

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 Assistant: Master Bui Huy Thong
Project Instructor: Master Bui Huy Thong
Class: 22CNTN
Student: 22120148 – Le Quang Khai

HO CHI MINH CITY, DECEMBER 2023

1 Introduction page

1.1 Personal information:

Name: Le Quang Khai

Student ID: 22120148

Class: 22CNTN

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

1.2 System information

The hardware specifications of the computer I used to run 11 sorting algorithms:

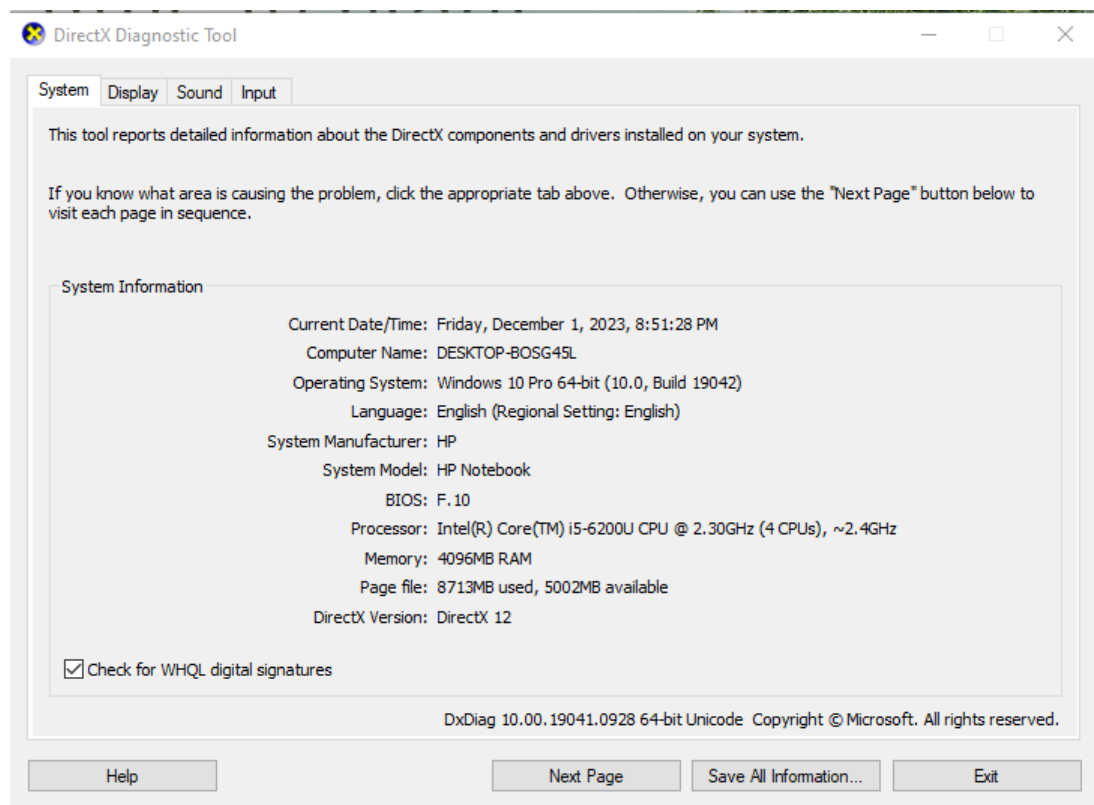


Figure 1 – Hardware specifications

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.

Contents

1	Introduction page	1
1.1	Personal information:	1
1.2	System information	1
2	Algorithm presentation	6
2.1	Selection sort	6
2.1.1	Ideas:	6
2.1.2	Step-by-step descriptions:	6
2.1.3	Pseudocodes	7
2.1.4	Time complexity:	7
2.1.5	Space complexity:	7
2.2	Insertion sort	7
2.2.1	Ideas:	7
2.2.2	Step-by-step descriptions:	7
2.2.3	Pseudocodes	8
2.2.4	Complexity evaluation	8
2.2.5	Improvements:	8
2.3	Bubble sort	9
2.3.1	Ideas	9
2.3.2	Step-by-step descriptions	9
2.3.3	Pseudocodes	10
2.3.4	Complexity evaluation	10
2.3.5	Variations	10
2.3.6	Improvements	10
2.4	Shaker sort (Cocktail sort)	10
2.4.1	Ideas	10
2.4.2	Step-by-step descriptions	10
2.4.3	Pseudocode	12
2.4.4	complexity evaluation	12
2.5	Shell sort	12
2.5.1	Ideas	12
2.5.2	Step-by-step descriptions:	13
2.5.3	Pseudocodes	14
2.5.4	Complexity evaluation	14
2.6	Heap sort	15
2.6.1	Heap data structure	15
2.6.2	Build a Max-heap	16
2.6.3	Build a min-heap	17
2.6.4	Heapsort Algorithm	17
2.6.5	Complexity evaluation	19
2.7	Merge sort	19
2.7.1	Ideas	19
2.7.2	Step-by-step descriptions	20
2.7.3	Pseudocodes	21
2.7.4	Complexity evaluation	22

2.8	Quick sort	22
2.8.1	Ideas	22
2.8.2	Step-by-step descriptions	23
2.8.3	Pseudocodes	23
2.8.4	Complexity evaluation	24
2.8.5	Variations	24
2.9	Counting sort	24
2.9.1	Ideas	24
2.9.2	Step-by-step descriptions	25
2.9.3	Pseudocodes	27
2.9.4	Complexity evaluation	28
2.10	Radix sort	28
2.10.1	Ideas	28
2.10.2	Step-by-step descriptions	28
2.10.3	Pseudocodes	29
2.10.4	Complexity evaluation	30
2.11	Flash sort	31
2.11.1	Ideas	31
2.11.2	Step-by-step descriptions and pseudocodes	31
2.11.3	Complexity	34
3	Experimental results and comments	35
3.1	Tables of running time and comparisons count	35
3.2	Line graphs of running time	39
3.3	Bar charts of comparisons	41
3.4	Comments	44
4	Project organization and Programming notes	46
4.1	Project organization	46
4.2	Programming notes	47

List of Figures

1	Hardware specifications	1
2	Line graph of running time for randomized input	39
3	Line graph of running time for sorted input	39
4	Line graph of running time for reversed input	40
5	Line graph of running time for nearly sorted input	40
6	Bar chart of comparisons for randomized input	41
7	Bar chart of comparisons for sorted input	42
8	Bar chart of comparisons for reversed input	43
9	Bar chart of comparisons for nearly sorted input	44
10	Files in project	46

List of Tables

1	Data order: Randomized - table 1	35
2	Data order: Randomized - table 2	35
3	Data order: Sorted - table 1	36
4	Data order: Sorted - table 2	36
5	Data order: Reversed - table 1	37
6	Data order: Reversed - table 2	37
7	Data order: Nearly sorted - table 1	38
8	Data order: Nearly sorted - table 2	38

2 Algorithm presentation

In this section, I will present the algorithms implemented in the project: ideas, step-by-step descriptions, pseudocodes 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.

2.1 Selection sort

2.1.1 Ideas:

The Selection Sort algorithm has a quite simple idea. The algorithm will divide the input array into 2 parts - one sorted part (usually placed on the left) and one unsorted part (usually placed on the right). The algorithm will go through $n - 1$ stages. In each stage, the algorithm will find the smallest element **in the unsorted part** and move it to **the sorted part**, then increase the length of the sorted part by 1 (accordingly, the unsorted part will be decreased by 1).

2.1.2 Step-by-step descriptions:

For example, we have the following input array:

$$a = \{4, 1, 0, 3, 2\}$$

Here I use the "|" character to separate the sorted and unsorted parts of the array. The element in **red** is the smallest element of the unsorted part.

Below are the steps to execute the algorithm:

Stage	Array a	Explain
1	{ 4, 1, 0 , 3, 2}	Initially, the sorted part is empty, and the smallest element of the unsorted part is 0. We will move element 0 to the front and increase the length of the sorted part to 1.
2	{0 4, 1 , 3, 2}	Now, the smallest element of the unsorted part is 1, so we move 1 to the sorted part.
3	{0, 1 4, 3, 2 }	Now, the smallest element of the unsorted part is 2, so we move 2 to the sorted part.
4	{0, 1, 2 4, 3 }	Now, the smallest element of the unsorted part is 3, so we move 3 to the sorted part.
5	{0, 1, 2, 3 4}	The algorithm stops here; we don't need to move element 4 to the front since it is already in the correct position.

2.1.3 Pseudocodes

```

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
end

```

2.1.4 Time complexity:

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

2.1.5 Space complexity:

Space complexity is $O(1)$.

2.2 Insertion sort

2.2.1 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 appropriate position.

2.2.2 Step-by-step descriptions:

For example, we have the following input array:

$$a = \{4, 1, 0, 3, 2\}$$

Here I use the "|" character to separate the sorted and unsorted parts of the array. The element in red is the element that will be inserted into the sorted part. The position with the "*" character is the proper position for this element.

Below are the steps to execute the algorithm:

Stage	Array a	Explain
1	{*, 4 1, 0, 3, 2}	With insertion sort, the sorted part will start with 1 element, and the next element to be inserted is element 1, it is inserted before number 4.
2	{*, 1, 4 0, 3, 2}	In this stage, we will bring element 0 to the front, and the proper position is before number 1.
3	{0, 1, *, 4 3, 2}	In this stage, we will bring element 0 to the front, and the proper position is before number 1.
4	{0, 1, *, 3, 4 2}	In this stage, we will bring element 2 to the front, and the proper position is before number 3.
5	{0, 1, 2, 3, 4 }	The algorithm stops here.

2.2.3 Pseudocodes

```

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];
          j = j - 1;
        end
      a[j + 1] = temp;
    end
  end
end

```

2.2.4 Complexity evaluation

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: $O(1)$. [4]

2.2.5 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 2.5

2.3 Bubble sort

2.3.1 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 .

2.3.2 Step-by-step descriptions

For example, we have the following input array:

$$a = \{4, 1, 0, 3, 2\}$$

Here, I mark the elements in **red** which are the two elements being compared. The "|" character separates the sorted and unsorted parts.

