructor: Master Bui Huy Thong
Topic: Sorting algorithms

2 Introduction page

I have completed 11/11 required algorithms, including: selection sort, insertion sort, bubble sort, shaker sort, shell sort, heap sort, merge sort, quick sort, counting sort, radix sort, and flash sort

For output specifications, I have completed 5/5 commands, 3 for algorithm mode and 2 for comparison mode.

Below is the hardware specifications of the computer I used to run these algorithms:

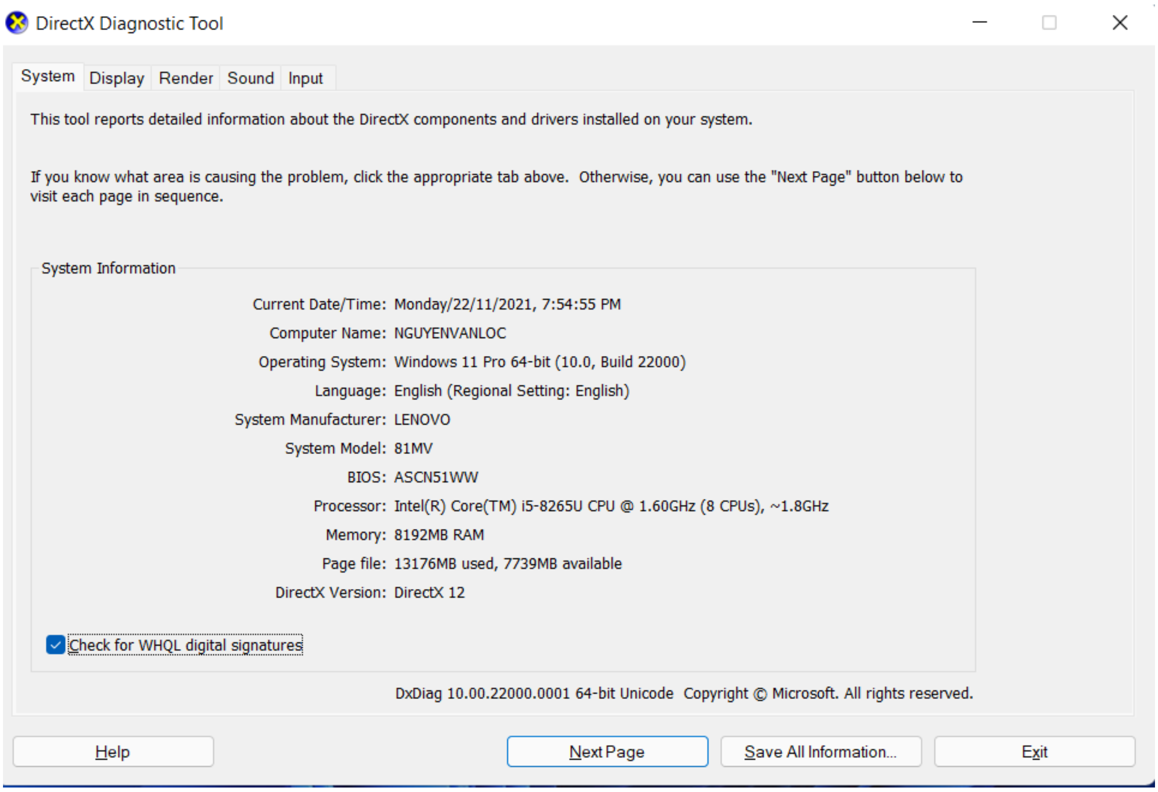


Figure 1 – Hardware specifications

Contents

List of Figures

List of Tables

List of Codes

3 Algorithm presentation

In this section, I will present the algorithms implemented in the project: ideas, step-by-step descriptions, and complexity evaluations. Varians/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 is one of the simplest sorting algorithms.
Basic ideas of this algorithm is as followed:

- In the first turn, choose the minimum element in a[1..n], then swap it with a[1], that means a[1] becomes the minimum element of the array.
- In the second turn, choose the minimum element in a[2..n], then swap it with a[2], so that a[2] becomes the second lowest element of the array.
- ...
- In the i-th turn, choose the minimum in a[i..n], then swap it with a[i].
- In the (n-1)-th turn, choose the lower between a[n - 1] and a[n], then swap it with a[n-1].

Pseudocodes: [?]

Code 1 – Selection sort

```
begin
   for i := 1 to n - 1 do
   begin
      jmin := i;
      for j := i + 1 to n do
         if (a[j] < a[jmin]) then jmin := j;
      if (jmin != i) then swap(a[jmin], a[i]);
   end
end
```

Time complexity: [?]

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

Space complexity: $O\left(1\right)$. [?]

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] ¡ 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 appopriate position.
Pseudocodes: [?]

Code 2 – Insertion sort

```
begin
   for i := 2 to n do
   begin
      temp := a[i];
      j := i - 1;
      while (j > 0) and (temp < a[j]) do
      begin
         a[j + 1] = a[j];
         dec(j);
      end
      a[j + 1] = temp;
   end
end
```

Time complexity: [?]

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

Space complexity: $O(1)$. [?]
Improvements:

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

3.3 Bubble sort

Ideas: Bubble sort is the simples 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: [?]

Code 3 – Bubble sort

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

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

Ideas: Shaker sort, also called cocktail sort or bi-directional bubble sort, is an improvement of bubble sort. In bubble sort, elements are traversed from left to right, i.e. in one direction only. But shaker sort will traverse in both direction, from left to right and from right to left, alternatively. [?]
Pseudocode: [?]

Code 4 – Shaker sort

```
begin
  left := 2;
  right := n;
  k := n;
  repeat
  begin
    for j := right downto left do
      if (a[j - 1] > a[j]) then
      begin
        swap(a[j - 1], a[j]);
        k = j;
      end
    left = k + 1; //the last swap position
    for j := left to right do
      if (a[j - 1] > a[j]) then
      begin
        swap(a[j - 1], a[j]);
        k = j;
      end
    right = k - 1;
  end

  until left > right;

end
```

Time complexity: [?]

