VIETNAM NATIONAL UNIVERSITY
HO CHI MINH CITY

UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

Project Report

Topic: SORTING ALGORITHMS

Subject: Data Structures and Algorithms

Students:
Vo Tran Quoc Duy
CLC07 - 23127359

Lecturers:
LT. Bui Huy Thong
LT. Nguyen Ngoc Thao

Thursday 27th June, 2024

Contents

List of Algorithms

1 Algorithms Presentation

1.1 Heap Sort

1.1.1 Ideas

The heap sort is comparison-based sorting technique base on Binary Heap data structure - a complete Binary Tree, as we only sort the data in ascending order so the Max Binary Heap, whose the key at the root is maximum among all keys, is taken in to use. It is similar to the selection sort where we find the minimum element and place it at the beginning then repeat the same process for the remaining elements.

1.1.2 Descriptions

Using these following steps to perform the heap sort:

- Step 1: Build a heap data structure from the given input array using heapify.
- Step 2: Swap the root element, which is now the largest element, with the last element of the heap.
- Step 3: Heapify the remaining elements of the heap except for the last element.

Repeat Step 2 and Step 3 until the heap contains only 1 element.

Algorithm 1 Heap Sort

```
function Swap(a, b)
    temp ← a
    a ← b
    b ← temp
end function

function Heapify(array, n, i)
    largest ← i
    left ← 2 * i + 1
    right ← 2 * i + 2

    if left < n && array[largest] < array[left] then
        largest ← left
    end if

    if right < n && array[largest] < array[right] then
        largest ← right
    end if

    if largest ≠ i then
        Swap(array[largest], array[i])
        Heapify(array, n, largest)
    end if
end function
```

```
function heapSort(array, n)
    for i ← 0 to n − 1 do
        Heapify(array, n, i)
    end for

    for i ← n − 1 to 0 do
        Swap(array[0], array[i])
        Heapify(array, i, 0)
    end for
end function
```

1.1.3 Time Complexity

The problem size is: n - the number of elements of the array.
The process of converting an array of n elements to a heap data structure takes $O(log_2n)$ logarithmic time - the $O(log_2n)$ factor is the height of the binary tree, then the last element will be extracted from the array so the size is now $n-1$ and the converting process will continue until the array only has 1 element left. So the total time complexity is:

$$O(log_2n) + O(log_2(n-1)) + ... + O(log_21) = O(log_2(n!))$$

Time complexity:

- **Best case:** $O(nlogn)$
- **Average case:** $O(nlogn)$
- **Worst case:** $O(nlogn)$

1.1.4 Space Complexity

Since heap sort is an in-place sorting algorithm, it does not require additional storage.
Space complexity: $O(n)$

1.2 Radix Sort

1.2.1 Ideas

The radix sort is a non comparison-based sorting technique. To achieve the finally sorted order, the radix sort distributes the elements into buckets based on each digit's value and repeatedly sorting the elements by their significant digits, from the least significant digit to the most significant digit. The radix sort uses the counting sort to sort the list considering a certain digit.

1.2.2 Descriptions

Using these following steps to perform the radix sort:

- Step 1: Find the largest value in the array.
- Step 2: Iterate *exp* times, *exp* is the number of digits of the largest value, once for each significant place. In each iteration, performing the counting sort to sort the elements.

Algorithm 2 Radix Sort

function GETMAX(*array*, *n*)
    $Max \leftarrow array[0]$

    for $i \leftarrow 0$ **to** $n - 1$ **do**
        if $Max < array[i]$ **then**
            $Max \leftarrow array[i]$
        end if
    end for

    return Max
end function

function COUNTSORT(*array*, *n*, *exp*)
    output is an array of n integers
    count is an array of 10 integers, the value of each element is 0

    for $i \leftarrow 0$ **to** $n - 1$ **do**
        $count[\frac{array[i]}{exp} \mod 10] \leftarrow count[\frac{array[i]}{exp} \mod 10] + 1$
    end for

    for $i \leftarrow 1$ **to** 10 **do**
        $count[i] \leftarrow count[i] + count[i - 1]$
    end for

    for $i \leftarrow n - 1$ **to** 0 **do**
        $output[count[\frac{array[i]}{exp} \mod 10] - 1] = array[i]$