Below are the steps to execute the algorithm:

Iteration	Array a	Explain
1	{ 4, 1, 0, 3 , 2 }	3 and 2 are in the wrong positions, we will swap these 2 elements.
1	{ 4, 1, 0 , 2 , 3}	0 and 2 are in the correct positions, we will iterate the next 2 elements.
1	{ 4, 1 , 0 , 2, 3}	1 and 0 are in the wrong positions, we will swap these 2 elements.
1	{ 4 , 0 , 1, 2, 3}	4 and 0 are in the wrong positions, we will swap these 2 elements
1	{ 0 4, 1, 2, 3}	We finish the first iteration here, 0 is the smallest element that has been moved to the front.
2	{ 0 4, 1, 2 , 3 }	2 and 3 are in the correct positions, we will iterate the next 2 elements.
2	{ 0 4, 1 , 2 , 3}	1 and 2 are in the correct positions, we will iterate the next 2 elements.
2	{ 0 4 , 1 , 2, 3}	4 and 1 are in the wrong positions, we will swap these 2 elements
2	{ 0, 1 4, 2, 3}	We finish the 2nd iteration here
3	{ 0, 1 4, 2 , 3 }	2 and 3 are in the correct positions, we will iterate the next 2 elements.
3	{ 0, 1 4 , 2 , 3}	4 and 2 are in the wrong positions, we will swap these 2 elements
3	{ 0, 1, 2 4, 3}	We finish the 3rd iteration here

Iteration	Array a	Explain
4	{ 0, 1, 2 4 , 3 }	4 and 3 are in the wrong positions, we will swap these 2 elements
4	{ 0, 1, 2, 3 4}	We finish the 4th iteration here

2.3.3 Pseudocodes

```
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
  end
```

2.3.4 Complexity evaluation

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

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

2.3.5 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.

2.3.6 Improvements

An improvement of bubble sort is shaker sort, which we will research in section 2.4.

2.4 Shaker sort (Cocktail sort)

2.4.1 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. [7]

2.4.2 Step-by-step descriptions

For example, we have the following input array:

$$a = \{4, 1, 0, 3, 2\}$$

Here, I mark the elements in **red** which are the two elements being compared. I use the "|" character to separate the sorted and unsorted parts of the array, there will be 2 "|" characters because there are 2 sorted parts (the beginning and end of the array).

Below are the steps to execute the algorithm:

Iteration	Array a	Explain
1	{ 4, 1, 0, 3, 2 }	3 and 2 are in the wrong positions, we will swap these 2 elements.
1	{ 4, 1, 0, 2, 3 }	0 and 2 are in the correct positions, we will iterate the next 2 elements.
1	{ 4, 1, 0, 2, 3 }	1 and 0 are in the wrong positions, we will swap these 2 elements.
1	{ 4, 0, 1, 2, 3 }	4 and 0 are in the wrong positions, we will swap these 2 elements
1	{ 0 4, 1, 2, 3 }	We complete iterating from right to left, now we add 0 to the sorted part on the left. And we start iterating in order from left to right.
1	{ 0 4, 1, 2, 3 }	4 and 1 are in the wrong positions, we will swap these 2 elements.
1	{ 0 , 1, 4, 2, 3 }	4 and 2 are in the wrong positions, we will swap these 2 elements.
1	{ 0 1, 2, 4, 3 }	4 and 3 are in the wrong positions, we will swap these 2 elements.
1	{ 0 1, 2, 3 4 }	We complete iterating forward from left to right, now we add 4 to the sorted part on the right. We finish iteration 1.
2	{ 0 1, 2, 3 4 }	2 and 3 are in the correct positions, we will iterate the next 2 elements.
2	{ 0 1, 2, 3 4 }	1 and 2 are in the correct positions, we will iterate the next 2 elements.
2	{ 0, 1 2, 3 4 }	We complete iterating from right to left, now we add 1 to the sorted part on the left. And we start iterating in order from left to right.
2	{ 0, 1 2, 3 4 }	2 and 3 are in the correct positions, we will iterate the next 2 elements.
2	{ 0, 1 2 3, 4 }	We complete iterating forward from left to right, now we add 3 to the sorted part on the right. We finish iteration 2.
3	{ 0, 1, 2 3, 4 }	We complete iterating backwards from right to left, we add 2 to the sorted part on the left. Now we stop the algorithm because the array has been sorted.

2.4.3 Pseudocode

```

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
end

```

2.4.4 complexity evaluation

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: $O(1)$. [8]

2.5 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.

2.5.1 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 $h = 1$.

2.5.2 Step-by-step descriptions:

For example, we have the following input array:

$$a = \{4, 1, 0, 3, 2\}$$

And the gaps sequence:

$$gaps = \{3, 2, 1\}$$

Here, I mark the elements in **red** which are the two elements being compared.

Below are the steps to execute the algorithm:

Gap	Array a	Explain
3	$\{ \text{4}, 1, 0, \text{3}, 2 \}$	Starting with $gap = 3$, we compare element 4 and 3, we see that element 3 is in the wrong position so it will be moved to the front.
3	$\{ 3, \text{1}, 0, 4, \text{2} \}$	With $gap = 3$, we compare element 1 and 2, we see that element 2 is already in the right position so we do nothing. Here we also finish iterating the array with $gap = 3$.
2	$\{ \text{3}, 1, \text{0}, 4, 2 \}$	With $gap = 2$, we compare element 3 and 0, we see that element 0 is in the wrong position so it will be moved to the front.
2	$\{ 0, \text{1}, 3, \text{4}, 2 \}$	With $gap = 2$, we compare element 1 and 4, we see that element 4 is already in the right position so we do nothing.
2	$\{ 0, 1, \text{3}, 4, \text{2} \}$	With $gap = 2$, we compare element 3 and 2, we see that element 2 is in the wrong position so it will be moved to the front.
2	$\{ \text{0}, 1, \text{2}, 4, 3 \}$	We continue comparing 0 and 2 (because our iteration follows Insertion sort), we see that element 2 is already in the right position so we do nothing. Here we also finish iterating the array with $gap = 2$.

Gap	Array a	Explain
1	{ 0, 1, 2, 4, 3 }	With $gap = 1$, we compare element 0 and 1, we see that element 1 is already in the right position so we do nothing.
1	{ 0, 1, 2, 4, 3 }	With $gap = 1$, we compare element 1 and 2, we see that element 2 is already in the right position so we do nothing.
1	{ 0, 1, 2, 4, 3 }	With $gap = 1$, we compare element 2 and 4, we see that element 4 is already in the right position so we do nothing.
1	{ 0, 1, 2, 4, 3 }	With $gap = 1$, we compare element 4 and 3, we see that element 3 is in the wrong position so it will be moved to the front.
1	{ 0, 1, 2, 3, 4 }	We continue comparing 2 and 3, we see that element 3 is already in the right position so we do nothing. Here we also finish iterating the array with $gap = 1$ and finish the algorithm.

2.5.3 Pseudocodes

```

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
        end
      gap := gap div 2;
    end
  end
end

```

2.5.4 Complexity evaluation

Time complexity:

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

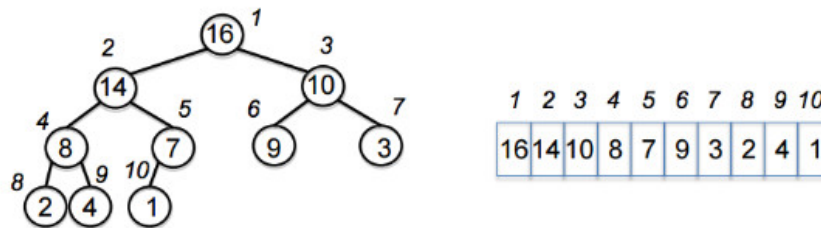
Space complexity: $O(1)$. [9]

2.6 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.

2.6.1 Heap data structure

The binary heap data structure is heap implementation. These are often shown as an array object that can be viewed as nearly complete binary tree built out of a given set of data. The heap data structure is also used in the construction of a priority queue. The complete binary tree maps the binary tree structure into array indices, as shown in the figure below. Each array index represents a node.



Example of a Max Heap. Source: [10]

The node's parent, left, and right child can be expressed as

- $\text{parent}(i) \rightarrow \lfloor \frac{i}{2} \rfloor$
- $\text{leftchild}(i) \rightarrow 2 \times i$
- $\text{rightchild}(i) \rightarrow 2 \times i + 1$

There are two kinds of binary heaps: **max-heap** and **min-heap**. Both types of heaps satisfy a certain heap property.

Max-heap Property:

If A is an array representation of a heap, then in max-heap:

$$A[\text{parent}(i)] \geq A[i]$$

which means that a node can't have a greater value than its parent. In a max-heap, the largest element is stored at the root, and the minimum elements are in the leaves.

Min-heap Property:

Similarly, if A is an array representation of a heap, then in min-heap:

$$A[\text{parent}(i)] \leq A[i]$$

which means that a parent node can't have a greater value than its children. Thus, the minimum element is located at the root, and the maximum elements are located in the leaves.

2.6.2 Build a Max-heap

To build a max heap, we: [11]

- 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 max heap, we: [11]

- 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 smaller than its children, swap them, and repeat until the heap property is satisfied.

Step-by-step descriptions:

For example, we have the following input array:

$$a = \{4, 1, 0, 3, 2\}$$

Here, when implementing a heap using an array, I consider the element at position i to have its left child at position $2i + 1$ and its right child at position $2i + 2$.

Below are the steps to execute the algorithm in the max-heap construction phase. The element in **red** is the element being considered, the element in **blue** is the child of the element being considered.

The position being considered	Array a	Explain
1	$\{ 4, \text{red } 1, 0, \text{blue } 3, \text{blue } 2 \}$	Because elements at positions in the range $[\frac{n}{2}, n-1]$ will not have children, we skip them and do not need to consider them. We will consider from the element at position $\frac{n}{2} - 1$. Element 1 has two children 3 and 2. Since we are building a max-heap, we need to move 3 up to position 1.
0	$\{ \text{red } 4, \text{blue } 3, \text{blue } 0, 1, 2 \}$	We consider the element at position 0, this element has two children with values 3 and 0 which are both smaller than 4, therefore we do nothing. Here we complete the max-heap construction phase.