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

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

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 $h : 1 \leqslant h \leqslant n$, we can divide the array into h subarrays:

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

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

Code 5 – Shell sort

```
begin
  gap := n div 2;
  while (gap > 0) do
  begin
    for i := gap to n do
    begin
      j := i - gap;
      k := a[i];
      while (j > 0 and a[j] > k) do
      begin
        a[j + gap] := a[j];
        j = j - gap;
      end
      a[j + gap] := k;
    end
    gap := gap div 2;
  end
end
```

Time complexity: [?]

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

Space complexity: $O\left(1\right)$. [?]

3.6 Heap sort

Heap sort was invented by J. W. J. Williams in 1981, this algorithm not only introduced an effective sorting algorithm but also built an important data structures to represent priority queues: heap data structure.

3.6.1 Heap data structure

Heap is a special binary tree. A binary tree is said to follow a heap data structure if:

- it is a complete binary tree,
- all nodes in the tree satisfy that they are greater than their children, i.e. the greatest element is the root. Such a heap is called a max-heap. If instead, all nodes are smaller than their childen, it is called a min-heap. [?]

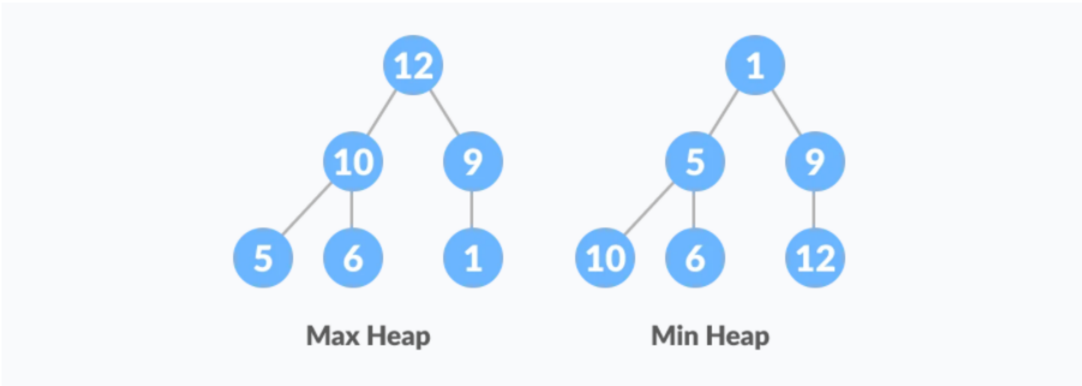


Figure 2 – Max-heap and min-heap. Source: [?]

3.6.2 Build a min-heap

To build a min heap, we: [?]

- 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: [?]

- 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

```
heapify(a[1..n], i)
begin
   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
   begin
      swap(a[i], a[max]);
      heapify(a, n, max);
   end
end

heapsort(a[1..n])
begin
   for i := n div 2 - 1 downto 1 do heapify(a, i);
   for i := n downto 1 do
   begin
      swap(a[0], a[i];
      heapify(a[1..i], 0)
   end
end
```

Time complexity: [?]

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

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

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:

- Divide the array into two subarrays at the middle position.
- Try to sort both subarrays, if we have not reached the base case yet, continue to divide them into subarrays.
- Merge the sorted subarrays.

Pseudocodes:

Code 7 – Merge sort

```
mergeSort(a[1..n])
```

```
begin
   if (n <= 1) do return;
   mid := n div 2;
   left[1..mid] := a[1..mid];
   right[1..n - mid] := a[mid + 1..n];

   mergeSort(left[1..mid]);
   mergeSort(right[1..n - mid]);

   i := 1; j := 1; k := 1;
   while (i <= mid and j <= n - mid)
   begin
      if (left[i] < right[j]) do
      begin
         a[k] := left[i];
         k := k + 1;
         i := i + 1;
      end
      else
      begin
         a[k] := right[j];
         k := k + 1;
         j := j + 1;
      end
   end
   while (i <= mid) do
   begin
      a[k] := left[i];
      k := k + 1;
      i := i + 1;
   end
   while (j <= n - mid) do
   begin
      a[k] := right[j];
      k := k + 1;
      j := j + 1;
   end
end
```

Time complexity: [?]

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

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

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.

In this project, I will choose the middle elements of the segments to be the pivot.
Pseudocodes: [?].

Code 8 – Quick sort

```
partition(a[1..n], l, r)
begin
   mid := (l + r) div 2;
   pivot := a[mid];
   i := l - 1, j := r + 1;
   repeat
      repeat
         inc(i);
      until (a[i] >= p);
      repeat
         dec(j);
      until (a[j] <= p)
      swap(a[i], a[j]);
   until (i >= j);
   swap(a[i], a[j]);
   swap(a[mid], a[j]);
   return j;
end

quicksort(a[1..n], l, r)
begin
   if (l < r) then
   begin
      s := partition(a, l, r);
      quicksort(a, l, s - 1);
      quicksort(a, s + 1, r);
   end
end
```

Time complexity: [?]

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

Space complexity: $O(1)$. [?]
Variations: Below is the implementation of quicksort using recursion. There is also an iterative algorithms, which can be found at: [?]

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). [?]
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. [?]
This algorithm works when the array contains of nonnegative integers in range $[l, u]$. The case that array is negative, the algorithms can also work but I will not mention it here.
Pseudocodes: [?]

Code 9 – Counting sort

```
countingsort(a[1..n])
begin
  f[0..u] := {0};
  for i:= 1 to n do inc(f[a[i]]);
  for i:= 1 to u do f[i] := f[i - 1] + f[i];
  //after this step, f[i] will be the number of elements that are less
  than or equal to i.
  b[1..n];
  for i := n downto 1 do
  begin
    b[f[a[i]]] = a[i];
    dec(f[a[i]]);
  end
  a := b;
end
```

Counting sort works well when $n \approx u$, but it will be "disastrous" if $u \gg n$. [?]
Time complexity: $O(n+u)$. [?]
Space complexity: $O(n+u)$. [?]