        $count[\frac{array[i]}{exp} \mod 10] \leftarrow count[\frac{array[i]}{exp} \mod 10] - 1$
    end for

    for $i \leftarrow 0$ **to** $n - 1$ **do**
        $array[i] \leftarrow output[i]$
    end for
end function

function RADIXSORT(*array*, *n*)
    $max \leftarrow$ GETMAX$(array, n)$

    for $exp \leftarrow 1$ **to** $\frac{max}{exp} > 1$ **do**
        COUNTSORT(*array*, *n*, *exp*)
    end for
end function

1.2.3 Time Complexity

The problem size are:

- k - the number of digits of the largest value.
- n - the number of elements of the array.

The counting sort used in the radix sort takes $O(n+b)$ logarithmic time with b is the base of the number system but in this case b is a constant, and since we have to perform the counting sort k times so the time complexity will be $O(k*n)$ in all cases.
Time complexity:

- **Best case:** $O(k*n)$
- **Average case:** $O(k*n)$
- **Worst case:** $O(k*n)$

1.2.4 Space Complexity

Since we have to create an auxiliary space to store each digit value and copy the elements back to the original array so the space complexity will be $O(n+b)$.
Space complexity: $O(n+b)$

1.3 Flash Sort

1.3.1 Ideas

The flash sort is a comparison-based sorting technique, a more efficient way to implement the bucket sort as it creates buckets and rearrange all elements according to buckets. Lastly, sorting each bucket using the insertion sort.

1.3.2 Descriptions

Using these following steps to perform the flash sort:

- Step 1: Find the positions of the minimum and the maximum value in the array.
- Step 2: Divide the array into m buckets.
- Step 3: Count the number of elements in each bucket.
- Step 4: Convert the counts of each bucket into prefix sum.
- Step 5: Rearrange all the elements of each bucket in a position $array_{\text{i}}$ where $bucket_{\text{bucketId - 1}} < i < bucket_{\text{bucketId}}$.
- Step 6: Sort each bucket using the insertion sort.

Algorithm 3 Flash Sort

function FLASHSWAP(*array*, *n*, *bucket*, *m*, *maxPos*, *minPos*, *c*)
    $bucketId \leftarrow m-1$
    $move \leftarrow 0, i \leftarrow 0, flash \leftarrow 0, k \leftarrow 0, hold \leftarrow 0$

    SWAP(*array[maxPos]*, *array[0]*)
    while $move < n-1$ **do**
        while $i > bucketId - 1$ **do**

```
            i ← i + 1
            bucketId ← c ∗ (array[i] − array[minPos])
        end while

        flash ← array[i]
        if bucketId < 0 then
            break
        end if
        while i ≠ bucketID do
            bucketId ← c ∗ (array[i] − arrray[minPos]
            bucketId ← bucketId − 1
            k ← bucketId
            hold ← array[k]
            array[k] ← flash
            flash ← hold
            move ← move + 1
        end while
    end while
end function

function FLASHSORT(array, n)
    m ← 0.45 ∗ n
    bucket is an array of m integers, the value of each element is 0
    maxPos ← 0, minPos ← 0

    for i ← 0 to n − 1 do
        if array[maxPos] < array[i] then
            maxPos ← i
        end if

        if array[minPos] > array[i] then
            minPos ← i
        end if
    end for

    c ← (m−1) / (array[maxPos]−array[minPos])
    for i ← 0 to n − 1 do
        k ← c ∗ (array[i] − array[minPos])
        bucket[k] ← bucket[k] + 1
    end for

    for i ← 1 to m − 1 do
        bucket[k] ← bucket[k] + bucket[k − 1]
    end for

    FLASHSWAP(array, n, bucket, m, maxPos, minPos, c)
    INSERTIONSORT(array, n)
end function
```

1.3.3 Time Complexity

The problem size is: n - the number of elements of the array.
The best and average time complexity is $O(n)$. However, there is a possibility that nearly all the elements are belong to one bucket which can make the time complexity go up to the worst time complexity of the insertion sort, which is $O(n^2)$.
Time complexity:

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

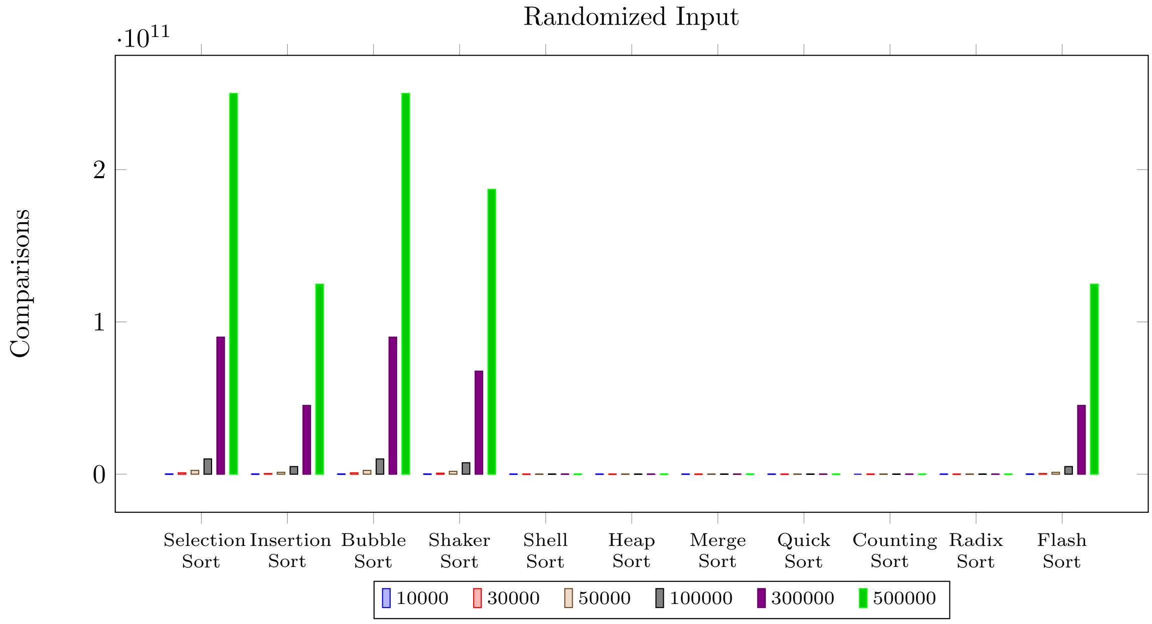
1.3.4 Space Complexity

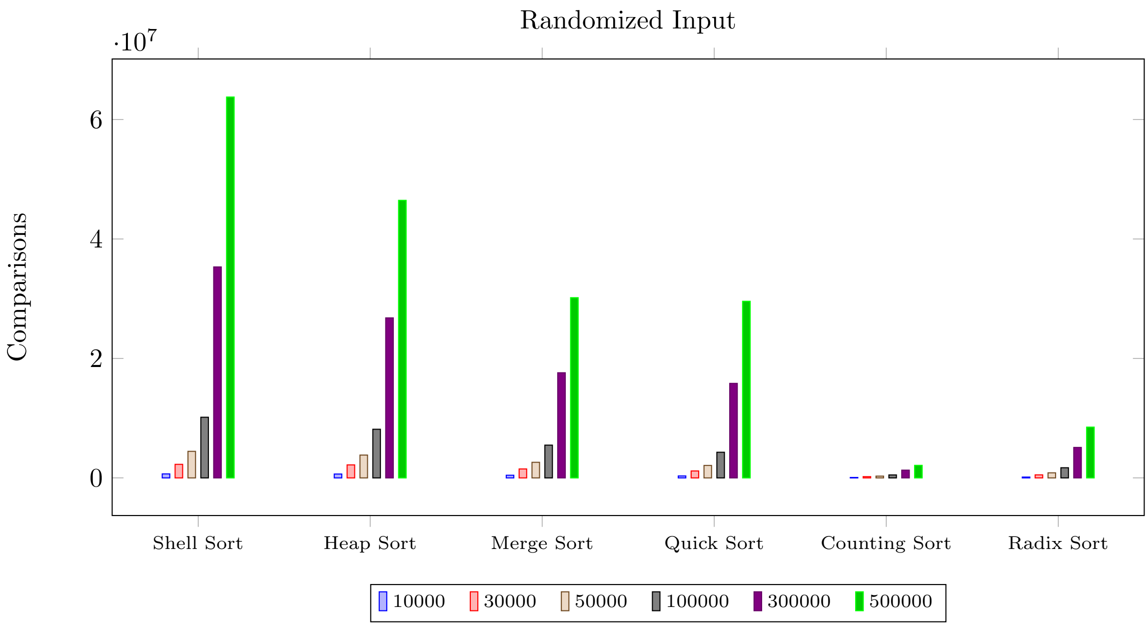
Since the flash sort is an in-place sorting algorithm, it does not require additional storage.
Space complexity: $O(n)$