Pseudocodes:

Here to build a Max-heap, we use the `heapify(a[1..n], i)` function. This function adjusts the nodes of the tree (as described above) to satisfy the max-heap property.

```
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
end

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

2.6.3 Build a min-heap

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

2.6.4 Heapsort Algorithm

The heapsort algorithm has two main parts (that will be broken down further below): building a max-heap and then sorting it. The max-heap is built as described in the above section. Then, heapsort produces a sorted array by repeatedly removing the largest element from the heap (which is the root of the heap), and then inserting it into the array. The heap is updated after each removal. Once all elements have been removed from the heap, the result is a sorted array.

Idea:

The heapsort algorithm uses the `heapify` function, and all put together, the heapsort algorithm sorts a heap array A like this:

1. Build a max-heap from an unordered array.
2. Find the maximum element, which is located at $A[0]$ because the heap is a max-heap.
3. Swap elements $A[n]$ and $A[0]$ so that the maximum element is at the end of the array where it belongs.
4. Decrement the heap size by one (this discards the node we just moved to the bottom of the heap, which was the largest element). In a manner of speaking, the sorted part of the list has grown and the heap (which holds the unsorted elements) has shrunk.

5. Now run **heapify** on the heap in case the new root causes a violation of the max-heap property. (Its children will still be max-heaps.)
6. Return to step 2.

Step-by-step descriptions:

For example, we have the following input array:

$$a = \{4, 1, 0, 3, 2\}$$

Here, when implementing a heap using an array, I consider the element at position i to have its left child at position $2i + 1$ and its right child at position $2i + 2$.

First phase, we build a Max-heap as above. Assume we have finished building it and the result of the initial array after constructing the Max-heap is

$$a = \{4, 3, 0, 1, 2\}$$

Next is phase 2. Here I will use the "|" character to separate the array into 2 parts - the max-heap part and the sorted part.

Iteration	Array a	Explain
1	{ 4, 3, 0, 1, 2 }	First we move element 4 to the end of the array, then move element 2 up to replace element 4.
1	{ 2, 3, 0, 1 4 }	We will push element 2 down to the proper position to ensure the max heap property.
1	{ 3, 2, 0, 1 4 }	Now element 2 is in the right position, we stop iteration 1. Note that now this element only has 1 child, because element 4 is out of the current managed range.
2	{ 3, 2, 0, 1 4 }	We move 3 to the back, bring element 1 up to replace 3.
2	{ 1, 2, 0 3, 4 }	We need to bring element 1 to the proper position, we will swap the positions of element 1 and 2.
2	{ 2, 1, 0 3, 4 }	Now element 1 is in the right position, we stop iteration 2.
3	{ 2, 1, 0 3, 4 }	We move 2 to the back, bring element 0 up to replace 2.
3	{ 0, 1 2, 3, 4 }	We need to bring element 0 to the proper position, we will swap the positions of element 0 and 1.
3	{ 1, 0 2, 3, 4 }	Element 0 is now in the right position, we finish iteration 3.
4	{ 1, 0 2, 3, 4 }	We move element 1 to the back, bring element 0 up to replace it.
4	{ 0 1, 2, 3, 4 }	Element 0 is now in the right position, we finish iteration 4 and finish the algorithm.

Pseudocodes:

```

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
end

heapsort(a[1..n])
begin
    // Phase 1: building Max-heap
    for i := n div 2 - 1 downto 1 do heapify(a, i);

    // Phase 2
    for i := n downto 1 do
        begin
            swap(a[0], a[i]);
            heapify(a[1..i], 0)
        end
    end
end

```

2.6.5 Complexity evaluation**Time complexity:**

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

Space complexity: $O(1)$. [11]

2.7 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.

2.7.1 Ideas

Merge sort uses divide-and-conquer:

1. **Divide** by finding the number q of the position midway between p and r . Do this step the same way we found the midpoint in binary search: add p and r , divide by 2, and round down.
2. **Conquer** by recursively sorting the subarrays in each of the two subproblems created by the divide step. That is, recursively sort the subarray `array[p..q]` and recursively sort the subarray `array[q + 1..r]`.
3. **Combine** by merging the two sorted subarrays back into the single sorted subarray `array[p..r]`.

We need a base case. The base case is a subarray containing fewer than two elements, that is, when $p \geq r$, since a subarray with no elements or just one element is already sorted. So we'll divide-conquer-combine only when $p < r$.

2.7.2 Step-by-step descriptions

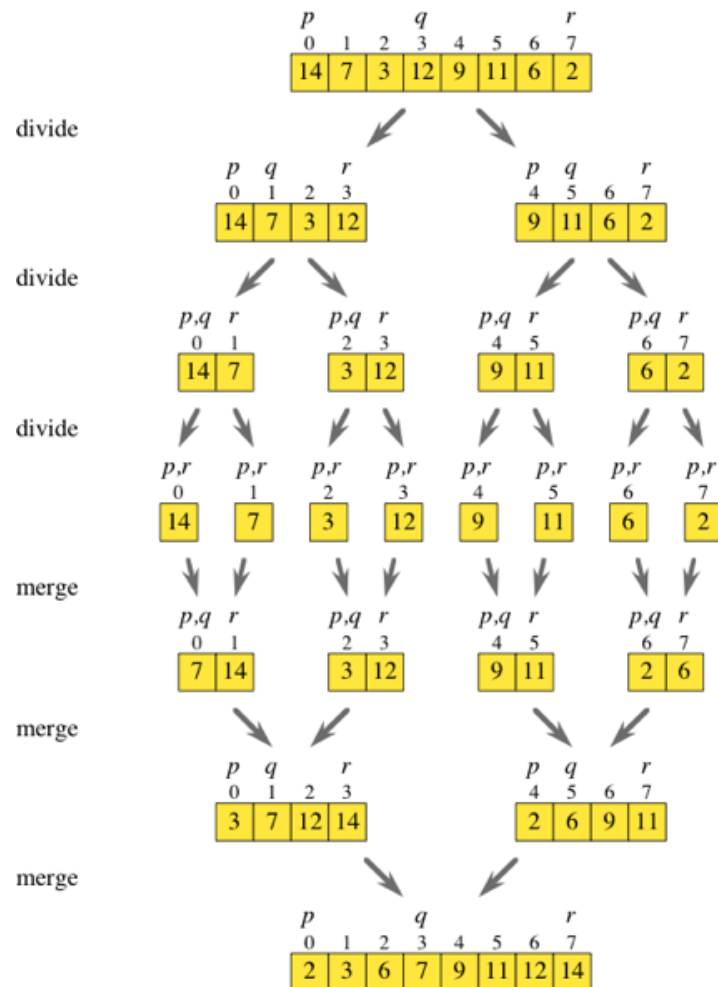
Let's see an example. Let's start with `array` holding `[14, 7, 3, 12, 9, 11, 6, 2]`, so that the first subarray is actually the full array, `array[0..7]` ($p = 0$ and $r = 7$). This subarray has at least two elements, and so it's not a base case.

- In the *divide* step, we compute $q = 3$.
- The *conquer* step has us sort the two subarrays `array[0..3]`, which contains `[14, 7, 3, 12]`, and `array[4..7]`, which contains `[9, 11, 6, 2]`. When we come back from the conquer step, each of the two subarrays is sorted: `array[0..3]` contains `[3, 7, 12, 14]` and `array[4..7]` contains `[2, 6, 9, 11]`, so that the full array is `[3, 7, 12, 14, 2, 6, 9, 11]`.
- Finally, the *combine* step merges the two sorted subarrays in the first half and the second half, producing the final sorted array `[2, 3, 6, 7, 9, 11, 12, 14]`.

How did the subarray `array[0..3]` become sorted? The same way. It has more than two elements, and so it's not a base case. With $p = 0$ and $r = 3$, compute $q = 1$, recursively sort `array[0..1]` (`[14, 7]`) and `array[2..3]` (`[3, 12]`), resulting in `array[0..3]` containing `[7, 14, 3, 12]`, and merge the first half with the second half, producing `[3, 7, 12, 14]`.

How did the subarray `array[0..1]` become sorted? With $p = 0$ and $r = 1$, compute $q = 0$, recursively sort `array[0..0]` (`[14]`) and `array[1..1]` (`[7]`), resulting in `array[0..1]` still containing `[14, 7]`, and merge the first half with the second half, producing `[7, 14]`.

The subarrays `array[0..0]` and `array[1..1]` are base cases, since each contains fewer than two elements. Here is how the entire merge sort algorithm unfolds:



Example of Merge sort. Source: [14]

Most of the steps in merge sort are simple. You can check for the base case easily. Finding the midpoint q in the divide step is also really easy. You have to make two recursive calls in the conquer step. It's the combine step, where you have to merge two sorted subarrays, where the real work happens.

2.7.3 Pseudocodes

```

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
end

```

2.7.4 Complexity evaluation

Time complexity:

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

Space complexity: $O(n)$. [13]

2.8 Quick sort

Like merge sort, quicksort uses divide-and-conquer, and so it's a recursive algorithm. The way that quicksort uses divide-and-conquer is a little different from how merge sort does. In merge sort, the divide step does hardly anything, and all the real work happens in the combine step. Quicksort is the opposite: all the real work happens in the divide step. In fact, the combine step in quicksort does absolutely nothing.

2.8.1 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.

2.8.2 Step-by-step descriptions

For example, we have the following input array:

$$a = \{4, 1, 2, 3, 0\}$$

Here I use square brackets "[]" to represent the arrays being recursively called. The elements in **red** are the pivot elements.

Array a	Explain
{ 4, 3, 2 , 1, 0 }	Choose 2 as the pivot element. Divide into two arrays [1, 0] and [4, 3].
{ [1 , 0], 2, [4 , 3] }	In [1, 0], we choose [1] as the pivot element. Divide into two arrays [0] and [1], here we have completed one part.
{ 0, 1, 2, [4 , 3] }	We choose 4 as the pivot element, divide into two arrays [3] and [4].
{ 0, 1, 2, 3, 4 }	We finish the algorithm.

2.8.3 Pseudocodes