3.10 Radix sort

Like counting sort mentioned in section **??**, radix sort only works with integer.
Ideas: sort the array using counting sort (or any stable algorithms) according to the $i-$th digit. [?]
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: [?]

Code 10 – Radix sort

```
sort(a[1..n], k)
begin
  f[0..b - 1] := {0};
  for i := 1 to n do inc(f[digit(a[i], k)]);
  for i := 1 to b - 1 to f[i] := f[i] + f[i - 1];
  b[1..n]
  for i := n downto 1 do
```

```
   begin
      j := digit(a[i], k);
      b[f[j]] = a[i];
      f[j]--;
   end
   a := b;
end

LSDradixsort(a[1..n], d)
begin
   for k := 0 to d do sort(a, k);
end
```

Time complexity: $O\left(d\left(n+b\right)\right)$. [?]
Space complexity: $O\left(n\right)$. [?]

3.11 Flash sort

Flash sort is a distribution sorting algorithm, which has the time complexity approxiamtely linear complexity. [?] 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. [?] [?]

- 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:

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

Pseudocodes: [?]

Code 11 – Flash sort - stage 1

```
L[1..m] := {0};
for i := 1 to n do
begin
   k := (m - 1) * (a[i] - min) div (max - min);
   inc(L[k]);
end
for k:= 2 to n do
begin
   L[k] := L[k] + L[k - 1];
end
```

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 $i \leqslant L_{k_{a_i}}$. [?]

Psedocodes: [?]

Code 12 – Flash sort - stage 2

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

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. [?]

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

Psedocodes: [?]

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. [**?**]
Space complexity: $O\left(m\right)$.

4 Experimental results and comments

4.1 Tables of running time and comparisons count

Data order: Randomized						
Data size	10000		30000		50000	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection sort	0.125588	100019998	1.02347	90059998	2.81835	2500099998
Insertion sort	0.060877	50096183	0.53271	452071438	1.46487	1250497473
Bubble sort	0.275781	100009999	2.70246	900029999	8.1757	2500049999
Shaker sort	0.221612	66920546	2.01225	602416029	5.53941	1668740213
Shell sort	0.001633	630438	0.00589	2331067	0.013429	4629303
Heap sort	0.009767	89996	0.007112	269996	0.018274	449996
Merge sort	0.003842	337226	0.010604	1104458	0.018144	1918922
Quick sort	0.001064	268649	0.003564	918072	0.006575	1571763
Counting sort	0.000272	50004	0.00088	150004	0.001899	250004
Radix sort	0.000581	140058	0.003068	510072	0.00367	850072
Flash sort	0.000342	98814	0.001074	301965	0.002341	479127

Table 1 – Data order: Randomized - table 1

Data order: Randomized						
Data size	100000		300000		500000	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection sort	11.2068	10000199998	98.3619	90000599998	298.723	250000999998
Insertion sort	5.781	5019075369	52.6586	44979677317	145.002	124933091986
Bubble sort	31.7843	10000099999	290.83	90000299999	805.643	250000499999
Shaker sort	22.3527	6684229390	203.353	59984439772	564.835	166525795318
Shell sort	0.025796	10033440	0.085129	36188397	0.153476	67911677
Heap sort	0.027132	899996	0.116084	2699996	0.179792	4499996
Merge sort	0.018512	4037850	0.060769	13051418	0.105907	22451418
Quick sort	0.013801	3302209	0.042058	10782282	0.073066	19032583
Counting sort	0.001915	500004	0.005841	1500004	0.012324	2500004
Radix sort	0.007252	1700072	0.025703	6000086	0.0458	10000086
Flash sort	0.00513	1019156	0.017335	3047744	0.036041	4843142

Table 2 – Data order: Randomized - table 2

Data order: Sorted						
Data size	10000		30000		50000	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection sort	0.113015	100019998	1.00498	900059998	2.81835	2500099998
Insertion sort	$3.6 \cdot 10^{-5}$	29998	0.0001	89998	0.000169	149998
Bubble sort	0.114492	100009999	1.00117	900029999	2.77147	2500049999
Shaker sort	$3 \cdot 10^{-5}$	20002	$6.7 \cdot 10^{-5}$	60002	0.000445	100002
Shell sort	0.000429	360042	0.00177	1170050	0.002384	2100049
Heap sort	0.003071	89996	0.005536	269996	0.010559	449996
Merge sort	0.000969	337226	0.002866	1104458	0.009389	1918922
Quick sort	0.000245	154959	0.000815	501929	0.001504	913850
Counting sort	0.000188	50004	0.000448	150004	0.000998	250004
Radix sort	0.000579	140058	0.003068	510072	0.00367	850072
Flash sort	0.000286	127992	0.000863	383992	0.001465	639992

Table 3 – Data order: Sorted - table 1

Data order: Sorted						
Data size	100000		300000		500000	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection sort	11.2068	10000199998	98.3619	90000599998	298.723	250000999998
Insertion sort	0.000325	299998	0.00099	899998	0.001617	1499998
Bubble sort	11.0277	10000099999	102.55	90000299999	278.658	250000499999
Shaker sort	0.000561	200002	0.000648	600002	0.001081	1000002
Shell sort	0.005229	4500051	0.018094	15300061	0.030237	25500058
Heap sort	0.032474	899996	0.072256	2699996	0.121933	4499996
Merge sort	0.011628	4037850	0.034167	13051418	0.059151	22451418
Quick sort	0.003875	1927691	0.01214	6058228	0.017523	10310733
Counting sort	0.001403	500004	0.004284	1500004	0.007221	2500004
Radix sort	0.007746	1700072	0.030804	6000086	0.045303	10000086
Flash sort	0.002968	1279992	0.00882	3839992	0.014181	6399992

