Course: Data Structure & Algoriths

Term 2 – Year 2016 – 2017

GROUP EXERCISE 1

Date: March 9th 2017

<u>Group ID:</u> 8

<u>Group Name:</u> Expecto Patronum

1. <u>Group Information:</u>

No.	Student ID	Name
1	1512223	Nguyễn Lê Hưng
2	1512222	Nguyễn Duy Hưng
3	1512002	Lê Dương Tuấn Anh

2. <u>Questions:</u>

i. Implement the following sorting algorithms using C/C++:

a. <u>Insertion Sort</u>

```
void InsertionSort(long* arr, long size)
{
	long key, i, j;
	for (i = 1; i < size; i++)
	{
		key = arr[i];
		j = i - 1;
		while ((j >= 0) && (arr[j] > key))
			arr[j + 1] = arr[j--];
		arr[j + 1] = key;
	}
}
```

b. <u>Merge Sort</u>

```
void Merge(long* arr, long left, long mid, long right)
{
	long *Temp = new long[right - left + 1];
	long i, j, k;
	for (i = 0, j = left, k = mid + 1; (j <= mid && k <=
right); i++)
	{
		if (arr[j] < arr[k])
			Temp[i] = arr[j++];
		else
```

```
                Temp[i] = arr[k++];
        }
        while (j <= mid)
            Temp[i++] = arr[j++];
        while (k <= right)
            Temp[i++] = arr[k++];
        for (i = 0; i <= right - left; i++)
            arr[i + left] = Temp[i];
        delete Temp;
 }

 void MergeSort(long* arr, long left, long right)
 {
        long mid;
        if (left < right)
        {
            mid = (left + right) / 2;
            MergeSort(arr, left, mid);
            MergeSort(arr, mid + 1, right);
            Merge(arr, left, mid, right);
        }
 }
```

c. Quick Sort

```
long Partition(long* arr, long left, long right)
{
        long pivot = (left + right) / 2;
        long i = left;
        long j = right;
        while (i < j)
        {
            while (arr[i] < arr[pivot])
                i++;
            while (arr[j] > arr[pivot])
                j--;
            if (i < j)
            {
                Swap(arr[i], arr[j]);
                i++;
                j--;
            }
        }
        return pivot;
}

void QuickSort(long* arr, long left, long right)
{
        if (left < right)
        {
            long p = Partition(arr, left, right);
            QuickSort(arr, left, p);
            QuickSort(arr, p + 1, right);
```

```
	}
}
```

d. Radix Sort

```
void RadixSort(long* arr, long size)
{
	long i, m = arr[0], exp = 1;
	long *Temp = new long[size];
	for (i = 0; i < size; i++)
	{
		if (arr[i] > m)
			m = arr[i];
	}
	while (m / exp > 0)
	{
		long bucket[10] = { 0 };
		for (i = 0; i < size; i++)
			bucket[arr[i] / exp % 10]++;
		for (i = 1; i < 10; i++)
			bucket[i] += bucket[i - 1];
		for (i = size - 1; i >= 0; i--)
			Temp[--bucket[arr[i] / exp % 10]] = arr[i];
		for (i = 0; i < size; i++)
			arr[i] = Temp[i];
		exp *= 10;
	}
	delete Temp;
}
```

e. Counting Sort

```
void countSort(char arr[])
{
    int count[RANGE + 1], i; //RANGE: constant = 1e7, we assume!
    memset(count, 0, sizeof(count));
    for(i = 0; arr[i]; ++i)
        ++count[arr[i]];
    for (i = 1; i <= RANGE; ++i)
        count[i] += count[i-1];
    for (i = 0; arr[i]; ++i)
    {
        output[count[arr[i]]-1] = arr[i];
        --count[arr[i]];
    }
    for (i = 0; arr[i]; ++i)
        arr[i] = output[i];
}
```