```

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]);
    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
end

```

2.8.4 Complexity evaluation

Time complexity: [15]

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

Space complexity: $O(1)$. [15]

2.8.5 Variations

Below is the implementation of quicksort using recursion. There is also an iterative algorithms, which can be found at: [16]

2.9 Counting sort

Counting sort is a sorting algorithm working by counting the number of objects having distinct key values (a kind of hashing). [17]

2.9.1 Ideas

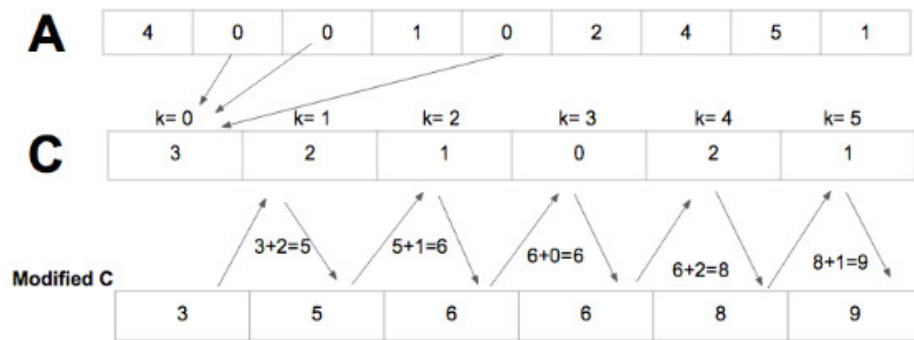
This algorithm works when the array contains of nonnegative integers in range $[l, u]$. The case that array is negative, the algorithms can also work but I will not mention it here.[18]

Counting sort assumes that each of the n input elements in a list has a key value ranging from 0 to k , for some integer k . For each element in the list, counting sort determines the number of elements that are less than it. Counting sort can use this information to place the element directly into the correct slot of the output array.

Counting sort uses three lists: the input list, $A[0, 1, \dots, n]$, the output list, $B[0, 1, \dots, n]$, and a list that serves as temporary memory, $C[0, 1, \dots, k]$. Note that A and B have n slots (a slot for each element), while C contains k slots (a slot for each key value).

Counting sort starts by going through A , and for each element $A[i]$, it goes to the index of C that has the same value as $A[i]$ (so it goes to $C[A[i]]$) and increments the value

of $C[A[i]]$ by one. This means that if A has seven 0's in its list, after counting sort has gone through all n elements of A , the value at $C[0]$ will be 7. Similarly, if A has two 4's, after counting sort has gone through all of the elements of A , $C[4]$ (using 0 indexing) will be equal to 2. In this step, C keeps track of how many elements in A there are that have the same value of a particular index in C . In other words, the indices of C correspond to the values of elements in A , and the values in C correspond to the total number of times that a value in A appears in A .



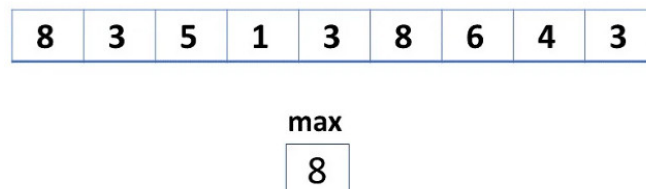
Example of Counting sort. Source: [18]

Next, modify C so that each $C[j]$ includes the number of elements less than or equal to it. This can be accomplished by going through C and replacing each $C[j]$ value with $C[j] + C[j-1]$. This step allows counting sort to determine at what index in B an element should be placed.

Then, starting at the end of A , add elements to B by checking the value of $A[i]$, going to $C[A[i]]$, writing the value of the element at $A[i]$ to $B[C[A[i]] - 1]$. Finally, decrement the value of $C[A[i]]$ by 1 since that slot in B is now occupied.

2.9.2 Step-by-step descriptions

Consider a given array that needs to be sorted. First, you'll have to find the largest element in the array and set it to the max.



Find the largest element in the array. Source: [19]

To store the sorted data, you will now initialize a new count array with length $max + 1$ and all elements set to 0.

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

All elements set to 0. Source: [19]

Later, as shown in the figure, you will store elements of the given array with the corresponding index in the count array.

0	1	2	3	4	5	6	7	8	9
0	1	0	3	1	1	1	0	2	0

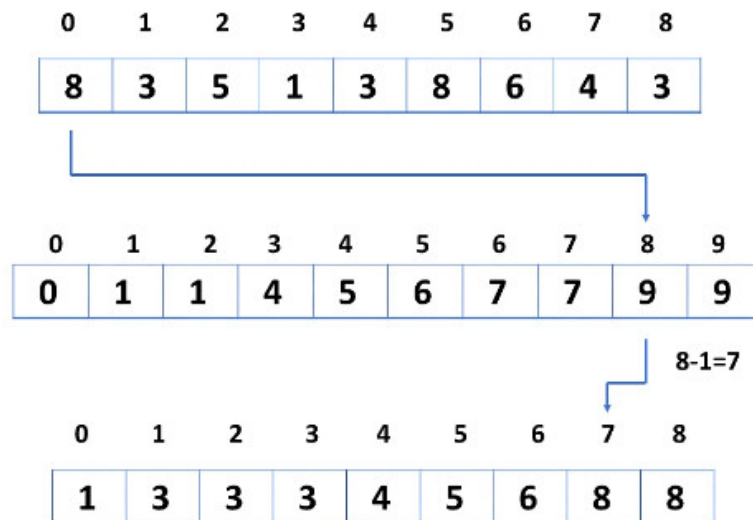
Store elements of the given array with the corresponding index in the count array. Source: [19]

Now, you will change the count array by adding the previous counts to produce the cumulative sum of an array, as shown below:

0	1	2	3	4	5	6	7	8	9
0	1	1	4	5	6	7	7	9	9

Add the previous counts to produce the cumulative sum of an array. Source: [19]

Because the original array has nine inputs, you will create another empty array with nine places to store the sorted data, place the elements in their correct positions, and reduce the count by one.



Place the elements in their correct positions, and reduce the count by one.. Source: [19]

As a result, the sorted array is:

0	1	2	3	4	5	6	7	8
1	3	3	3	4	5	6	8	8

The sorted array. Source: [19]

2.9.3 Pseudocodes

```

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 $u \gg n$. [2]

2.9.4 Complexity evaluation

Time complexity: $O(n + u)$. [17]

Space complexity $O(n + u)$. [17]

2.10 Radix sort

Like counting sort mentioned in section 2.9, radix sort only works with integer.

2.10.1 Ideas

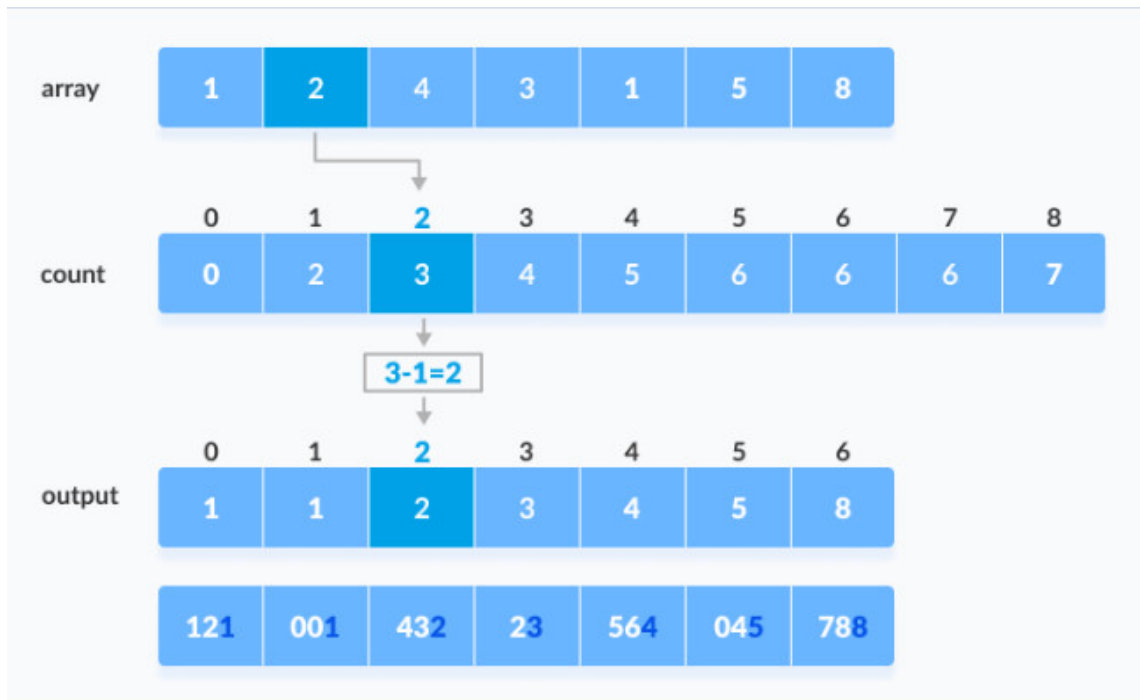
Radix sort is an integer sorting algorithm that sorts data with integer keys by grouping the keys by individual digits that share the same significant position and value (place value). Radix sort uses *counting sort* as a subroutine to sort an array of numbers. Because integers can be used to represent strings (by hashing the strings to integers), radix sort works on data types other than just integers. Because radix sort is not comparison based, it is not bounded by $\Omega(n \log n)$ for running time – in fact, radix sort can perform in linear time. [21]

Radix sort incorporates the counting sort algorithm so that it can sort larger, multi-digit numbers without having to potentially decrease the efficiency by increasing the range of keys the algorithm must sort over (since this might cause a lot of wasted time). [21]

2.10.2 Step-by-step descriptions

Step 1: Find the largest element in the array, i.e. \max . Let X be the number of digits in \max . X is calculated because we have to go through all the significant places of all elements.

In this array {121, 432, 564, 23, 1, 45, 788 }, we have the largest number 788. It has 3 digits. Therefore, the loop should go up to hundreds place (3 times). [22] **Step 2:** Now, go through each significant place one by one. Use any stable sorting technique to sort the digits at each significant place. We have used counting sort for this. Sort the elements based on the unit place digits ($X = 0$).



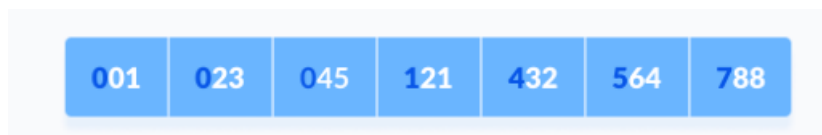
Using counting sort to sort elements based on unit place . Source: [22]

Step 3: Now, sort the elements based on digits at tens place.



Sort elements based on tens place. Source: [22]

Step 4: Finally, sort the elements based on the digits at hundreds place.