Table 4 – Data order: Sorted - table 2

Data order: Reversed						
Data size	10000		30000		50000	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection sort	0.10871	100019998	0.959207	900059998	2.79584	2500099998
Insertion sort	0.119653	100009999	1.04109	900029999	2.95031	2500049999
Bubble sort	0.261722	100009999	2.17328	900029999	6.11784	2500049999
Shaker sort	0.252528	100005001	2.24013	900015001	6.17515	2500025001
Shell sort	0.000561	475175	0.003068	1554051	0.003583	2844628
Heap sort	0.003353	89996	0.005536	269996	0.010559	449996
Merge sort	0.001244	337226	0.003341	1104458	0.00547	1918922
Quick sort	0.00028	164975	0.000948	531939	0.001639	963861
Counting sort	0.000137	50004	0.000929	150004	0.000998	250004
Radix sort	0.000598	140058	0.002213	510072	0.003443	850072
Flash sort	0.000267	110501	0.000806	331501	0.001302	552501

Table 5 – Data order: Reversed - table 1

Data order: Reversed						
Data size	100000		300000		500000	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection sort	10.7125	10000199998	95.9219	90000599998	267.031	250000999998
Insertion sort	11.5629	10000099999	103.857	90000299999	291/045	250000499999
Bubble sort	24.4371	10000099999	221.602	90000299999	616.86	250000499999
Shaker sort	24.8658	10000050001	226.276	90000150001	628.922	250000250001
Shell sort	0.007358	6089190	0.023358	20001852	0.04012	33857581
Heap sort	0.021055	899996	0.068389	2699996	0.123764	4499996
Merge sort	0.011581	4037850	0.033862	13051418	0.058475	22451418
Quick sort	0.00334	2027703	0.01059	6358249	0.017891	10810747
Counting sort	0.001819	500004	0.0049	1500004	0.007136	2500004
Radix sort	0.007137	1700072	0.025565	6000086	0.042483	10000086
Flash sort	0.002797	1105001	0.009906	3315001	0.013324	5525001

Table 6 – Data order: Reversed - table 2

Data order: Nearly sorted						
Data size	10000		30000		50000	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection sort	0.114235	100019998	1.01536	900059998	2.79563	2500099998
Insertion sort	0.00027	219622	0.000755	595570	0.001001	858990
Bubble sort	0.113966	100009999	0.98869	900029999	2.76097	2500049999
Shaker sort	0.000583	219726	0.001456	603170	0.001938	873634
Shell sort	0.000578	416560	0.00194	1321108	0.005379	2380925
Heap sort	0.002457	89996	0.00639	269996	0.011723	449996
Merge sort	0.001216	337226	0.004687	1104458	0.006034	1918922
Quick sort	0.000246	154999	0.000826	501973	0.001523	913898
Counting sort	0.000152	50004	0.00046	150004	0.000718	250004
Radix sort	0.000592	140058	0.002119	510072	0.003794	850072
Flash sort	0.000289	127968	0.000882	383962	0.001456	639962

Table 7 – Data order: Nearly sorted - table 1

Data order: Nearly sorted						
Data size	100000		300000		500000	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection sort	11.5729	10000199998	97.5573	90000599998	278.073	250000999998
Insertion sort	0.002272	1975946	0.005013	4436926	0.00916	8122834
Bubble sort	11.0646	10000099999	99.9988	90000299999	279.598	250000499999
Shaker sort	0.011098	1970942	0.009868	4527181	0.01822	8481121
Shell sort	0.008393	5163287	0.030022	16636275	0.035062	27752013
Heap sort	0.022278	899996	0.097582	2699996	0.121831	4499996
Merge sort	0.011873	4037850	0.033909	13051418	0.059821	22451418
Quick sort	0.003395	1927735	0.010117	6058260	0.017855	10310769
Counting sort	0.001416	500004	0.004279	1500004	0.008047	2500004
Radix sort	0.007354	1700072	0.025246	6000086	0.045863	10000086
Flash sort	0.003038	1279962	0.008703	3839958	0.014624	6399962

