Idris Rotlomwala
C31-2003145

Experiment NO: 1B

Aim: Implementation of Insertion sort.

Theory: Working of insertion sort is similar to sorting of playing cards in hand.

Its assumed that first card is already sorted and then we select an unsorted card. If the selected card is greater than the first card then it is placed at the right side, otherwise it could be placed on the left side. Similarly all the unsorted cards are sorted, and put in their exact place.

The same approach is applied in the ~~selection~~ insertion sort. The idea behind this insertion sort is that we first take ~~the~~ one element, ~~it~~ iterate it through the sorted array. Although its easy to use its not appropriate for large data sets, as the time complexity of insertion sort in average case ~~is~~ and worst case is $O(n^2)$.

Insertion sort is less efficient than the other sorting algorithms like heap sort, quick sort, merge sort, etc.

Algorithm:

	cost	time
for $j \leftarrow 2$ to n: do	c_1	n
Key $\leftarrow A[j]$	c_2	$n-1$
$i \leftarrow j - I$	c_3	$n-1$

while (i>0 && A[i] > key)	c_4	$\sum_{j=2}^{n} t_j$
A [i+1] ← A [i]	c_5	$\sum_{2}^{n} (t_j - 1)$
i ← i - 1	c_6	"
end	-	-
A [i+1] ← key	b_7	$n - 1$
end	-	-

Size of input array is n. Total time taken by algorithm will be,

$$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \left(\sum_{2}^{n} t_j\right) + c_5 \sum_{2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 (n-1)$$

- Best case analysis:

ie. data is already sorted.

for this case, condition in while loop never gets executed & hence $t_j = 1$

$$\therefore T(n) = c_1 n + c_2 (n-1) + c_3 (n-1) + c_4 \sum_{2}^{n} 1 + c_5 \sum_{2}^{n} 0 + c_6 \cdot \sum_{2}^{n} 0 + c_7 (n-1)$$

where $\sum_{j=2}^{n} 1 = 1 + 1 + 1 \ldots (n-1)$ times $= n - 1$

$$\therefore T(n) = c_1 n + c_2 n - c_2 + c_3 n - c_3 + c_4 n - c_4 + c_7 n - c_7$$

$$= (c_1 + c_2 + c_3 + c_7) n - (c_2 + c_4 + c_4 + c_7)$$

$$= an + b$$

$$= O(n)$$

Worst case analysis :

ie. data is arranged in reversed order

so we have to compare A[j] with each elements of sorted array A[1 .... j-1]

so $t_j = j$

$$\& \sum_{j=2}^{n} t_j = 2 + 3 + 4 \cdots n = \Sigma n - 1$$

$$= \frac{n(n+1)}{2} - 1$$

$$4 \sum_{j=2}^{n} (j-1) = 1 + 2 + 3 \cdots (n-1) = \frac{n(n-1)}{2}$$

$$\therefore T(n) = c_1 n + c_2 (n-1) + c_3 (n-1) + c_4 \left(\sum_{2}^{n} j\right) + c_5 \sum_{2}^{n} (j-1) + c_6 \cdot \sum_{2}^{n} (j-1) + c_7 (n-1)$$

$$= c_1 n + c_2 (n-1) + c_3 (n-1) + c_4 \left(\frac{n(n+1)}{2} - 1\right) + c_5 \left(\frac{n(n-1)}{2}\right) + c_6 \frac{n(n-1)}{2} + c_7 (n-1)$$

$$= \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right) n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4}{2} + \frac{c_5}{2} - \frac{c_6}{2} + c_7\right) n - (c_2 + c_3 + c_4 + c_7)$$

$$= an^2 + bn + c = O(n^2)$$

Average case analysis : Average case is often roughly as bad as worst case. Here, half of the elements are greater than A[j] and rest are less.

So $t_j = j/2$

which again turns out to be quadratic ie $O(n^2)$

Example :

We here on array; 5, 3, 4, 2 to be sorted in ascending order.

1. We start with assumption 1st element is already sorted.

5 | 3, 4 2
Sorted unsorted

2. Nowe we compare 3 with 5, ∵ 3 < 5
∴ we swap them

3, 5 | 4, 2

3. Comparing 4 with 5, 4 < 5 ;
comparing 4 with 3, 4 > 3
∴ swap 5 & 4

3, 4, 5 | 2

4. Comparing 2 with 5, ∵ 2 < 5 ∴ swap 2 & 5
2 with 4, ∵ 2 < 4 ∴ swap 2 & 4
2 with 3, ∵ 2 < 3
∴ swap 3 & 2.

2, 3, 4, 5 |

Which is the sorted array.

Applications :

1) Insertion sort is used when the number of elements in the dataset is less.
2) It is also used where the array is almost sorted.

Program :

```
import java.util.Scanner;

public class InsertionSort{

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter the number of elements : ");
        int n = sc.nextInt();
        int[] arr = new int[n];

        System.out.print("Enter the Elements : ");
        for(int i=0;i<n;i++) arr[i]= sc.nextInt();

        int swaps = IntsertnSort(arr);

        System.out.println("\nSorted array :");
        for (int k=0;k<n;k++) {
            System.out.print("\t"+arr[k]);
        }
        System.out.println("\n\nNumber of swaps : "+ swaps);
        sc.close();
    }


    private static int IntsertnSort(int[] arr) {

        int swaps=0;
```

```
        System.out.println("\n***Insertion Sort*** \nPasses :");
        for(int i=1;i<arr.length;i++){
            print(arr,i);
            int temp = arr[i];
            int j;
            for(j=i-1;j>=0;j--){
                if(arr[j]>temp){
                    arr[j+1] = arr[j];
                    swaps++;
                }
                else break;
            }
            arr[j+1]=temp;
        }
        return swaps;
    }



    private static void print(int[] arr, int i) {
        for (int k=0;k<arr.length;k++) {
            System.out.print("\t"+arr[k]);
            if(i-1 == k) System.out.print("   |");
        }
        System.out.println();
    }
}
```

Output :

```
Enter the number of elements : 7
Enter the Elements : 7 5 2 4 1 3 8

***Insertion Sort***
Passes :
        7   |   5       2       4       1       3       8
        5       7   |   2       4       1       3       8
        2       5       7   |   4       1       3       8
        2       4       5       7   |   1       3       8
        1       2       4       5       7   |   3       8
        1       2       3       4       5       7   |   8

Sorted array :
        1       2       3       4       5       7       8

Number of swaps : 12
PS C:\Users\IsmailRatlamwala\Documents\College prog\AOA>
```

Conclusion : Insertion sort works best with small number of elements. The worst case runtime complexity of insertion sort is $O(n^2)$ similar to bubble sort. However Insertion sort is considered better than bubble sort.