Sort elements based on hundreds place. Source: [22]

2.10.3 Pseudocodes

```
digit(x, k: integer): integer
begin
    digit := x div exp(10, k-1) mod 10;
end
```

```

sort(a[1..n], k)
begin
    // Use any stable sorting technique to sort the digits at each
    //significant place
    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];

    // Sort the elements based on digits at kth place.
    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
    // Find the largest element in the array
    max := a[1]
    for i := 2 to n do
    begin
        if a[i] > max then max := a[i];
    end

    // Find the number of digits in max
    exp := 1
    d := 0
    while max div exp != 0 do
    begin
        inc(d);
        exp = exp*10;
    end

    for k := 0 to d do sort(a, k);
end

```

2.10.4 Complexity evaluation

Time complexity: $O(d(n + b))$. [22]

Space complexity: $O(n)$. [22]

2.11 Flash sort

Flash sort is a distribution sorting algorithm, which has the time complexity approximately linear complexity. [23] 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.

2.11.1 Ideas

The algorithm is divided into three stages. [2] [24]

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

2.11.2 Step-by-step descriptions and pseudocodes

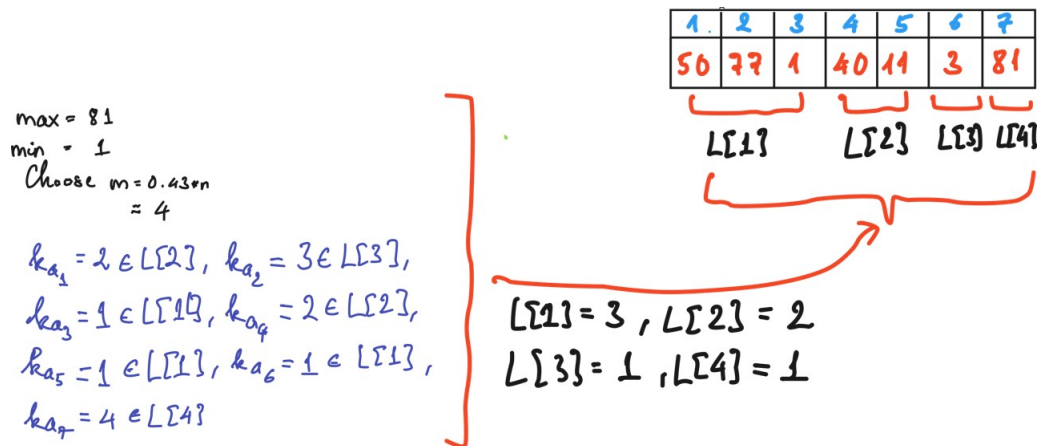
Stage 1: Classification of elements of the array

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

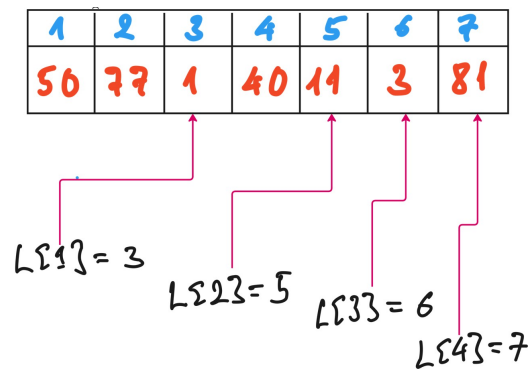
$$k_{a_i} = \left\lfloor \frac{(m-1)(a_i - \min_a)}{\max_a - \min_a} \right\rfloor + 1.$$

Descriptions:

Calculate the number of elements in each class:



Adjusting the classes $L[i]$ means representing the number of elements in a class as pointers to the far right end of each class.



Pseudocodes: [2]

```

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

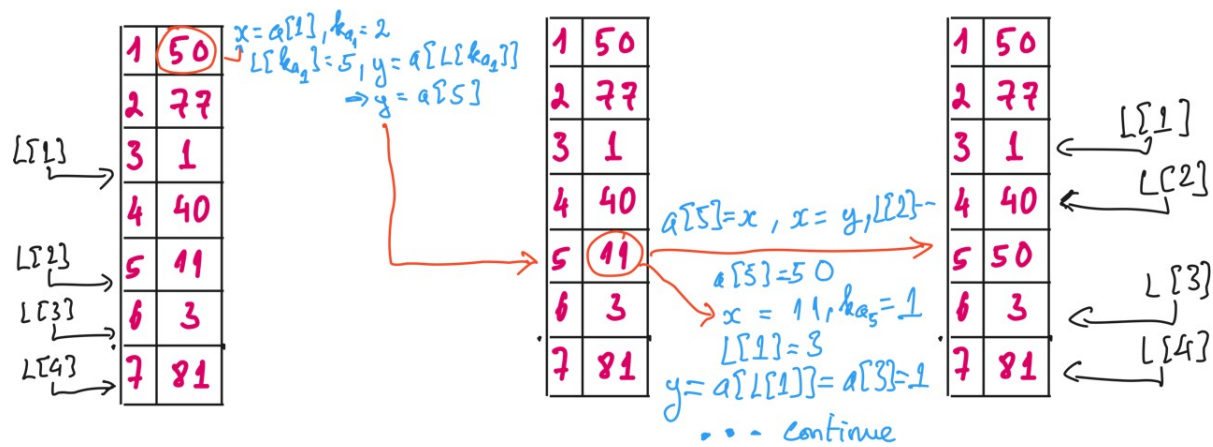
```

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

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 $i \leq L_{k_{a_i}}$. [24]

Decriptions:

Move the elements into their correct class:

**Pseudocodes:** [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

```

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. [24]

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

Pseudocodes: [2]

```

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.

2.11.3 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: $O(m)$.

3 Experimental results and comments

3.1 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	0.160063	100019998	1.368900	900059998	3.893490	2500099998
Insertion sort	0.023106	50258848	0.163568	450753951	0.339820	1254687888
Bubble sort	0.268950	100009999	2.686420	900029999	8.156500	2500049999
Shaker sort	0.257278	67046847	2.525520	601287285	6.563840	1674719206
Shell sort	0.001503	407729	0.004626	1341751	0.006408	2567546
Heap sort	0.000802	637775	0.004132	2150592	0.004924	3771140
Merge sort	0.006089	337226	0.028536	1104458	0.037824	1918922
Quick sort	0.001383	279033	0.003320	945002	0.005251	1585467
Counting sort	0.000258	60005	0.000447	180005	0.000770	265541
Radix sort	0.000578	140058	0.001934	510072	0.004033	850072
Flash sort	0.000339	98554	0.001319	287994	0.002118	481896

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	14.942500	10000199998	134.698000	90000599998	386.115000	250000999998
Insertion sort	1.534230	5017032819	15.113400	45033334887	38.823600	125150641269
Bubble sort	30.853800	10000099999	277.519000	90000299999	773.650000	250000499999
Shaker sort	26.112500	6687790327	234.617000	60034741593	657.148000	166894524549
Shell sort	0.016343	5880567	0.057619	20020053	0.092058	36191258
Heap sort	0.013370	8044792	0.052706	26490297	0.138432	45969706
Merge sort	0.054960	4037850	0.260414	13051418	0.383066	22451418
Quick sort	0.011212	3386422	0.048287	10434431	0.132994	18127223
Counting sort	0.001707	465541	0.006603	1265541	0.018610	2065541
Radix sort	0.007562	1700072	0.022573	5100072	0.033952	8500072
Flash sort	0.005839	894994	0.025978	2851150	0.078891	4504447

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	0.152932	100019998	1.351850	900059998	4.287380	2500099998
Insertion sort	0.000021	29998	0.000034	89998	0.000056	149998
Bubble sort	0.073611	100009999	0.631910	900029999	1.753310	2500049999
Shaker sort	0.000016	20002	0.000047	60002	0.000079	100002
Shell sort	0.000261	240037	0.000618	780043	0.001122	1400043
Heap sort	0.000724	670333	0.007213	2236652	0.004581	3925355
Merge sort	0.006654	337226	0.026689	1104458	0.219562	1918922
Quick sort	0.000174	154959	0.001272	501929	0.000972	913850
Counting sort	0.000113	60005	0.000154	180005	0.000384	300005
Radix sort	0.000538	140058	0.001426	510072	0.003705	850072
Flash sort	0.000181	127992	0.000602	383992	0.000993	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	14.932500	10000199998	133.277000	90000599998	393.520000	250000999998
Insertion sort	0.000112	299998	0.000336	899998	0.001012	1499998
Bubble sort	6.799890	10000099999	60.819900	90000299999	168.808000	250000499999
Shaker sort	0.000120	200002	0.000457	600002	0.000764	1000002
Shell sort	0.003292	3000045	0.010048	10200053	0.014889	17000051
Heap sort	0.008747	8365084	0.035410	27413234	0.090547	47404890
Merge sort	0.043892	4037850	0.150830	13051418	0.136720	22451418
Quick sort	0.001899	1927691	0.007209	6058228	0.010912	10310733
Counting sort	0.000681	600005	0.003148	1800005	0.009361	3000005
Radix sort	0.007034	1700072	0.026852	6000086	0.044579	10000086
Flash sort	0.002337	1279992	0.005885	3839992	0.010751	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	0.153952	100019998	1.354180	900059998	3.764810	2500099998
Insertion sort	0.035954	100009999	0.346777	900029999	0.765436	2500049999
Bubble sort	0.343138	100009999	3.063080	900029999	8.732210	2500049999
Shaker sort	0.391938	100005001	3.140640	900015001	8.714390	2500025001
Shell sort	0.000356	302597	0.000846	987035	0.001528	1797323
Heap sort	0.000668	606775	0.034989	2063328	0.004350	3612728
Merge sort	0.005621	337226	0.025378	1104458	0.042812	1918922
Quick sort	0.000221	164975	0.000481	531939	0.000945	963861
Counting sort	0.000146	60005	0.000161	180005	0.000199	300005
Radix sort	0.000613	140058	0.002859	510072	0.004317	850072
Flash sort	0.000209	110501	0.001247	331501	0.001355	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	14.942500	10000199998	134.698000	90000599998	386.115000	250000999998
Insertion sort	1.534230	5017032819	15.113400	45033334887	38.823600	125150641269
Bubble sort	30.853800	10000099999	277.519000	90000299999	773.650000	250000499999
Shaker sort	26.112500	6687790327	234.617000	60034741593	657.148000	166894524549
Shell sort	0.016343	5880567	0.057619	20020053	0.092058	36191258
Heap sort	0.013370	8044792	0.052706	26490297	0.138432	45969706
Merge sort	0.054960	4037850	0.260414	13051418	0.383066	22451418
Quick sort	0.011212	3386422	0.048287	10434431	0.132994	18127223
Counting sort	0.001707	465541	0.006603	1265541	0.018610	2065541
Radix sort	0.007562	1700072	0.022573	5100072	0.033952	8500072
Flash sort	0.005839	894994	0.025978	2851150	0.078891	4504447

