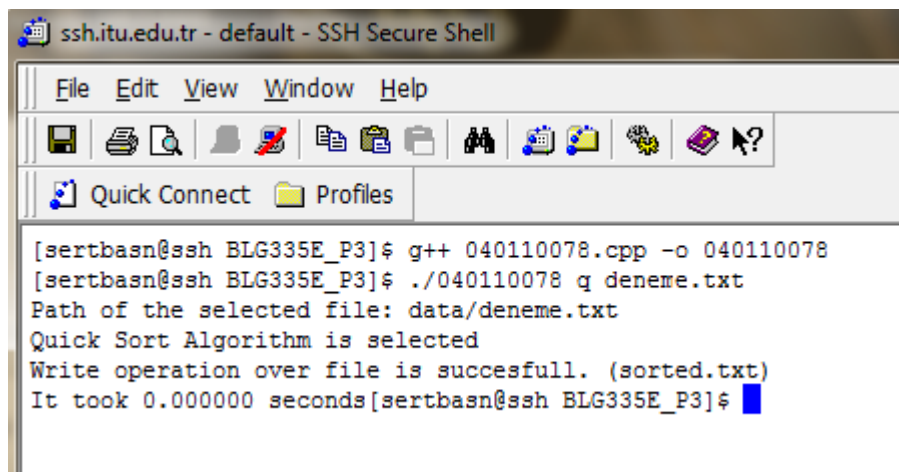


Part A. Implementation

In this project we implement three different sorting algorithm and compare their performances for different cases.

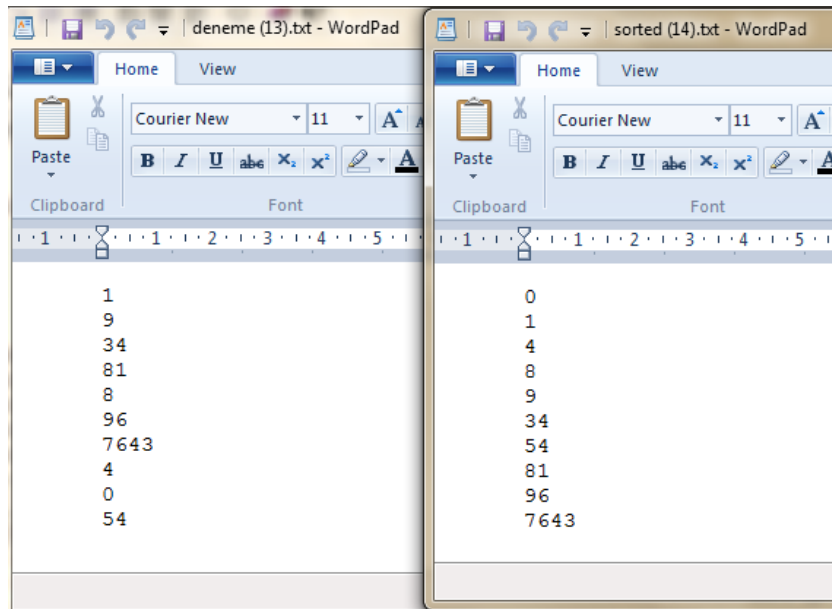
Example compile results are given below. Note that our original (unsorted) data is founded in a directory such as "data/".

Also all code is compiled and run over SSH platform.



```
ssh.itu.edu.tr - default - SSH Secure Shell
File Edit View Window Help
[sertbasn@ssh BLG335E_P3]$ g++ 040110078.cpp -o 040110078
[sertbasn@ssh BLG335E_P3]$ ./040110078 q deneme.txt
Path of the selected file: data/deneme.txt
Quick Sort Algorithm is selected
Write operation over file is succesfull. (sorted.txt)
It took 0.000000 seconds[sertbasn@ssh BLG335E_P3]$
```

Example of implementation:



Note: Above screen shot is from an example array. It is just builded to show how the program works.

Part B. Report

1. Running Time of the Algorithms

	Quick Sort	Counting Sort	Radix Sort
1k-10k.txt	0.000000	0.000000	0.000000
1k-10M.txt	0.000000	*	0.000000
100k-10k.txt	0.020000	0.000000	0.030000
100k-10M.txt	0.020000	*	0.060000

Note1: At the lines which are represented with " *" I got an error. I think that is related with max number in the array because it just gives an error if our range ends at 10M. Unfortunately, I do not have enough time to fix it. But except these two all algorithms operate correctly for all data.

Note2: Most of the lines are 0.000000. The algorithms are very fast especially for small numbers. They can differ for large numbers as can be seen in above table.

2. Detail in Quick Sort

a. Complexity of selecting the first element/ last element as a pivot can vary for different inputs. Another approach is selecting the element at the middle as a pivot. It is better on average case. Also, if we choose random pivot it decreases the probability of encountering with the worst case. So it gives better results but in this case our cost increases.

b. In quick sort the worst case occurs when partition is unbalanced such as 2 sub array with the length of $n-1$ and 0. In this case complexity becomes $O(n^2)$ like in Insertion Sort. In our algorithm, choosing the last element in the partition as the pivot in results $O(n^2)$. Already sorted array is the worst case for quick sort.

3. Complexity of Counting Sort

We need to specify k which is related with the max element of the array. We implement counting algorithm in the assumption of all elements in the array are in the range of 1 to k . Because its' basic idea is determining the number of elements smaller than the related one. So, if there exists 10 element smaller than mine, my element should be placed at index 11. In a case of $k = O(n)$ the algorithm runs with $O(n)$. Also, when we creating 'count' array we should allocate place for t related with k .

4. Worst Case for Radix Algorithm

The idea is that underlying radix sort is sort integers by first sorting them by their least significant digit and continue to the procedure until the most significant digit.

In counting sort k is restricted to the range of elements, but we need to change it as range of digit and call it for each digit. So we change a little bit a counting sort algorithm which is also given as a pseudo code.

If there are n integers and the maximum length of the integers is d , then radix sort will call counting sort with a complexity of $O(n + k)$ for each of the d digits. So, result complexity of a radix sort is $O(d(n + k))$. In our algorithm $k=10$ which is related with all integers from 0 to 9. So the complexity of counting sort becomes $O(n)$. Also we need d digits which is $d = \log n$.

So, $O(d O(n)) = O(n \log n)$ becomes as the total running time.