Table 8 – Data order: Nearly sorted - table 2

4.2 Line graphs of running time

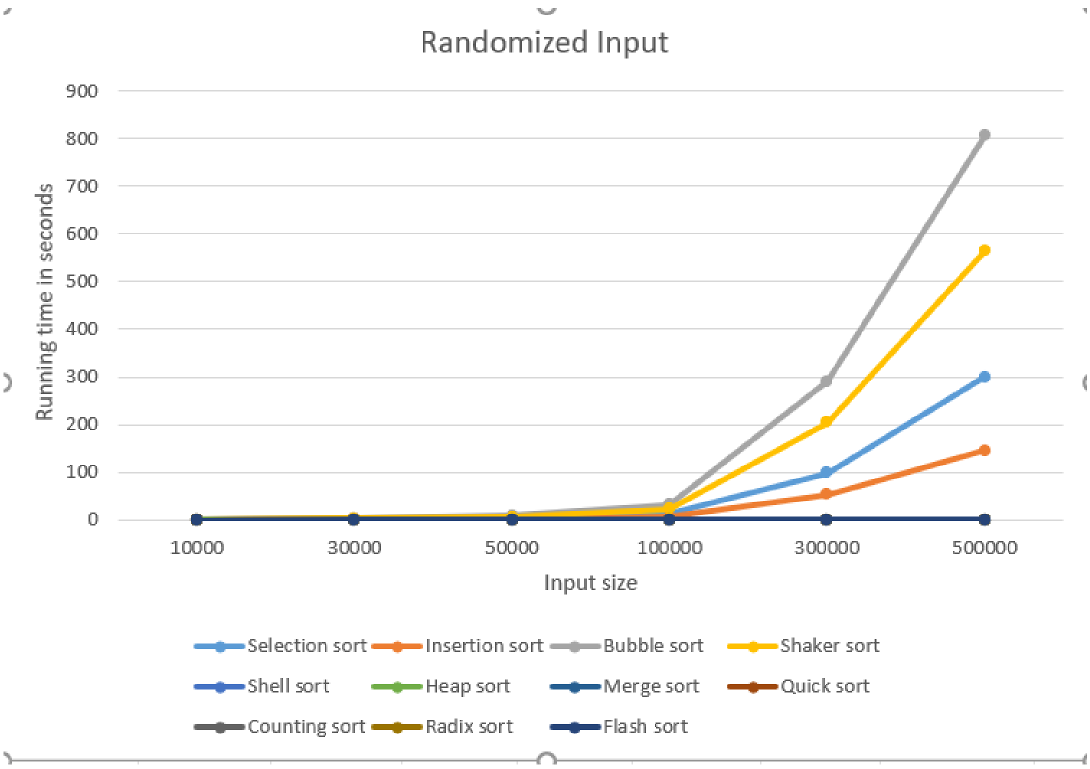


Figure 3 – Line graph of running time for randomized input

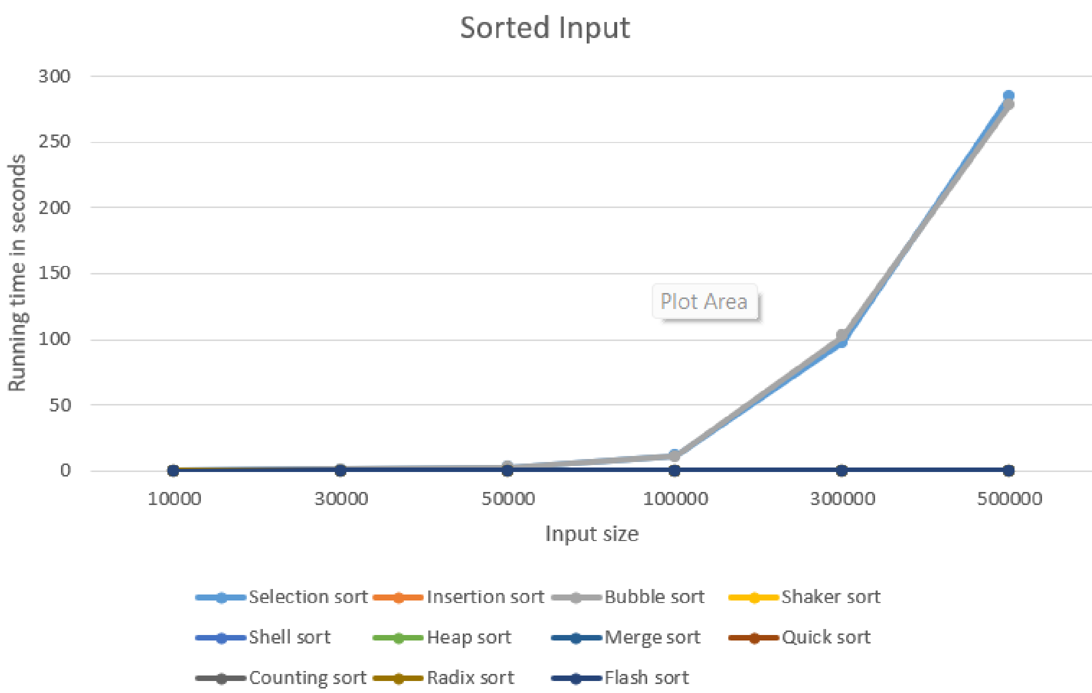


Figure 4 – Line graph of running time for sorted input

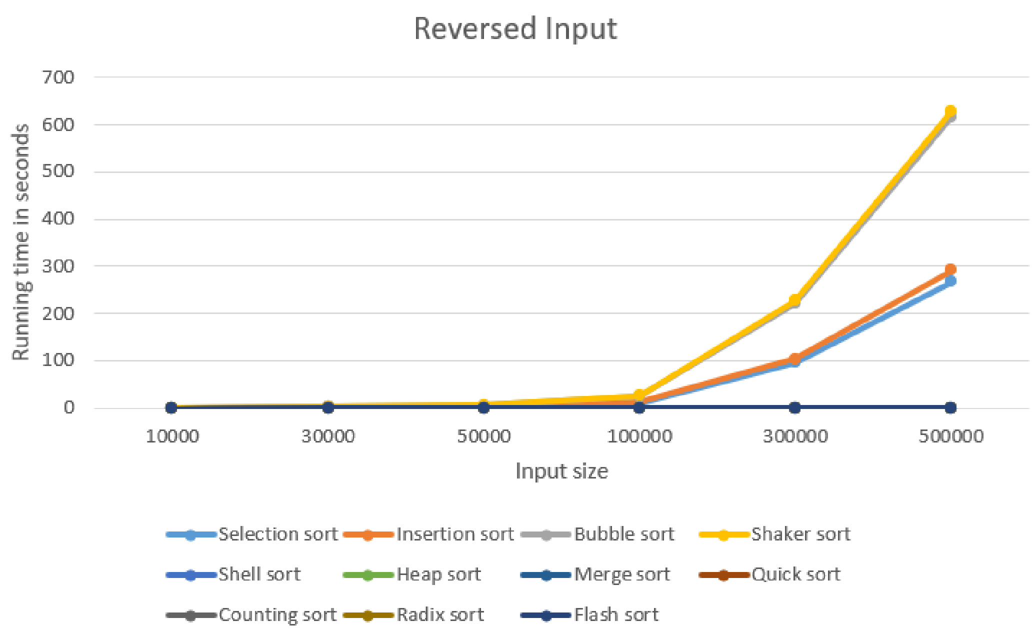


Figure 5 – Line graph of running time for reversed input

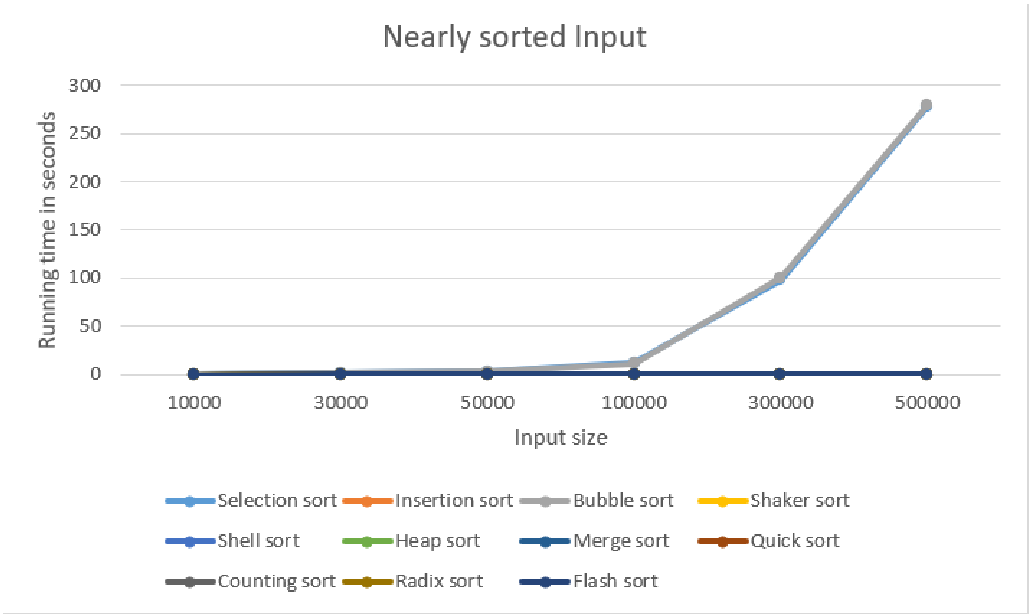


Figure 6 – Line graph of running time for nearly sorted input

4.3 Bar charts of comparisons

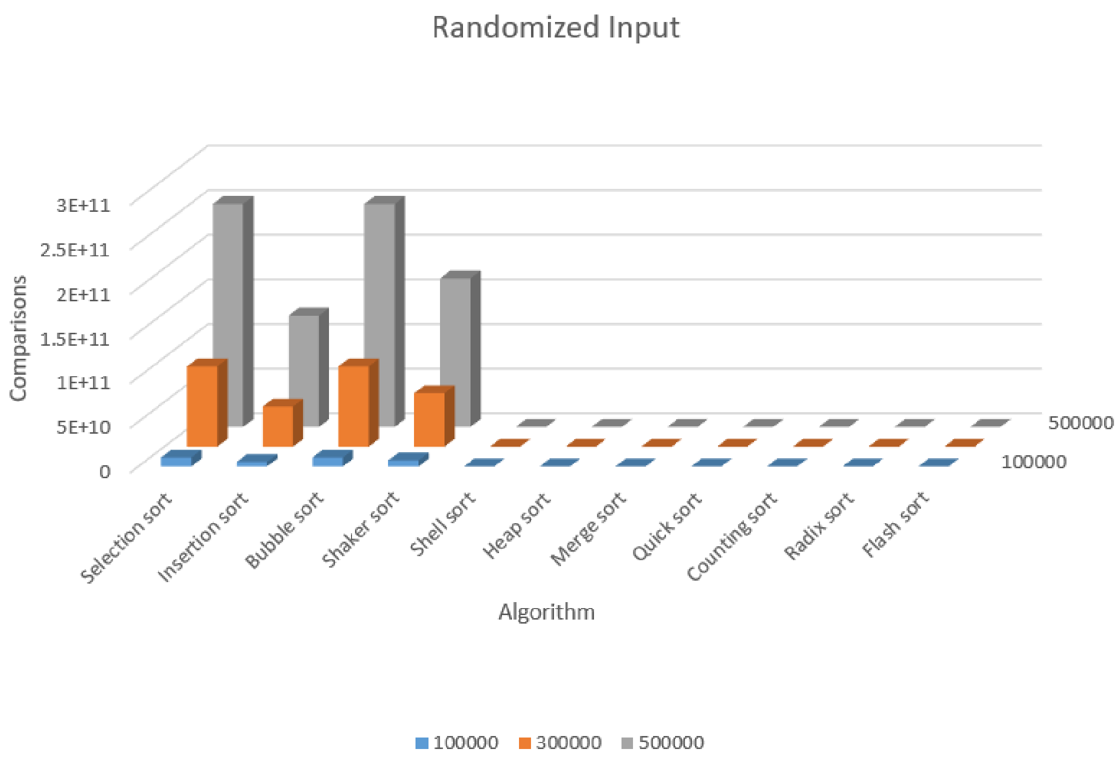


Figure 7 – Bar chart of comparisons for randomized input

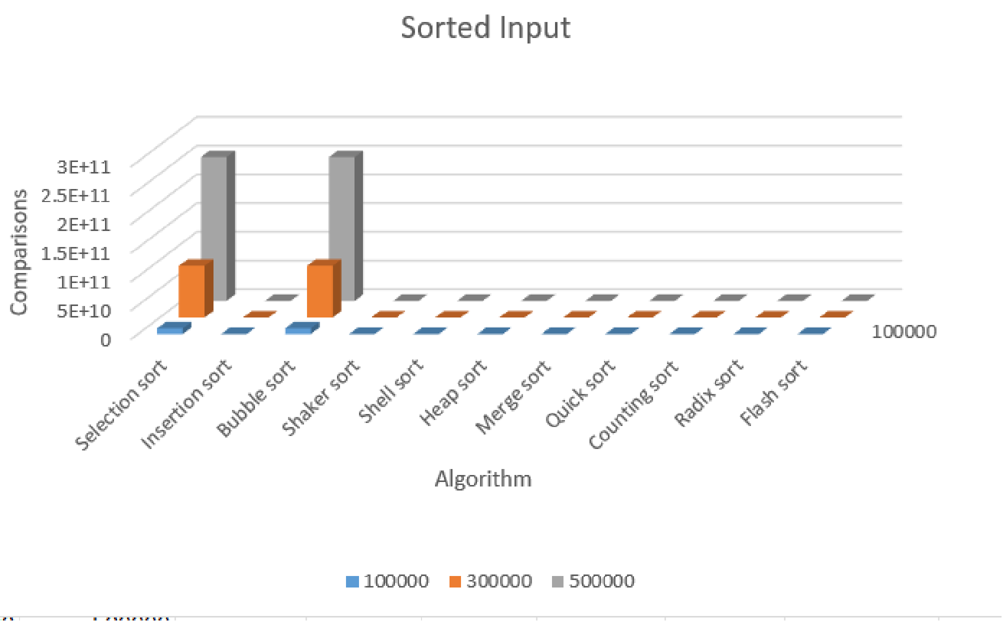


Figure 8 – Bar chart of comparisons for sorted input

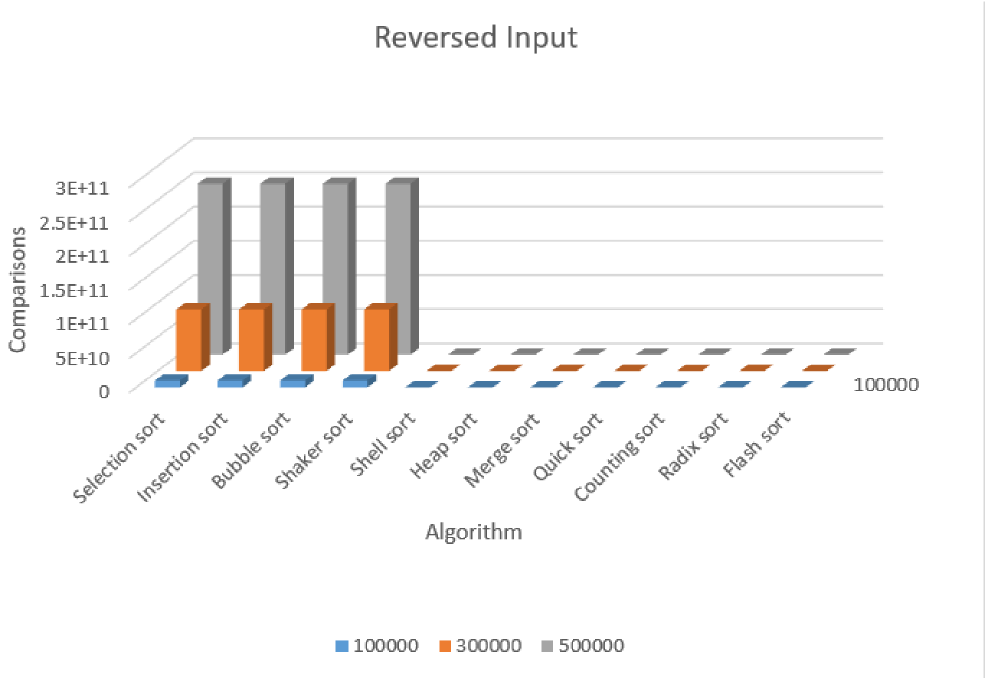


Figure 9 – Bar chart of comparisons for reversed input

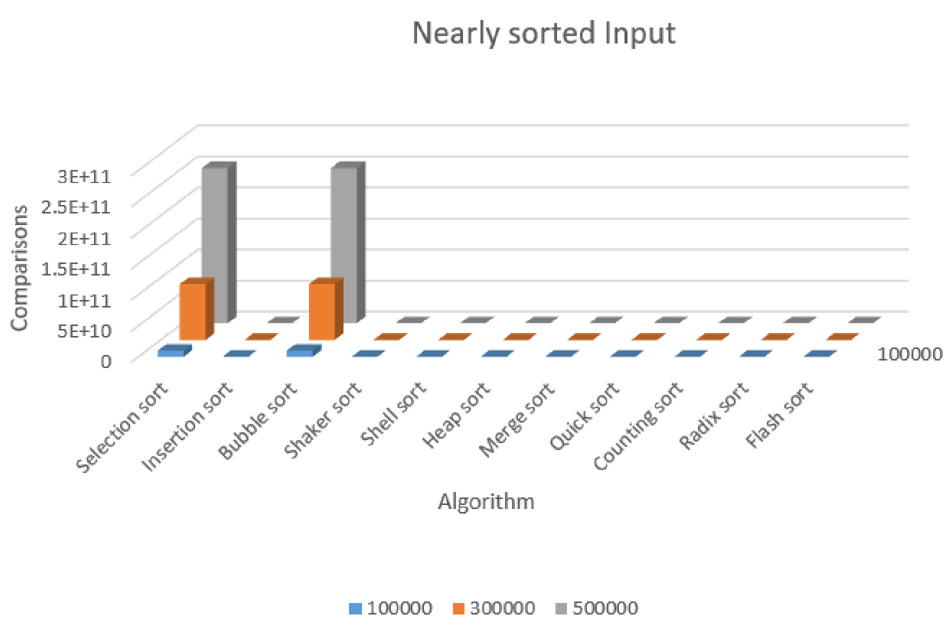


Figure 10 – Bar chart of comparisons for nearly sorted input