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	0.153107	100019998	1.347440	900059998	3.752370	2500099998
Insertion sort	0.000072	155318	0.000103	326334	0.000184	601762
Bubble sort	0.086162	100009999	0.871811	900029999	1.846150	2500049999
Shaker sort	0.000496	172268	0.000914	378603	0.001633	606547
Shell sort	0.000423	259489	0.001293	832149	0.001495	1490775
Heap sort	0.000692	669808	0.002169	2236534	0.015137	3924746
Merge sort	0.004860	337226	0.013239	1104458	0.021664	1918922
Quick sort	0.000253	154987	0.000525	501957	0.000996	913886
Counting sort	0.000141	60005	0.000133	180005	0.000195	300005
Radix sort	0.000548	140058	0.002277	510072	0.002461	850072
Flash sort	0.000222	127962	0.000604	383958	0.001039	639964

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	14.927900	10000199998	139.189000	90000599998	396.272000	250000999998
Insertion sort	0.000407	751762	0.000792	1351762	0.000690	1951762
Bubble sort	6.905240	10000099999	60.711200	90000299999	168.971000	250000499999
Shaker sort	0.001694	706547	0.002128	1106547	0.002163	1506547
Shell sort	0.002763	3090777	0.013199	10279721	0.014245	17083549
Heap sort	0.008738	8364882	0.029167	27413341	0.066579	47405008
Merge sort	0.070852	4037850	0.193553	13051418	0.158808	22451418
Quick sort	0.002428	1927727	0.006435	6058264	0.010196	10310773
Counting sort	0.000430	600005	0.004379	1800005	0.012809	3000005
Radix sort	0.007001	1700072	0.020778	6000086	0.044153	10000086
Flash sort	0.002781	1279966	0.008069	3839964	0.010117	6399964

