BBM 204: Software Laboratory II

ASSIGNMENT 1:

Analyzing the complexity of sorting algorithms

Name: Mehmet Emin Tuncer

Idendity Number: 21591022

TAs: Selim Yilmaz, Levent Karacan, Merve Ozdes

Due Date: 15.03.2018 23:59:59

PROBLEM and AIM:

Throughout the history of programming, coders compare different problems with solutions for various problems: time and memory problems, etc. They have developed various algorithms to get rid of these problems. The sorting algorithm is also one of them. In sorting algorithms; it was aimed to save as much time and memory as possible during the sorting process. While time-memory is not very important for lists with a small number of elements, these algorithms are very convenient for lists with very large numbers.

In table you are provided comparative complexity/runtime comparison of well-known sorting algorithms.

Table 1: Complexity comparison of sorting algorithms

	Best	Average	Worst
Selection Sort	$\Omega(n^2)$	$\theta(n^2)$	$O(n^2)$
Bubble Sort	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$
Insertion Sort	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$
Heap Sort	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n \log(n))$
Quick Sort	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n^2)$
Merge Sort	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n \log(n))$
Radix Sort	$\Omega(nk)$	$\theta(nk)$	$O(nk)$

In this homework, the files containing the data about the traffic flow were given. The files were kept in different numbers. We sorted these data files according to the sorting algorithms we selected; we wanted to specify how long it took to sort and compare the algorithms by analyzing them.

SOLUTION SUMMARY AND SELECTED ALGORITHMS

The sorting algorithms I chose are: “Selection Sort” , “Quick Sort” and “Insertion Sort”.

For the problem solving, 6 classes were defined in total except main class. Each class has been defined for sorting algorithms and file operations. Finally, a class named “Item” is defined to facilitate file reading operations.

1-GENERATED CLASSES

1.1-Item Class:

There are two pieces of data held by this class that are defined to facilitate sorting and reading when the input file is read and dropped into a list.

```
public class Item {
    public double value;
    public String word;

    public Item(double value,String word){
        this.value=value;
        this.word=word;
    }
```

Lines in the input file given the "word" value defined in the object; "Value" is for assigning the data to be selected and sorted from that row.

Sorting and replacing is performed by comparing "value" data in the object. If it is desired to print the sorted list, the object's "word" data is printed to the csv file and the input file is updated.

1.2-FileRead Class:

- **getLines():**The file path is given as an argument. Reads the given input file line by line and assigns it to a string list. Returns the String list.
- **setItems():**An empty Item list, the String list returned by getItems (), and the index of the data to be sorted are the arguments. Assign the lines in the String list to the "word" data of the Item object. It divides the rows by "," and assigns the indexed value to the "value" data of the Item object. Finally, it adds the Item objects to the empty Item list.

1.3-WritetoFile Class:

- **getFinalList():**It creates a sorted String list of the lines in the input file using the sorted item list and the first element of the String list returned by getLines ().
- **writeCsv():**Called if the user receives an update command. The input file is updated in sorted order.

1.4-Sort Classes:

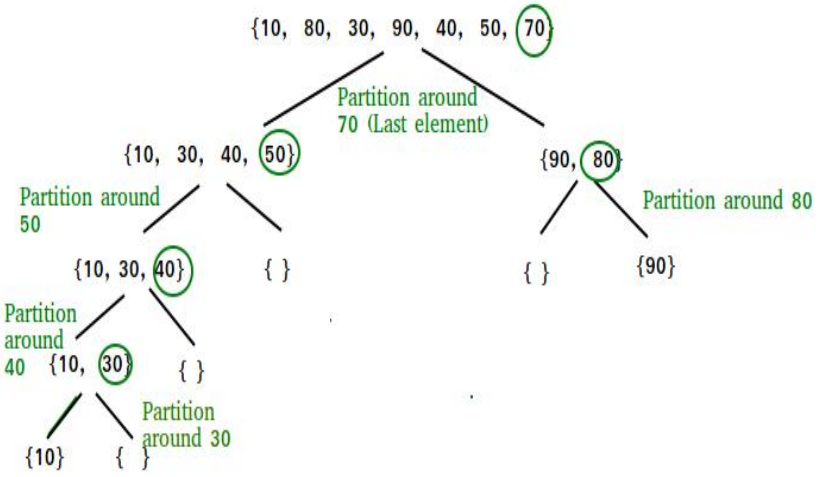
Each sorting class compares the Item list with its own algorithm by comparing the "value" data of the Item objects in the Item list. These three algorithms will be explained in detail

2-SELECTED ALGORITHMS AND EXPLANATIONS

2.1-Quick Sort Algorithm

Quicksort is a “Divide and Conquer algorithm”. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays.