4.4 Comments

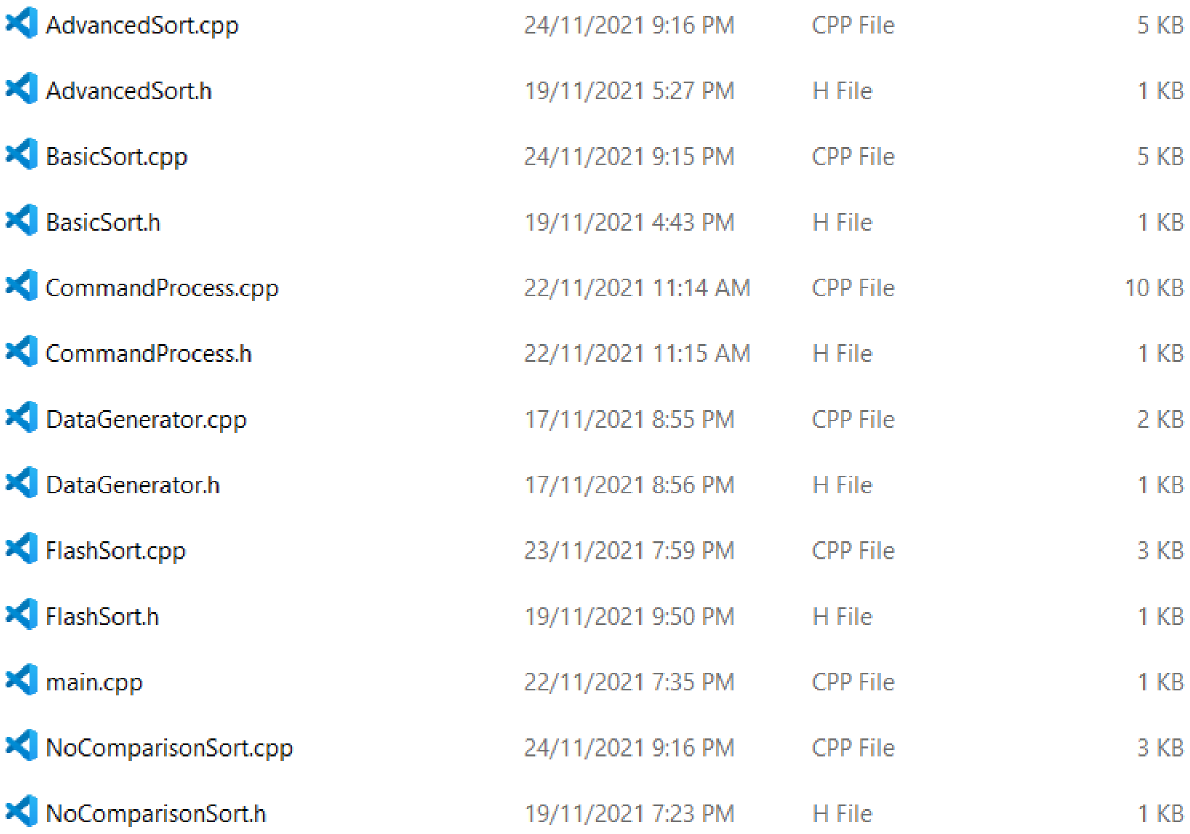
Due to my own observations on the graphs and charts above, I can claim that flash sort is the fastest and the most effective algorithm, because I chose $m \approx 0.43n$, to have the best performance. Moreover, bubble sort is the slowest and the least effective in almost all cases of input because of 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 **??**, it will be very uneffective when $u \gg n$.

5 Project organization and Programming notes

5.1 Project organization

Figure below shows files in my project.



AdvancedSort.cpp	24/11/2021 9:16 PM	CPP File	5 KB
AdvancedSort.h	19/11/2021 5:27 PM	H File	1 KB
BasicSort.cpp	24/11/2021 9:15 PM	CPP File	5 KB
BasicSort.h	19/11/2021 4:43 PM	H File	1 KB
CommandProcess.cpp	22/11/2021 11:14 AM	CPP File	10 KB
CommandProcess.h	22/11/2021 11:15 AM	H File	1 KB
DataGenerator.cpp	17/11/2021 8:55 PM	CPP File	2 KB