Table 8 – Data order: Nearly sorted - table 2

3.2 Line graphs of running time

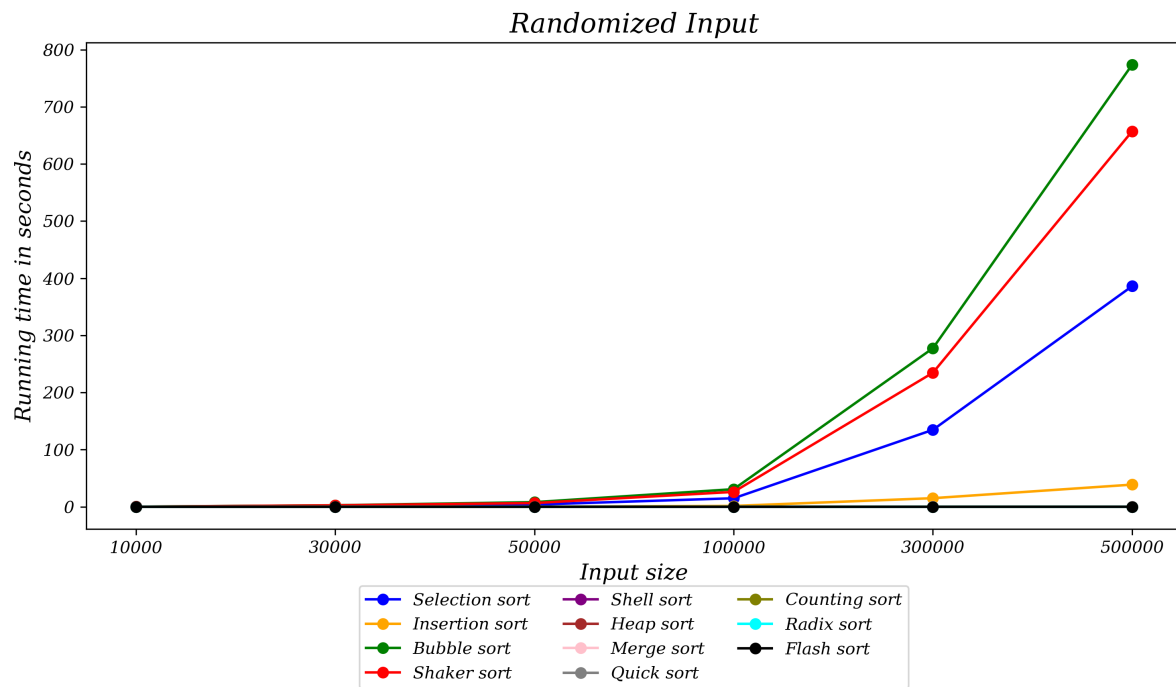


Figure 2 – Line graph of running time for randomized input

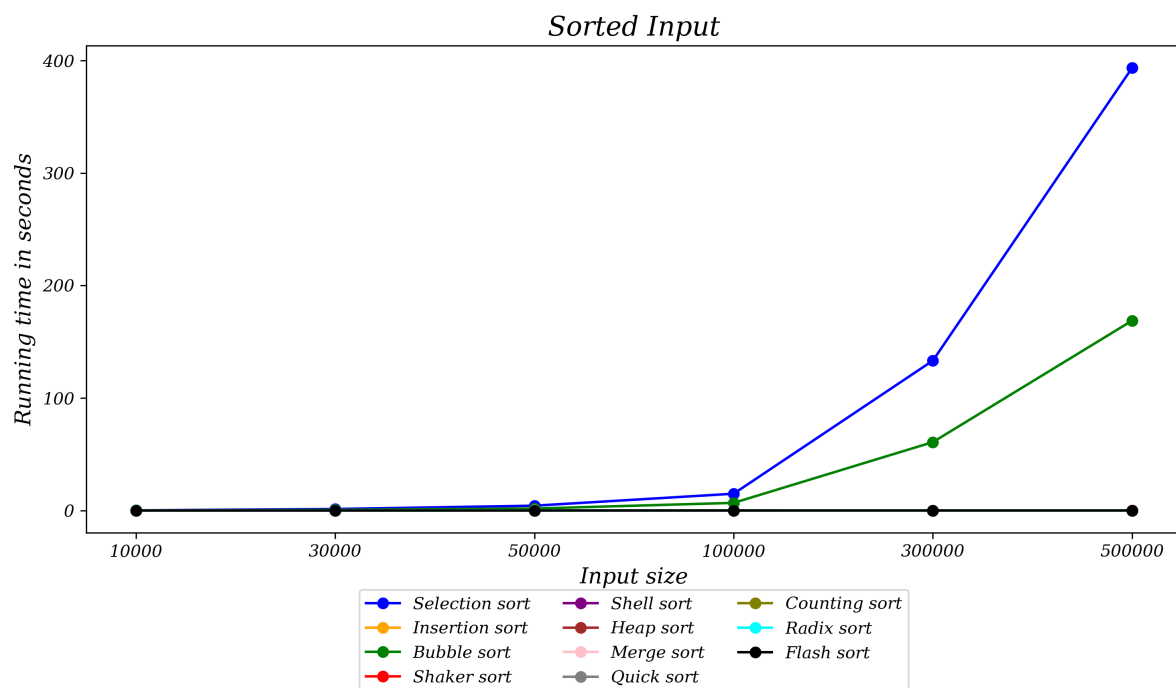


Figure 3 – Line graph of running time for sorted input

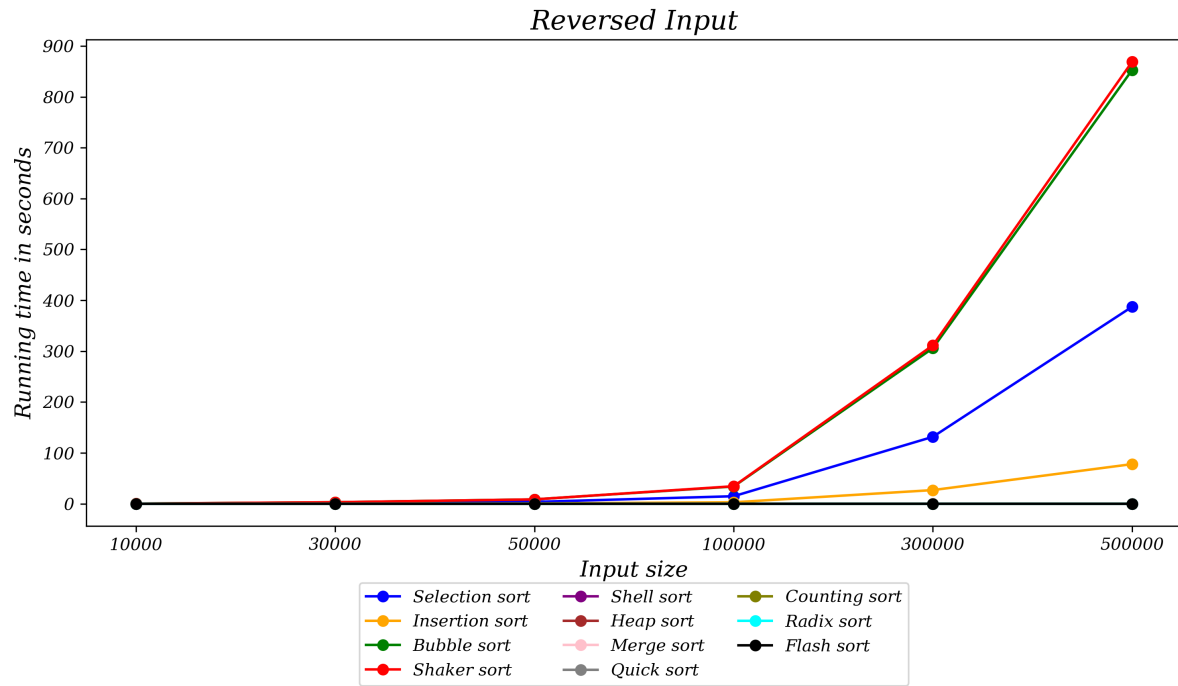


Figure 4 – Line graph of running time for reversed input

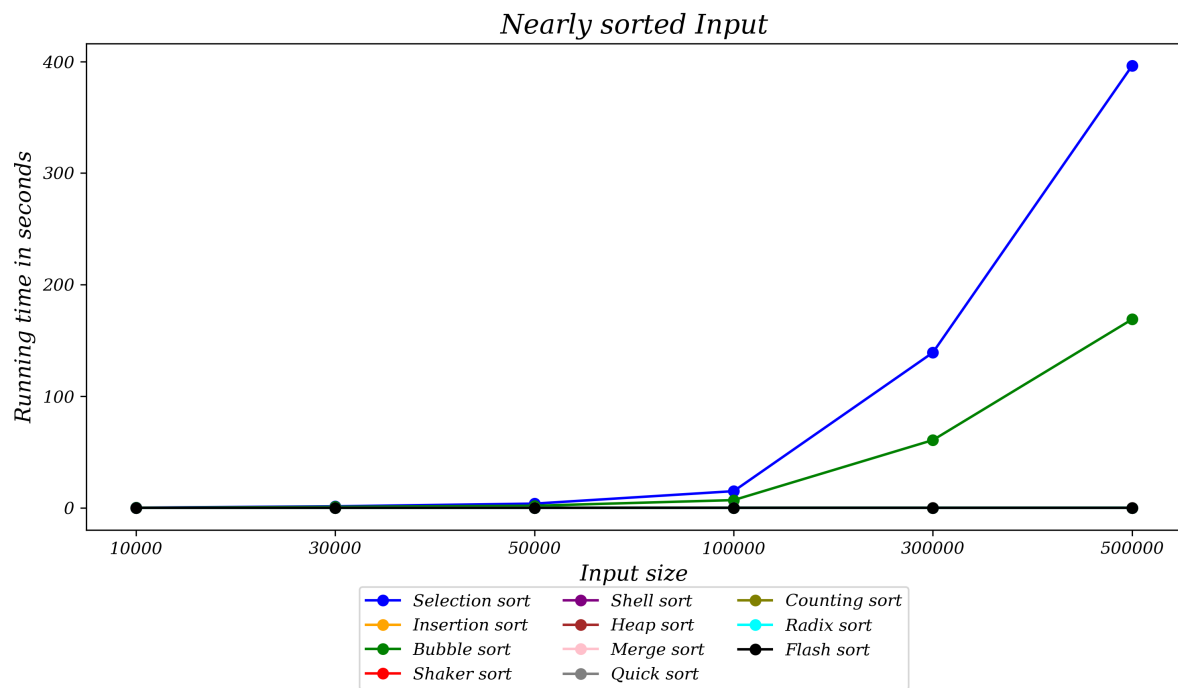


Figure 5 – Line graph of running time for nearly sorted input

3.3 Bar charts of comparisons

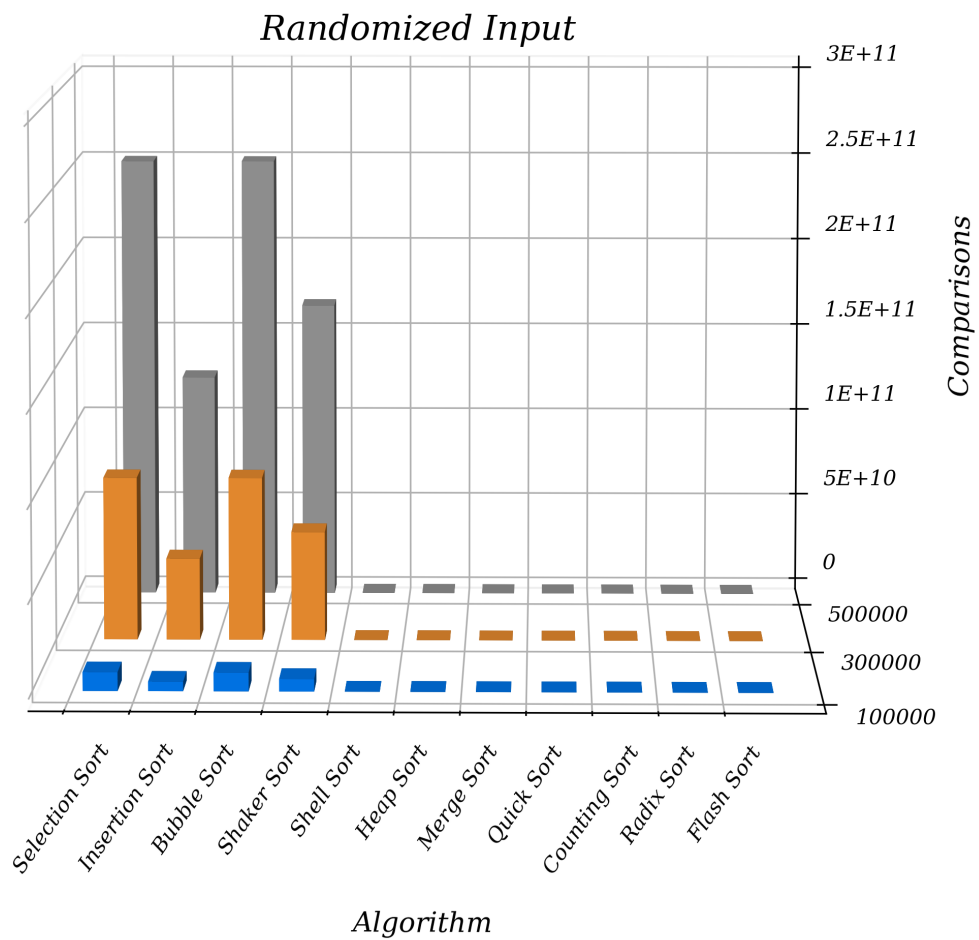


Figure 6 – Bar chart of comparisons for randomized input

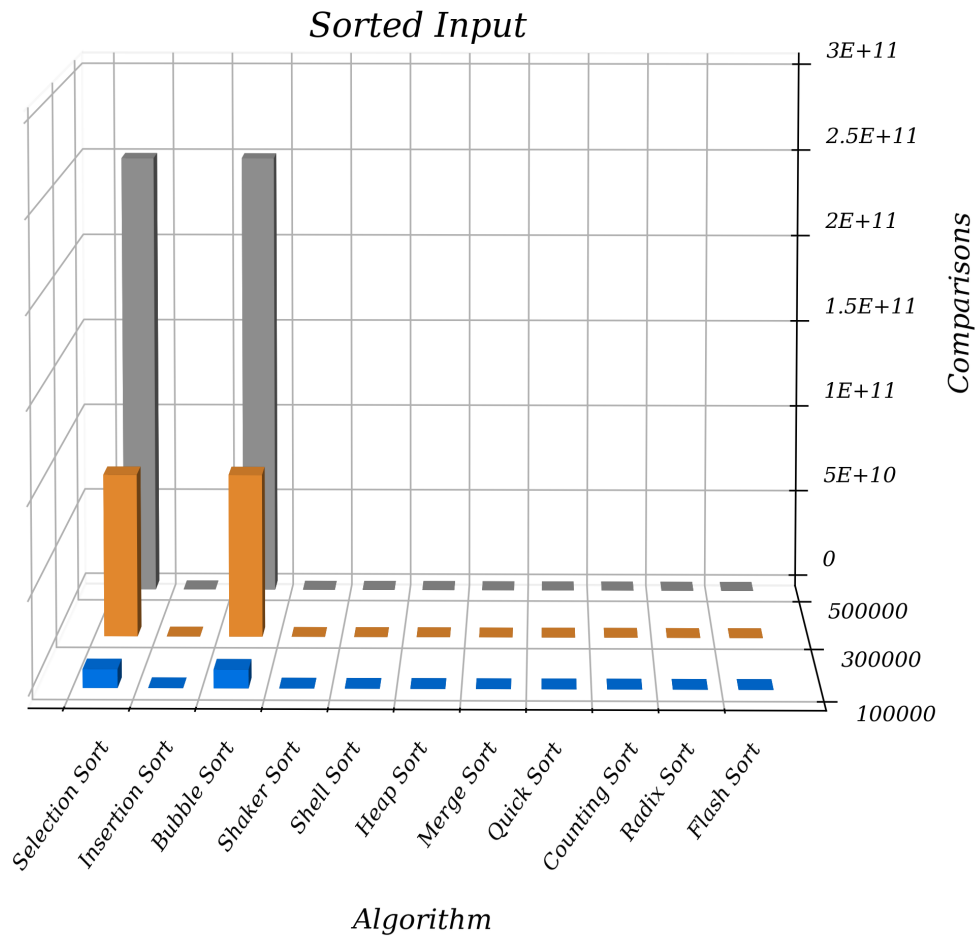


Figure 7 – Bar chart of comparisons for sorted input

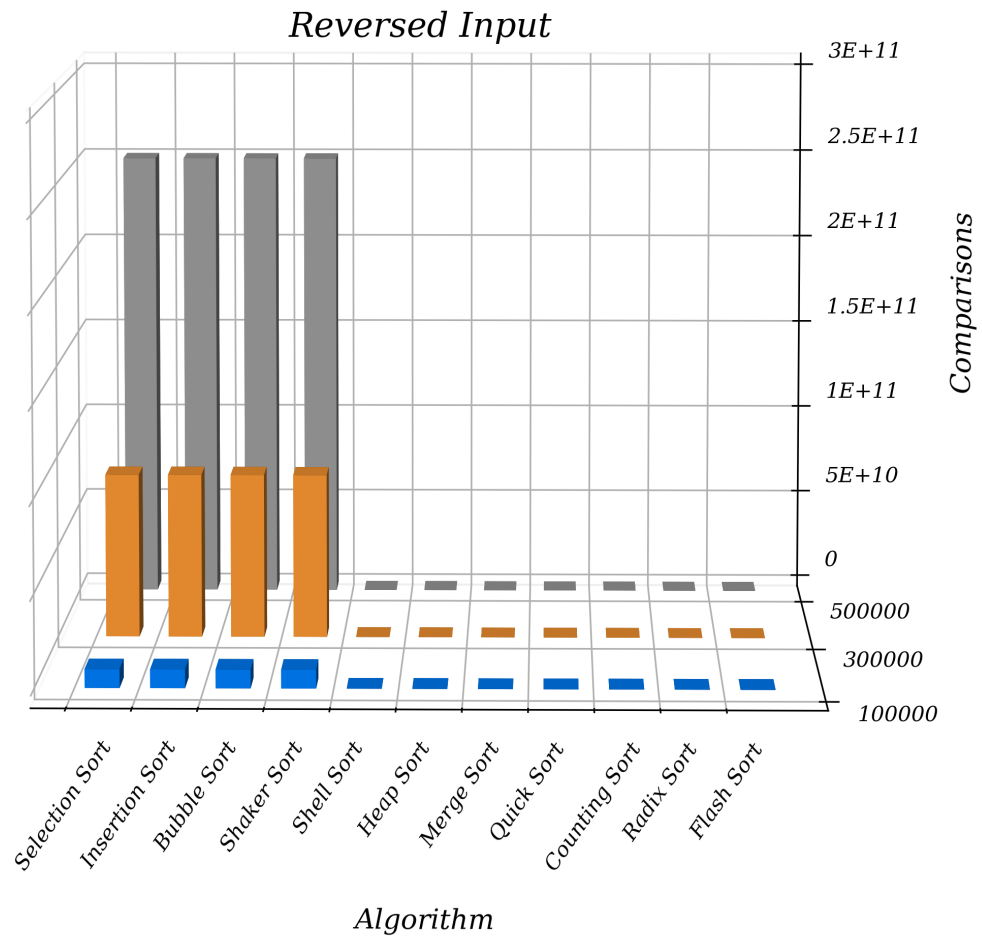


Figure 8 – Bar chart of comparisons for reversed input

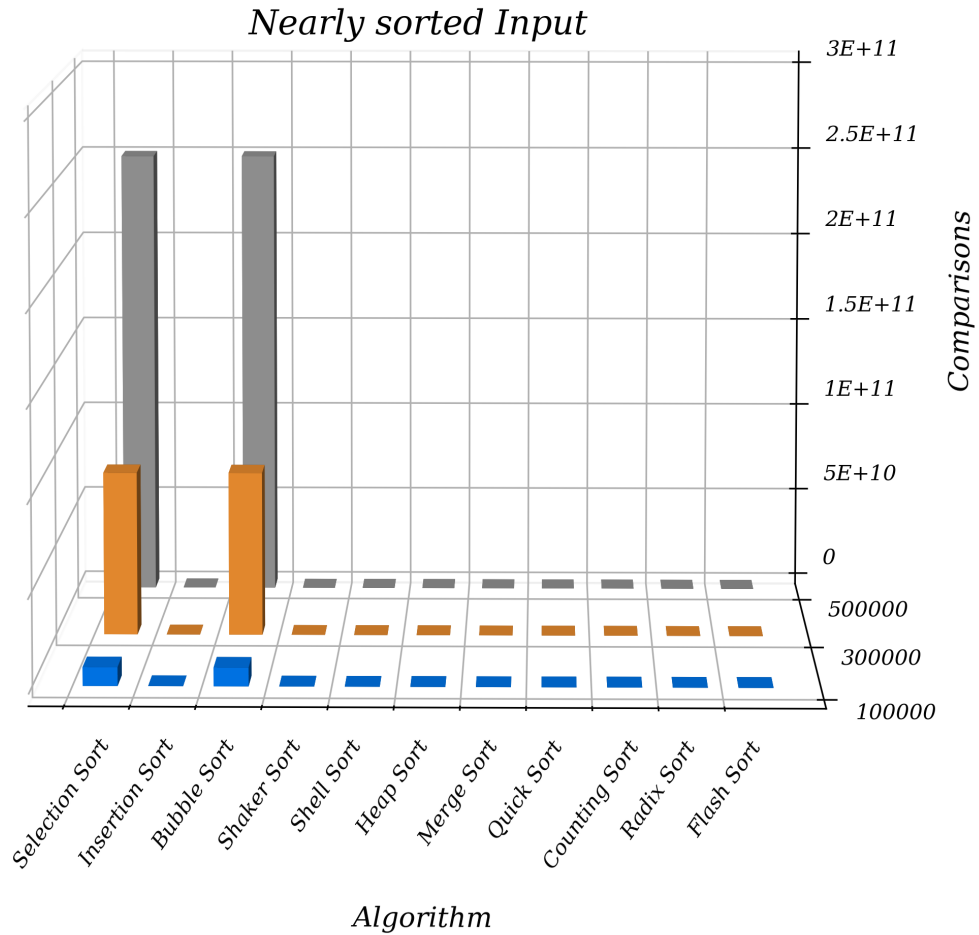


Figure 9 – Bar chart of comparisons for nearly sorted input

3.4 Comments

In-depth scrutiny of the empirical data, as illustrated in the accompanying graphs and charts, allows for a comprehensive evaluation of the performance metrics across a spectrum of sorting algorithms. Significantly, Flash Sort emerges as a paradigm of efficiency, chiefly attributed to the judicious selection of the parameter m , optimally set at approximately $0.43n$. This strategic alignment with theoretical best practices in algorithm design is crucial for maximizing the algorithm's performance potential, particularly in scenarios involving large and diverse datasets.

Contrastingly, Bubble Sort is empirically identified as the antithesis of efficiency within this cohort of algorithms. Its inherent design, characterized by nested iterations, results in a quadratic increase in comparison operations as a function of dataset size. This leads to impractical time complexities, especially in large-scale data applications, rendering it

suboptimal for most practical purposes.

A salient observation is that non-comparative sorting algorithms such as Counting Sort, Radix Sort, and Flash Sort have a distinctive advantage in terms of comparison counts. Their innovative approach circumvents the need for direct element-by-element comparisons, thereby facilitating a more streamlined and efficient sorting process. This characteristic is particularly advantageous in handling large datasets where the reduction in comparative operations can lead to substantial improvements in computational efficiency.

A recurring motif across most sorting algorithms is their enhanced performance with pre-sorted or nearly sorted inputs, which significantly decreases computational overhead. This pattern underscores the potential for optimized sorting in systems where data exhibits partial ordering a priori.

A detailed categorization based on stability further elucidates the nuances of these algorithms:

- **Stable algorithms:**
 - **Selection sort:** Notable for its uniform performance across diverse input scenarios. Its predictability makes it a viable choice in contexts where consistency is paramount.
 - **Shell sort:** Demonstrates an exceptional adaptability to varying data patterns, making it a robust choice for a wide range of applications.
 - **Heap sort:** Its efficiency hinges on the invariant cost of heap construction, a feature that ensures consistent performance across varied input types.
 - **Merge sort:** Exhibits a hallmark of steady processing efficiency, attributed to its divide-and-conquer approach, which remains unaffected by input order.