2 Chart Draw and Comment

2.1 Random Data

Bar Chart:



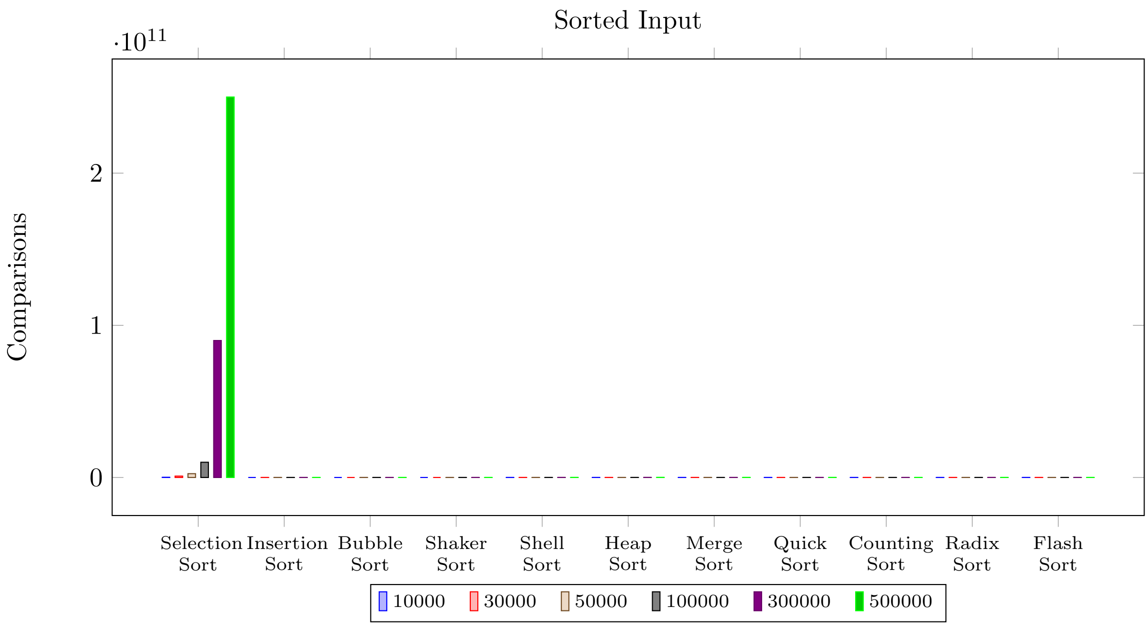


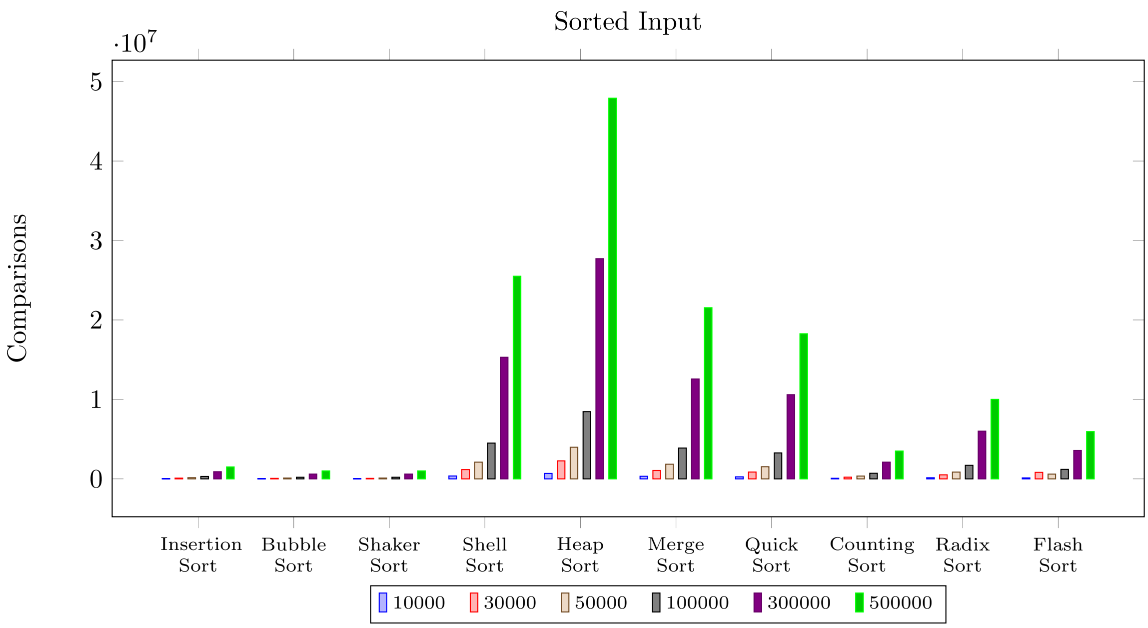
Comments:

- The bar chart above depicted the number of comparisons among eleven sorting techniques. It can be seen from the bar chart that selection sort and bubble sort both needs a significant amount of comparisons to deal with large data size. Counting sort, on the other hand, takes the least amount of comparisons to sort large data size.

2.2 Sorted Data

Bar Chart:



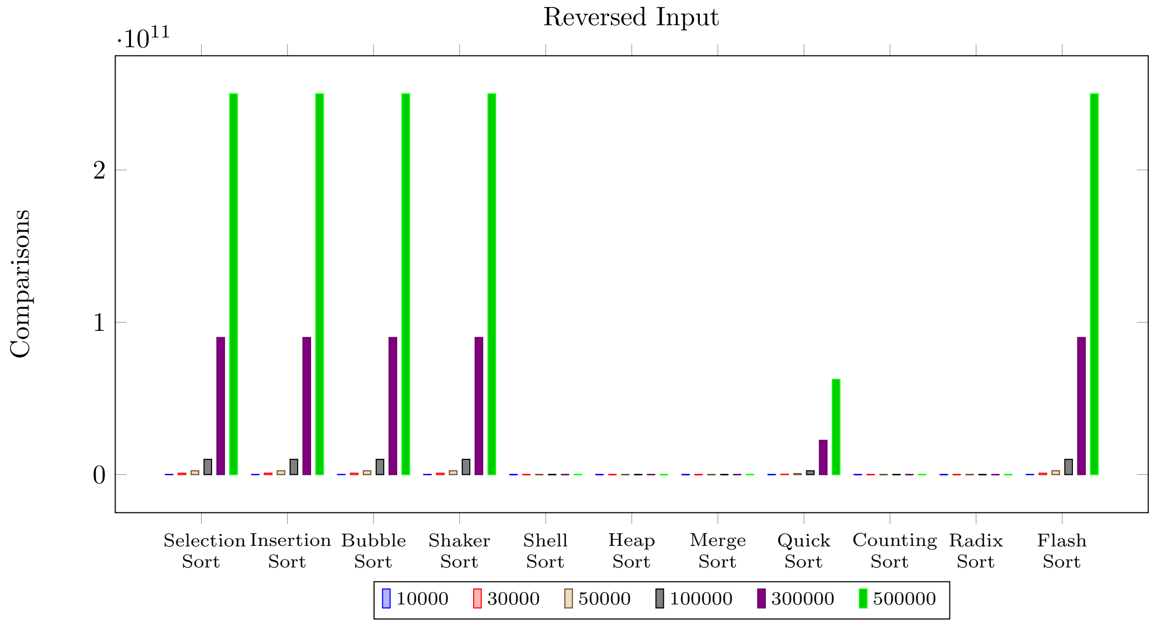


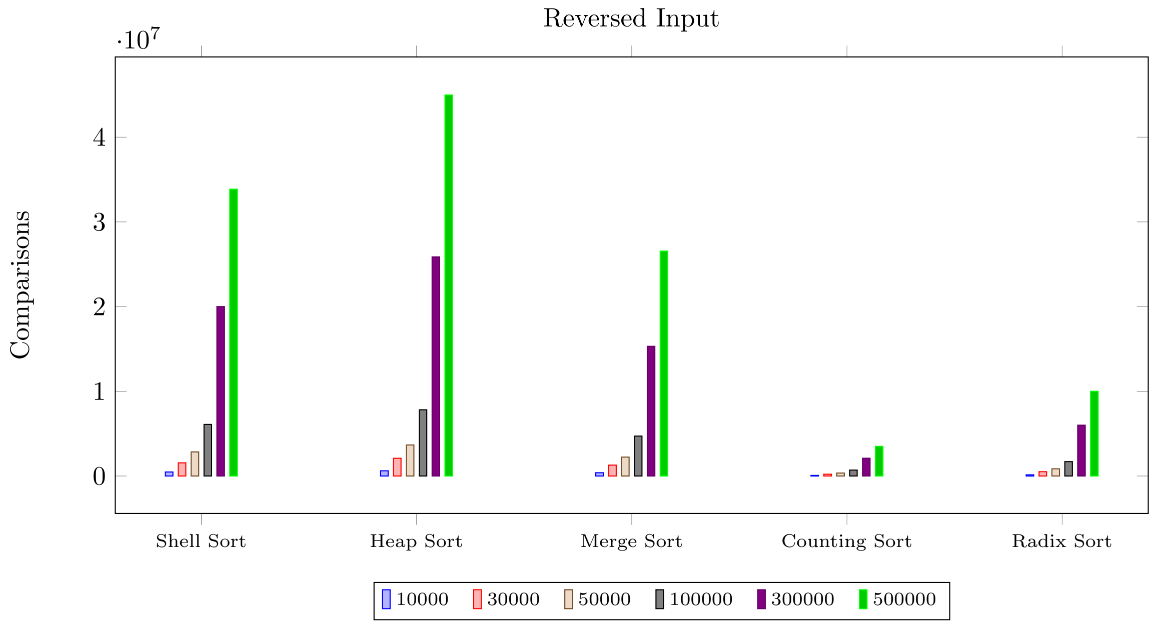
Comments:

- The bar chart illustrated the amount of comparisons carried out by sorting techniques. It can be noticed from the bar chart that mostly all sorting techniques take a small number of comparisons to sort the sorted input except for selection sort which has the most significant number of comparisons.

2.3 Reversed Data

Bar Chart:



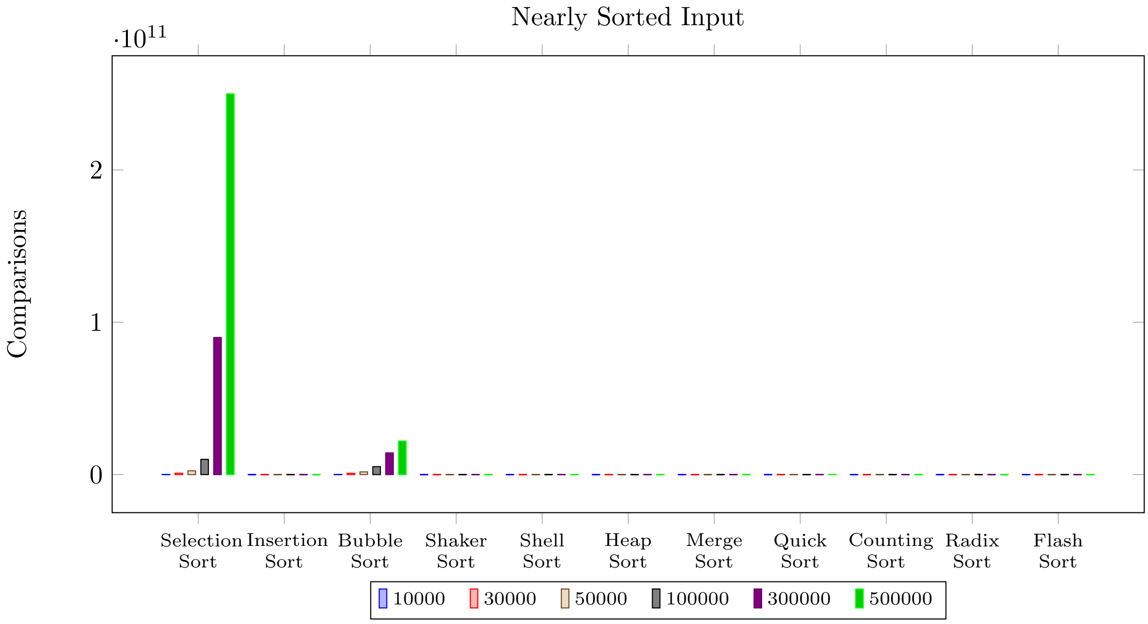


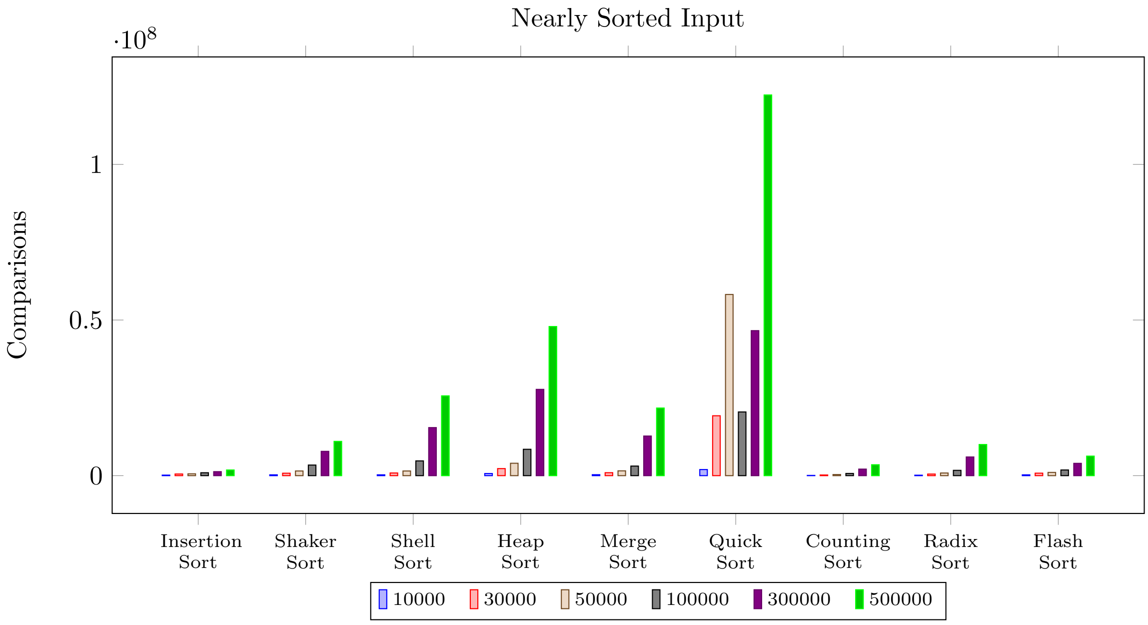
Comments:

- The bar chart above depicted the number of comparisons carried out by sorting techniques. It is noticeable that selection sort, insertion sort, bubble sort, shaker sort and flash sort all take the same amount of comparisons to deal with large data size of reversed input.

2.4 Nearly Sorted Data

Bar Chart:





Comments:

- The bar chart demonstrated the number of comparisons taken by eleven sorting techniques. It can be seen from the bar chart that selection sort needs a significant amount of comparisons to deal with large data size. The number of comparisons usually increase with the data size, however, the comparisons carried out by quick sort at the size of 50000 are higher than that of 100000 and 300000.

2.5 Overall Comments:

In overall, the number of comparisons usually increase with the size of the data input, moreover, among four types of data input, selection sort often takes the most amount of comparisons compared to other sorting techniques and counting sort, on the other hand, takes the least amount of comparisons.