**Batch: B3 Roll No.: 121**

**Experiment No.\_\_\_**1**\_\_\_**

**Grade: AA / AB / BB / BC / CC / CD /DD**

**Signature of the Staff In-charge with date**

|  |
| --- |
| **Title: Implementation of selection sort/ Insertion sort** |

**Objective:** To analyse performance of sorting methods

**CO to be achieved:**

|  |  |
| --- | --- |
| CO 1 | Analyze the asymptotic running time and space complexity of algorithms. |

**Books/ Journals/ Websites referred:**

1. **Ellis horowitz, Sarataj Sahni, S.Rajsekaran,” Fundamentals of computer algorithm”, University Press**
2. **T.H.Cormen ,C.E.Leiserson,R.L.Rivest and C.Stein,” Introduction to algortihtms”,2nd Edition ,MIT press/McGraw Hill,2001**
3. [**http://en.wikipedia.org/wiki/Insertion\_sort**](http://en.wikipedia.org/wiki/Insertion_sort)
4. [**http://www.sorting-algorithms.com/insertion-sort**](http://www.sorting-algorithms.com/insertion-sort)
5. [**http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Insertion\_sort.html**](http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Insertion_sort.html)
6. [**http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/insertionSort.htm**](http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/insertionSort.htm)
7. [**http://en.wikipedia.org/wiki/Selection\_sort**](http://en.wikipedia.org/wiki/Selection_sort)
8. [**http://www.sorting-algorithms.com/selection-sort**](http://www.sorting-algorithms.com/selection-sort)
9. [**http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/selectionSort.htm**](http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/selectionSort.htm)
10. **http://courses.cs.vt.edu/~csonline/Algorithms/Lessons/SelectionCardSort/selectioncardsort.html**
11. **https://www.javatpoint.com/daa-insertion-sort#:~:text=Therefore%2C%20the%20insertion%20sort%20algorithm,key%20variable%20to%20perform%20swaps.**
12. **https://www.programiz.com/dsa/selection-sort**

**Pre Lab/ Prior Concepts:**

Data structures, sorting techniques.

**Historical Profile:**

There are various methods to sort the given list. As the size of input changes, the performance of these strategies tends to differ from each other. In such case, the priori analysis can helps the engineer to choose the best algorithm.

**New Concepts to be learned:**

Space complexity, time complexity, size of input, order of growth.

**Topic: Sorting Algorithms**

**Theory:** Given a function to compute on n inputs the divide-and-conquer strategy suggests splitting the inputs into k distinct subsets, 1< k ≤n, yielding k sub problems. These sub problems must be solved and then a method must be found to combine sub solutions into a solution of the whole. If the sub problems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied. Often the sub problems resulting from a divide-and-conquer design are the same type as the original problem. For those cases the reapplication of the divide-and- conquer principle is naturally expressed by a recursive algorithm. Now smaller and smaller sub problems of the same kind are generated until eventually sub problems that are small enough to be solved without splitting are produced.

**Algorithm Insertion Sort**

INSERTION\_SORT (*A,n*)

//The algorithm takes as parameters an array *A*[1.. *n*] and the length *n* of the array.

//The array *A* is sorted in place: the numbers are rearranged within the array

// A[1..n] of eletype, n: integer

**FOR** j ← 2 **TO** length[*A*]   
             **DO**  key ← *A*[*j*]      
                   {Put *A*[*j*] into the sorted sequence *A*[1 . . *j* − 1]}     
                    *i* ← *j* − 1      
                    **WHILE** *i* > 0 and *A*[*i*] > key  
                                 **DO** *A*[*i* +1] ← *A*[*i*]              
                                         *i* ← *i* − 1       
                     *A*[*i* + 1] ← key

**Algorithm Selection Sort**

SELECTION\_SORT (A,n)

//The algorithm takes as parameters an array *A*[1.. *n*] and the length *n* of the array.