- Pick an element, called a *pivot*, from the array.
- *Partitioning*: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the *partition* operation.
- Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.



The last element (70) is selected as the pivot member in the adjacent drawing. Values that are small in the pivot are left to the left of the pivot, to the right of large values. The process is also applied recursively to the lists that are split. This process continues until each split list is sorted among themselves.

In this assignment the middle element was chosen as the pivot.

This algorithm uses two main functions: partition and sort.

Partition algorithm:

Two indexes are defined from the beginning(i) and end(j) of the list, the middle element is selected as the pivot. The index is incremented as long as the values of the selected index (i) are smaller in the pivot value. Similarly, the values of the index (j) selected from the end are reduced if greater than the value of the pivot element. These index values are changed when the progress of the two indexes is finished.

My Pseudo code for partition() :

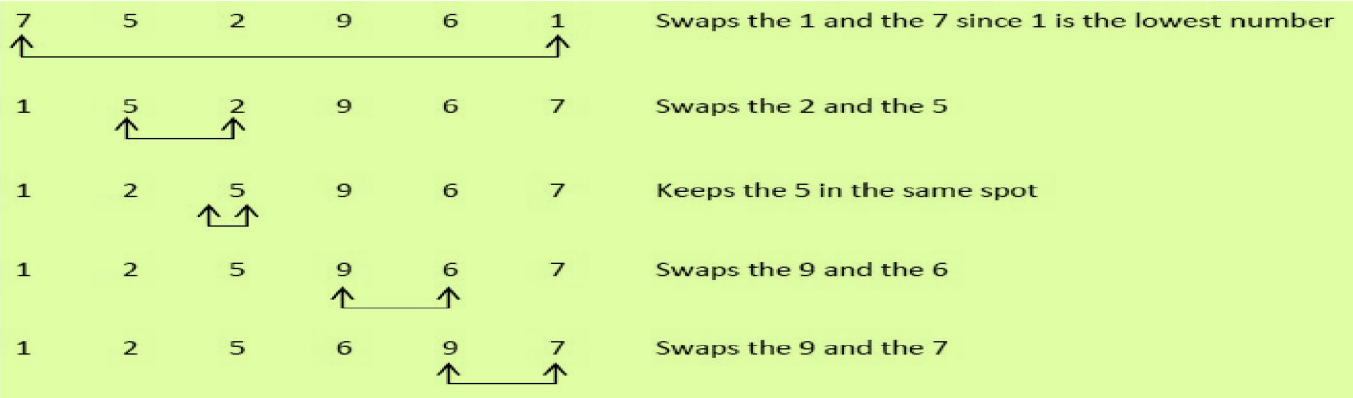
```
partition(itemlist, left, right) {
    i = left, j = right
    pivot = (left+right)/2
    while (i <= j) {
        while (itemlist(i).value < pivot)
            i++
        while (itemlist (j).value > pivot)
            j--
        if (i <= j) {
            swap itemlist(i) and itemlist(j)

            i++
            j--
        }
    }
    return i
}
```

Time Complexity: Best→O(n*logn) Worst → $O(n^2)$

2.2-Selection Sort Algorithm

The selection sort algorithm sorts an array by repeatedly finding the minimum or maximum element from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array: “sorted “ and “unsorted” parts.

In every iteration of selection sort, the minimum or maximum element from the unsorted subarray is picked and moved to the sorted subarray.