 - **Radix sort:** The empirical evidence underscores its efficacy in handling integer datasets, aligning with its theoretical design to exploit digit-based sorting.
 - **Flash sort:** Distinguished by its near-linear complexity, it stands out for its efficiency and stable performance across diverse data ranges.
- **Unstable algorithms:**
 - **Insertion sort:** Displays a performance dichotomy; it is highly efficient with nearly sorted data but exhibits marked inefficiency with randomized or reversed datasets.
 - **Bubble sort:** Its simplicity belies its inefficiency for large datasets, with performance varying substantially based on initial data order.
 - **Shaker sort:** Similar to Insertion Sort in its performance characteristics, showing marked efficiency with pre-sorted data but faltering with disordered datasets.
 - **Quick sort:** Notable for its general efficiency, but its performance is highly sensitive to pivot selection, with suboptimal choices leading to significant performance degradation.

- **Counting sort:** While fast, its effectiveness is constrained in situations where the value range u greatly exceeds the number of elements n , as explored in section 2.9.

This comprehensive analysis not only delineates the distinct operational paradigms and efficiencies of each algorithm but also highlights the criticality of aligning algorithm selection with specific data characteristics and application requirements. Such strategic alignment is indispensable for optimizing sorting operations in real-world data processing scenarios.

4 Project organization and Programming notes

4.1 Project organization

Figure below shows files in my project.


















Name	Date modified	Type	Size
 AdvancedSort.cpp	11/30/2023 8:35 AM	CPP File	5 KB
 AdvancedSort.h	11/29/2023 2:43 PM	H File	1 KB
 AlgorithmTesting.cpp	12/2/2023 6:33 PM	CPP File	3 KB
 AlgorithmTesting.h	11/29/2023 5:41 PM	H File	1 KB
 BasicSort.cpp	11/30/2023 9:01 PM	CPP File	5 KB
 BasicSort.h	11/30/2023 8:26 AM	H File	1 KB
 CommandProcess.cpp	11/30/2023 9:01 PM	CPP File	12 KB
 CommandProcess.h	11/29/2023 4:09 PM	H File	1 KB
 DataGenerator.cpp	11/30/2023 8:31 AM	CPP File	2 KB
 DataGenerator.h	11/29/2023 2:43 PM	H File	1 KB
 FlashSort.cpp	11/30/2023 9:08 AM	CPP File	3 KB
 FlashSort.h	11/29/2023 2:43 PM	H File	1 KB
 main.cpp	11/30/2023 9:07 PM	CPP File	2 KB
 makefile	11/30/2023 9:26 AM	File	2 KB
 NoComparisonSort.cpp	12/2/2023 6:47 PM	CPP File	3 KB
 NoComparisonSort.h	11/29/2023 2:43 PM	H File	1 KB
 Utilities.h	11/29/2023 4:36 PM	H File	1 KB

Figure 10 – 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.

4.2 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://brilliant.org/wiki/heap-sort/>
- [11] <https://www.programiz.com/dsa/heap-sort>
- [12] <https://www.educative.io/blog/data-structure-heaps-guide>
- [13] <https://www.programiz.com/dsa/merge-sort>
- [14] <https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/overview-of-merge-sort>
- [15] <https://www.geeksforgeeks.org/quick-sort/>
- [16] <https://www.geeksforgeeks.org/iterative-quick-sort/>
- [17] <https://www.geeksforgeeks.org/counting-sort/>
- [18] <https://brilliant.org/wiki/counting-sort/>
- [19] <https://www.simplilearn.com/tutorials/data-structure-tutorial/counting-sort-algorithm>
- [20] <https://www.interviewcake.com/concept/java/counting-sort>
- [21] <https://brilliant.org/wiki/radix-sort/>
- [22] <https://www.programiz.com/dsa/radix-sort>
- [23] <https://www.w3resource.com/javascript-exercises/searching-and-sorting-algorithm/searching-and-sorting-algorithm-exercise-12.php>
- [24] <https://www.neubert.net/FS0Intro.html>