DataGenerator.h	17/11/2021 8:56 PM	H File	1 KB
FlashSort.cpp	23/11/2021 7:59 PM	CPP File	3 KB
FlashSort.h	19/11/2021 9:50 PM	H File	1 KB
main.cpp	22/11/2021 7:35 PM	CPP File	1 KB
NoComparisonSort.cpp	24/11/2021 9:16 PM	CPP File	3 KB
NoComparisonSort.h	19/11/2021 7:23 PM	H File	1 KB

Figure 11 – Files in project

- In BasicSort files, I declared and implemented Selection sort, Insertion sort, Bubble sort, Shaker Sort, and Shell sort.
- In AdvancedSort files, I declared and implemented Heap sort, Merge sort, and Quicksort.
- In NoComparisonSort files, I declared and implemented Counting sort and Radix sort.
- In FlashSort files, I spent to declare and implement Flash sort only.

5.2 Programming notes

My project does not use any special libraries or data structures. All are included in basic C++ 17.

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://iq.opengenus.org/time-complexity-of-selection-sort/

[4] https://www.geeksforgeeks.org/analysis-of-different-sorting-techniques

[5] https://www.geeksforgeeks.org/binary-insertion-sort/

[6] https://www.geeksforgeeks.org/bubble-sort/

[7] https://www.javatpoint.com/cocktail-sort

[8] https://www.geeksforgeeks.org/cocktail-sort/

[9] https://www.tutorialspoint.com/Shell-Sort

[10] https://www.programiz.com/dsa/heap-sort

[11] https://www.educative.io/blog/data-structure-heaps-guide

[12] https://www.programiz.com/dsa/merge-sort

[13] https://www.geeksforgeeks.org/quick-sort/

[14] https://www.geeksforgeeks.org/iterative-quick-sort/

[15] https://www.geeksforgeeks.org/counting-sort/

[16] https://www.interviewcake.com/concept/java/counting-sort

[17] https://www.geeksforgeeks.org/radix-sort/

[18] https://www.w3resource.com/javascript-exercises/searching-and-sorting-algorithm/searching-and-sorting-algorithm-exercise-12.php

[19] https://www.neubert.net/FSOIntro.html