//The array *A* is sorted in place: the numbers are rearranged within the array

// A[1..n] of eletype, n: integer

**FOR** *i* ← 1 **TO** *n*-1 **DO**    
    min *j* ← *i*;  
    min *x* ← A[*i*]  
   **FOR** *j* ← *i* + 1 to n do  
        **IF** A[*j*] < min x then  
            min *j* ← *j*  
            min *x* ← A[j]  
    A[min *j*] ← A [*i*]  
    A[*i*] ← min *x*

**The space complexity of Insertion sort:**

1. Insertion sort is an in-place algorithm
2. It performs all computations in the original array and no other array is used.
3. The insertion sort encompasses a space complexity of O(1) due to the usage of an extra variable “key”.

**The space complexity of Selection sort:**

1. Selection sort is an in-place algorithm
2. It performs all computations in the original array and no other array is used.
3. Space complexity is O(1) because an extra variable “temp” is used.

**Time complexity for Insertion sort:**

Input: Given n input elements.

Output: Number of steps incurred to sort a list.

Logic: If we are given n elements, then in the first pass, it will make n-1 comparisons; in the second pass, it will do n-2; in the third pass it will do n-3 and so on. Thus, the total number of comparisons can be found by:

i.e., O(n2)

Best case complexity: The insertion sort algorithm has a best-case time complexity of O(n) for the already sorted array because here, only the outer loop is running n times, and the inner loop is kept still.

Average case complexity: The average-case time complexity for the insertion sort algorithm is O(n2), which is incurred when the existing elements are jumbled in order, i.e., neither in ascending order nor in descending order.

Worst case complexity: The worst-case time complexity is also O(n2), which occurs when we sort the ascending order of the array into descending order. In this case, every individual element is compared with the rest of the elements, due to which (n-1) comparisons are made for every nth element.

The insertion sort algorithm is highly recommended, especially when a few elements are left for sorting or in case the array encompasses few elements.

**Time complexity for selection sort:**

In the first cycle, (n-1) comparisons are made; (n-2) in the second; (n-3) in the third and so on until one comparison is made in the last cycle.

i.e., O(n2).

Best Case Complexity: O(n2)

It occurs when the array is already sorted.

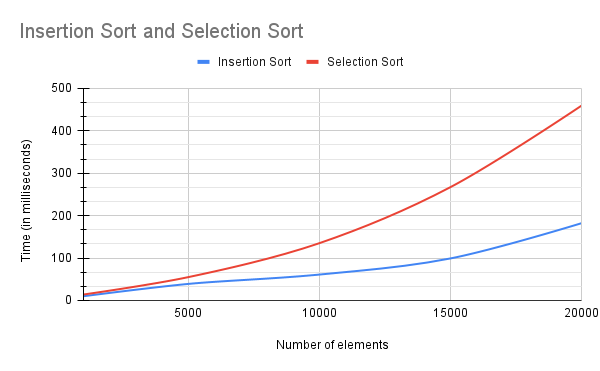
Average Case Complexity: O(n2)

It occurs when the array elements are in a jumbled order i.e., neither ascending nor descending.

Worst Case Complexity: O(n2)

It occurs when an array, which is in ascending order, is sorted in descending order.

**Graphs for varying input sizes: (Insertion Sort & Selection sort)**



**CONCLUSION:**

Thus, in this experiment, the space and time complexity analysis of selection sort and insertion sort has been performed. It is found that insertion sort is faster than selection sort. This means that the time complexity of insertion sort is superior to the time complexity of selection sort. This is because for Insertion sort, the best, average and worst case time complexities are O(n), O(n2) and O(n2) respectively, while for Selection sort, the best, average and worst case time complexities are O(n2), O(n2) and O(n2) respectively. Simultaneously, the Space Complexity for both Insertion sort and Selection sort is the same, i.e., O(1) as both are in-place sorting algorithms. Therefore, it can be concluded that of the two, Insertion sort is the better choice because it is of the same size as Selection sort, but faster.