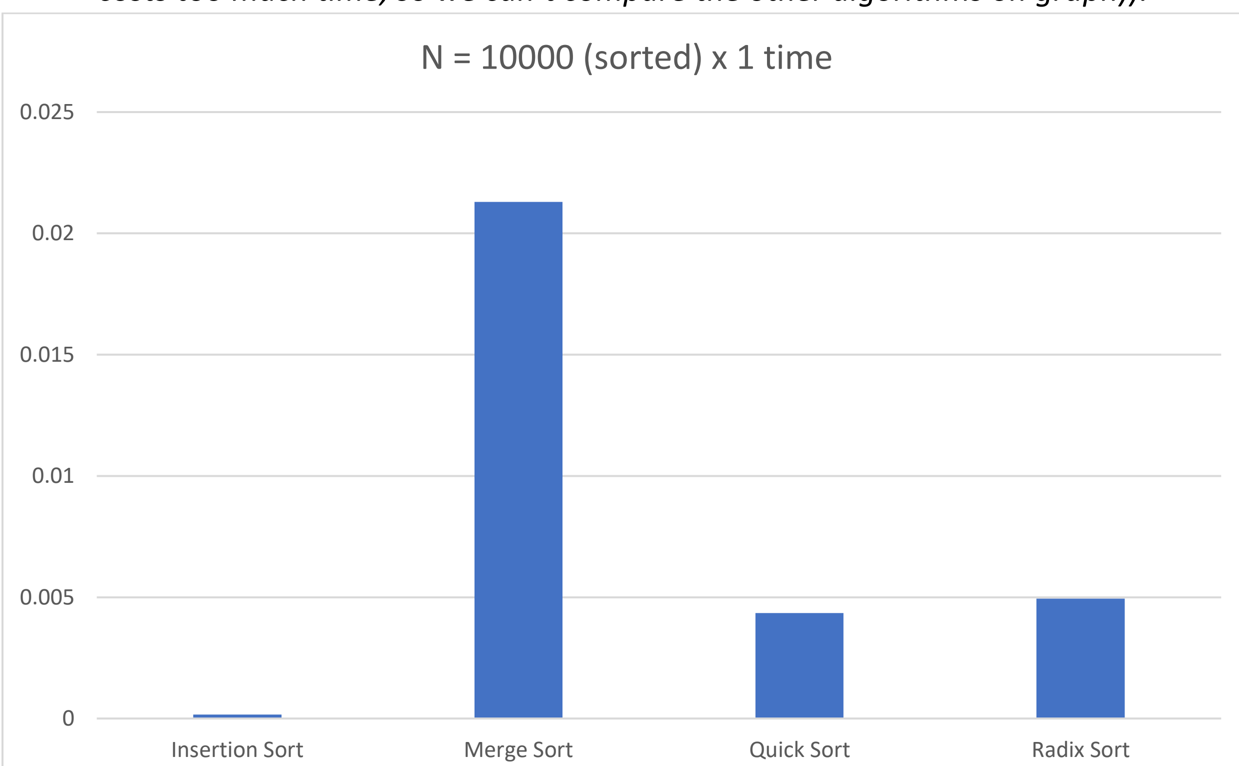
ii. The running time of these Sorting Algorits:

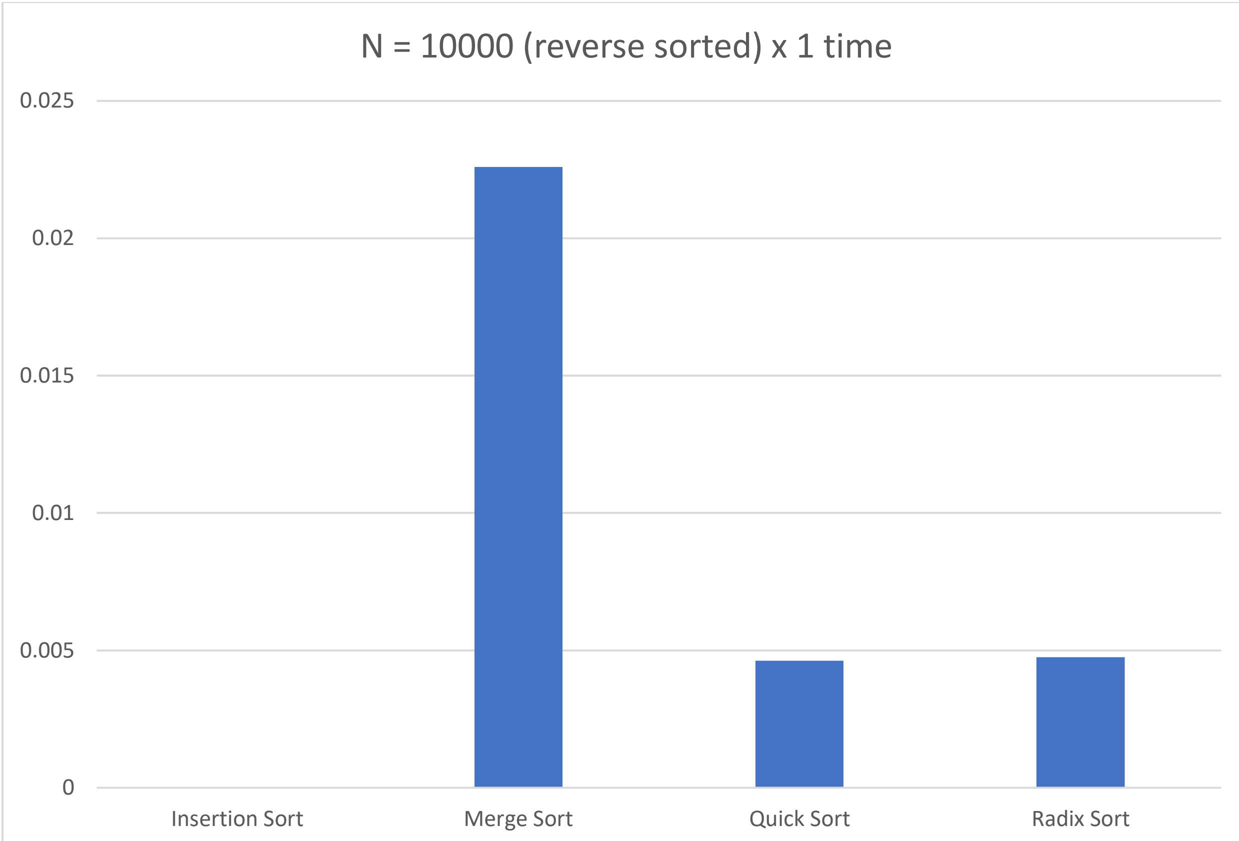
(We took average-times of 10 times we ran the algorithm)