My Pseudo code for SelectionSort() :

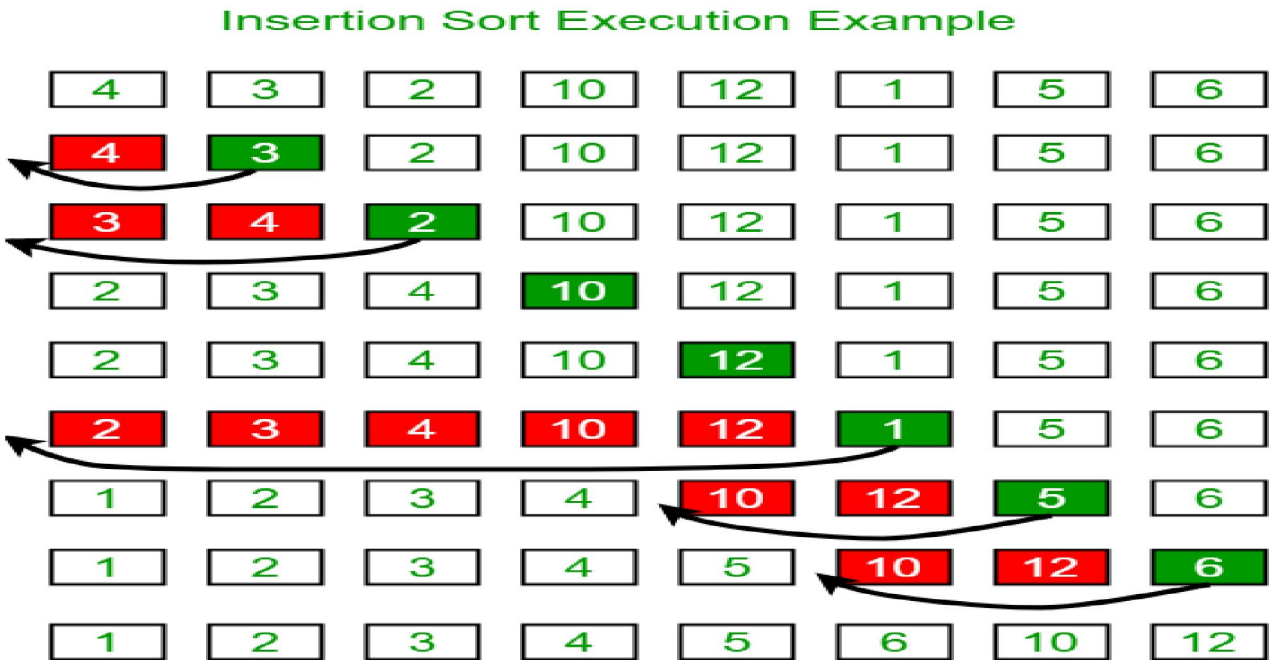
```
selectionSort(itemlist){
    i, j, minIndex;
    n =itemlist.size();
    for (i = 0; i < n - 1; i++) {
        minIndex = i;
        for (j = i + 1; j < n; j++)
            if (itemlist(j).value <itemlist(minIndex).value)
                minIndex = j;

        if (minIndex != i) {
                swap itemlist(i) and itemlist(minindex)

        }
    }
}
```

Time Complexity: $O(n^2)$ as there are two nested loops.

2.3-Insertion Sort Algorithm

The insertion sequence is repeated and an input element is consumed each time it is repeated and the list of ordered output is increased. At each iteration, the insert sort removes an item from the input, finds the position of the sorted list, and adds it to it. Repeats until no input element is left.

Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it. At each array-position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked). If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position.



My Pseudo code for InseritonSort() :

```
insertionSort(itemlist) {
    i, j
    Item newValue
    for (i = 1; i < itemlist.size(); i++) {
        newValue =itemlist(i)
        j = i;
        while (j > 0 && itemlist(j-1).value  > newValue.value) {
            swap itemlist(j) and itemlist(j-1)
            j--
        }
        swap itemlist(j) and newVaule
    }
}
```

Time Complexity: $O(n^2)$.

3-COMPARİSON OF ALGORITHMS AND GRAPH OF RESULTS

The data in the given input file are also listed separately for the three algorithms and the measured time values are specified in the table below.

Algorithm & Data Set	TrafficFlow100	TrafficFlow1000	TrafficFlow50000	TrafficFlow100000	TrafficFlowAll
Quick sort	0.00338235 seconds	0.006733922 seconds	0.030520985 seconds	0.068640377 seconds	0.103675058 seconds
Selection sort	0.001263835 seconds	0.035667766 seconds	17.086366724 seconds	67.847127283 seconds	340.973039586 seconds
Insertion sort	0.01066959 seconds	0.16825007 Seconds	17.37456823 Seconds	103.88118645 Seconds	1456.5427138 seconds

Table2

When the values in the table are examined, approximately “O (n * logn)” for Quick sort; For Insertion and Selection it is seen that it is close to “$O(n^2)$” measurements.