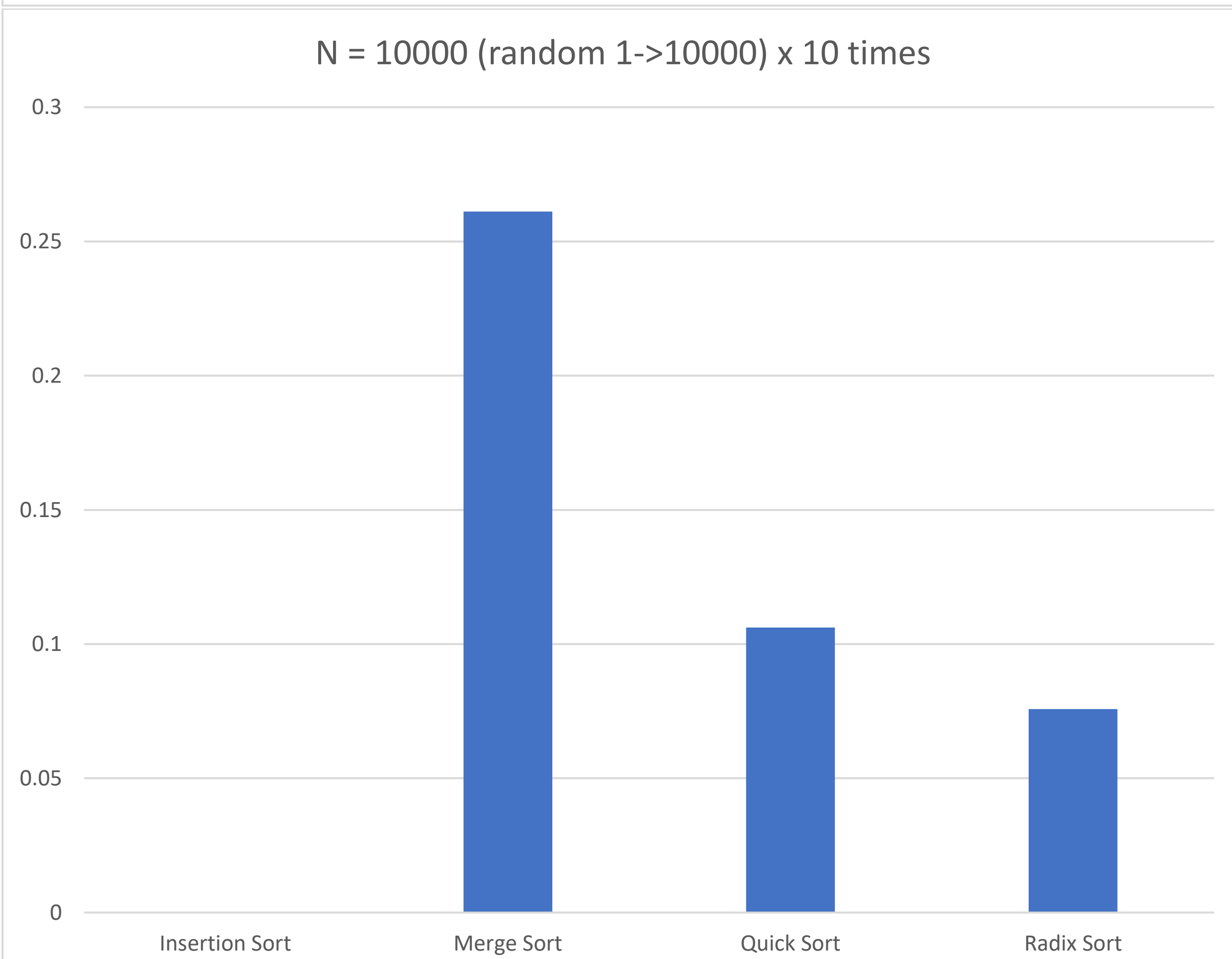
	Insertion Sort	Merge Sort	Quick Sort	Radix Sort
N = 10000 (sorted) x 1 time	0.000161s	0.021294s	0.004345s	0.004947 s
N = 10000 (reverse sorted) x 1 time	0.514267 s	0.022597 s	0.004622 s	0.004753 s
N = 10000 (random 1->10000) x 10 times	2.629274 s	0.261091 s	0.106150 s	0.075821 s
N = 20000 (random 1->20000) x 10 times	10.541194 s	0.521305 s	0.216198 s	0.192793 s
N = 30000 (random 1->30000) x 10 times	24.450407 s	0.793444 s	0.344260 s	0.252208 s
N = 5000000 (random 1->5000000) x 10 times	(too slow)	145.667489 s	23.222543 s	39.818839 s

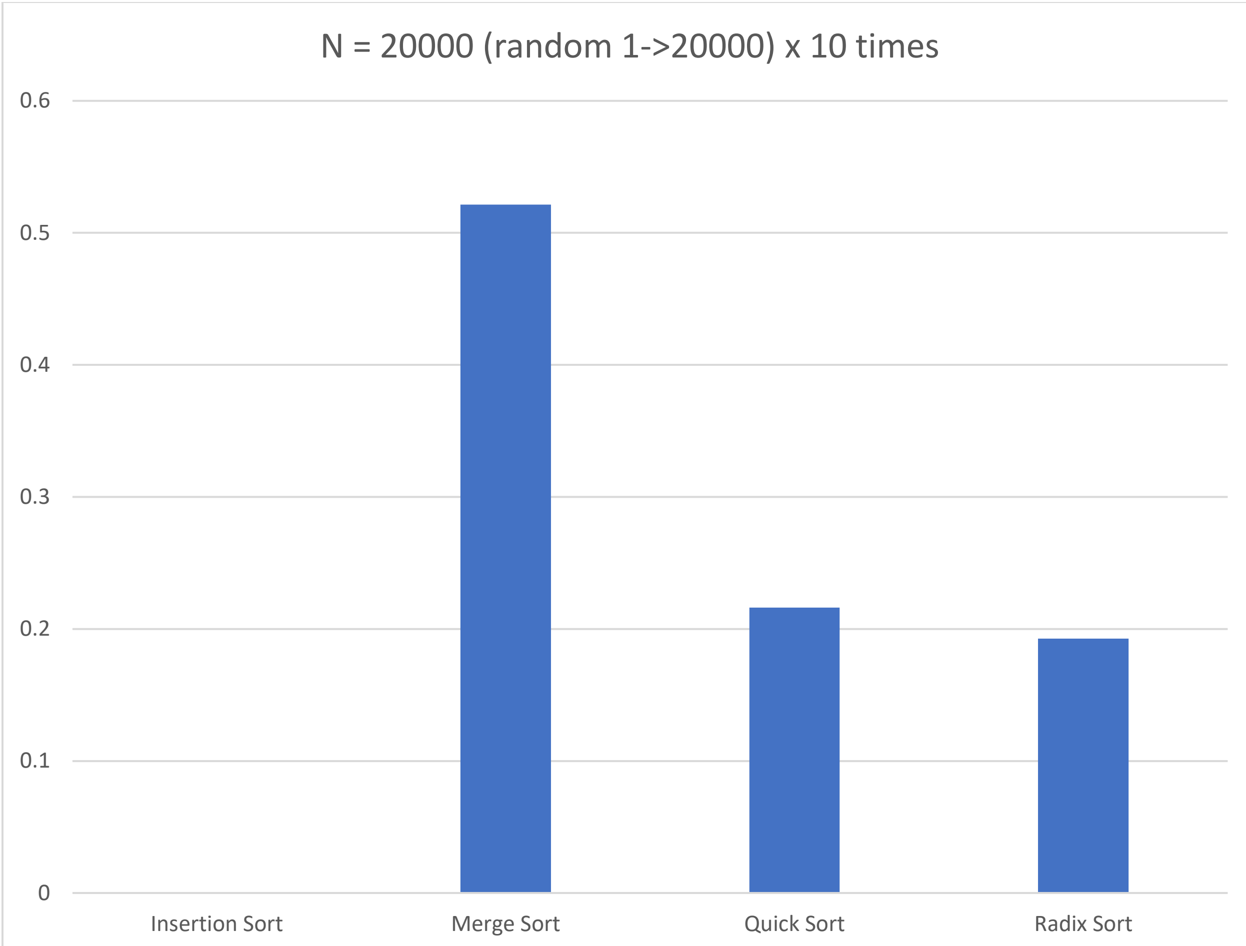
<u>Graph to describe more clearly:</u>

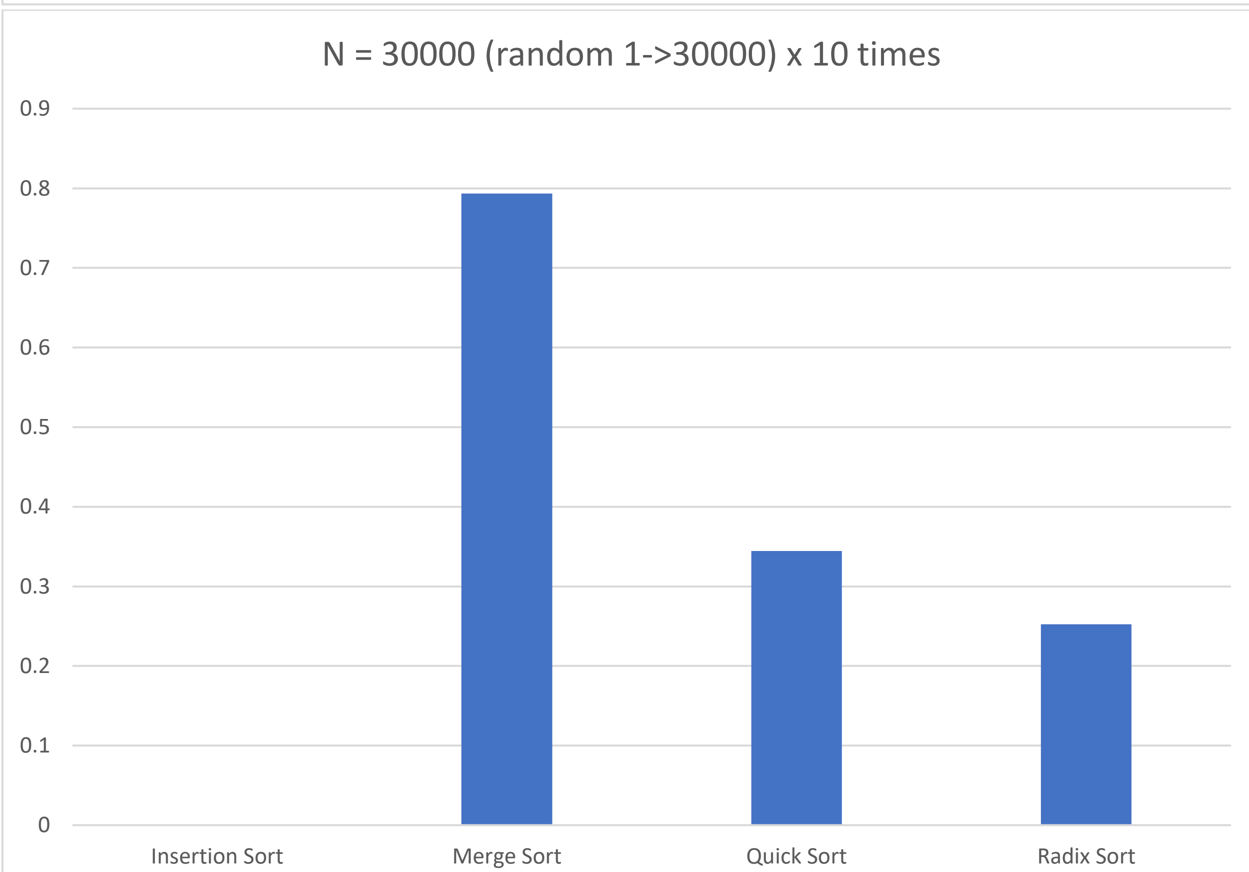
(With n >= 30000, we don't graph the running time of Insertion Sort (because it costs too much time, so we can't compare the other algorithms on graph)).

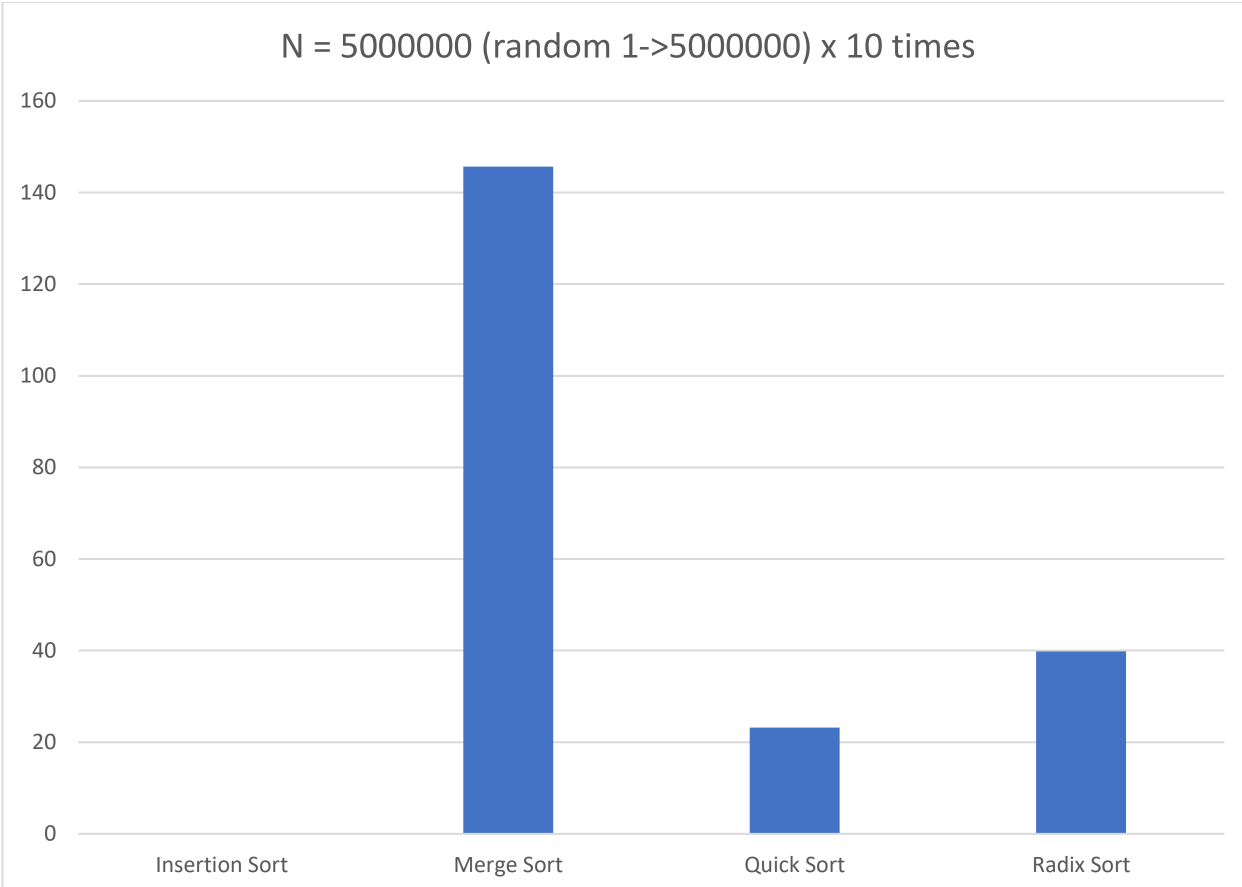


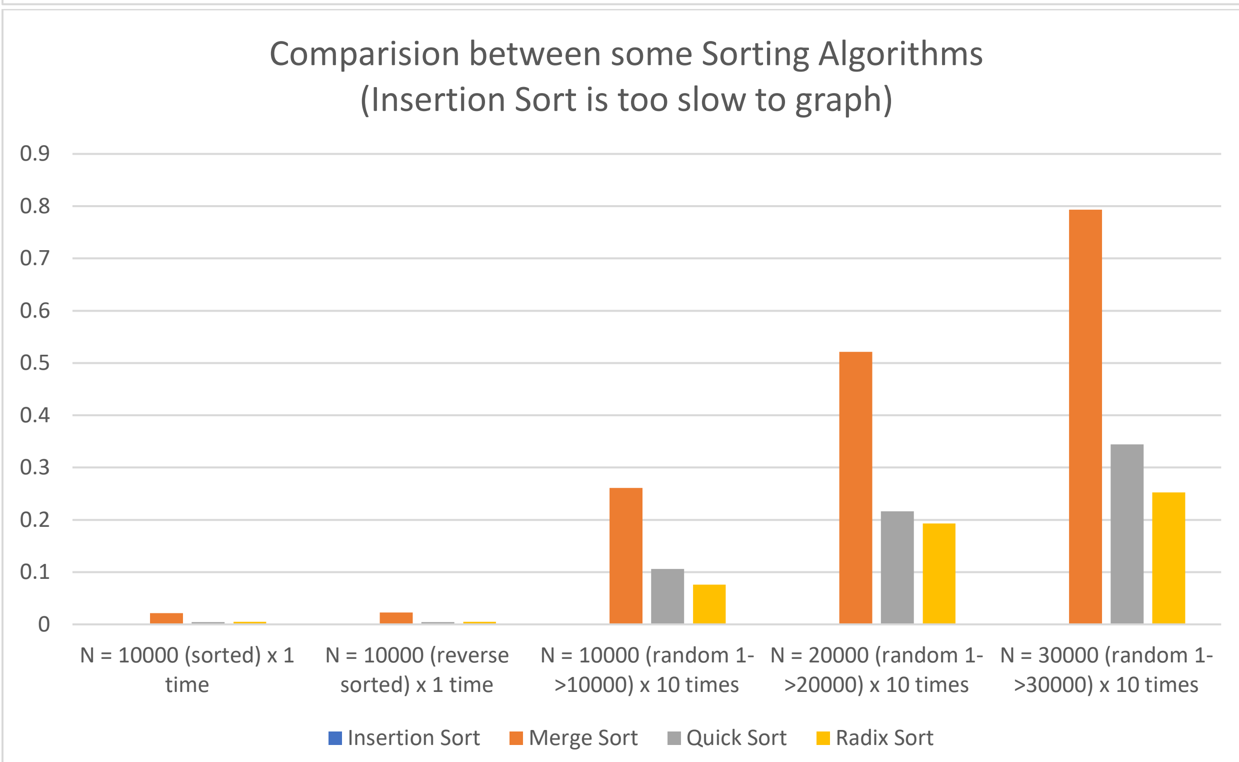

N = 10000 (reverse sorted) x 1 time
0.025
0.02
0.015
0.01
0.005
0
Insertion Sort
Merge Sort
Quick Sort
Radix Sort


N = 10000 (random 1->10000) x 10 times
0.3
0.25
0.2
0.15
0.1
0.05
0
Insertion Sort
Merge Sort
Quick Sort
Radix Sort


N = 20000 (random 1->20000) x 10 times
0.6
0.5
0.4
0.3
0.2
0.1
0
Insertion Sort
Merge Sort
Quick Sort
Radix Sort


N = 30000 (random 1->30000) x 10 times
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
Insertion Sort
Merge Sort
Quick Sort
Radix Sort


N = 5000000 (random 1->5000000) x 10 times
160
140
120
100
80
60
40
20
0
Insertion Sort
Merge Sort
Quick Sort
Radix Sort


Comparision between some Sorting Algorithms
(Insertion Sort is too slow to graph)
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
N = 10000 (sorted) x 1 time
N = 10000 (reverse sorted) x 1 time
N = 10000 (random 1->10000) x 10 times
N = 20000 (random 1->20000) x 10 times
N = 30000 (random 1->30000) x 10 times
Insertion Sort
Merge Sort
Quick Sort
Radix Sort

Comments:

- With small test (N = 10000), Insertion Sort runs faster than others, but when we increase the size of array (>= 10000), it runs too slowly.
- The running time of Quick Sort and Merge Sort seems to be equal, because both takes (average) O(nlogn).
- In some random case, Quick Sort runs better than Merge Sort. I think it depends on test case (we used random array to calculate the running time).
- In most of cases, the Radix Sort runs fastest, because it works on Linear Time.

In conclusion:

- If the size of array is small (less than 10^4), we can use Insertion Sort for simply coding and implementation that without affecting the running time.
- **If the limitation of each element in an array is less than 10^7**, we can use Radix Sort or Counting Sort to sort an array **in Linear Time** (without comparision).
- In the other cases, we know that Heap Sort is the most stable algorithms to work in **O(nlogn)** time. But Quick Sort is also the good choice to sort array with size **not bigger than 10^6 elements**. *More than that, we can improve Quick Sort by select pivot randomly (in the test, we chose the mid-element to be a pivot) to avoid the bad partitions..* Merge Sort is also good, but it seems **to be difficult to implement right.**