|  |  |
| --- | --- |
| **Course Code:** CT2352 | **Course Name:** Lab - Design & Analysis of Algorithms |

|  |  |
| --- | --- |
| **Name of the Student:**  S Akshansh | **Sem/Sec:**  6th/A |
| **Branch:**  Computer Technology | **Roll no.:** 72  **Enrolment no.:** 19010927 |

**PRACTICAL NO. 01**

**Aim: -**

Simulate both the insertion sort and selection sort algorithms on the following three arrays.

U = [1, 2, 3, 4, 5, 6], V = [6, 5, 4, 3, 2, 1] and W = [1, 3, 4, 5, 2, 6]

Does insertion sorting run faster on the array U or the array V? Justify your answer.

Does selection sorting run faster on the array U or the array V? Justify your answer.

Does insertion/selection sorting run faster on the array W? Justify your answer.

Derive the time complexity of Insertion Sort and Selection Sort algorithms.

**Theory: -**

**INSERTION SORT**

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.  
**Algorithm**   
To sort an array of size n in ascending order:   
1. Iterate from arr[1] to arr[n] over the array.   
2. Compare the current element (key) to its predecessor.   
3. If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

**SELECTION SORT**

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

**Algorithm**   
1) The subarray which is already sorted.   
2) Remaining subarray which is unsorted.  
In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

**Program: -**

**INSERTION SORT**

#include<iostream>

using namespace std;

int main(){

    int n;

    cout<<"Enter number of elements in array: ";

    cin>>n;

    int arr[n];

    int count1 = 0;

    int count2 = 0;

    for (int i = 0; i <n; i++)

    {

        cin>>arr[i];

    }

    for (int i = 0; i < n; i++)

    {

        int current = arr[i];

        int j=i-1;

        count1++;

        while (arr[j]>current && j>=0)

        {

            arr[j+1]=arr[j];

            j--;

            count2++;

        }

        arr[j+1]=current;

    }

    for (int i = 0; i < n; i++)

    {

        cout<<arr[i]<<" ";

    }

    cout<<endl;

    cout<<count1<<" "<<count2<<endl;

    cout<<count1 + count2<<endl;

    return 0;

}

**SELECTION SORT**

#include<iostream>

using namespace std;

int main(){

    int n;

    cout<<"Enter number of elements in array: ";

    cin>>n;

    int arr[n];

    int count1 = 0;

    int count2 = 0;

    for (int i = 0; i <n; i++)

    {

        cin>>arr[i];

    }

    for (int i = 0; i < n-1; i++)

    {

       for (int j = i+1; j < n; j++)

       {

           count1++;

           if (arr[j]<arr[i])

           {

              int temp = arr[j];

              arr[j]=arr[i];

              arr[i]=temp;

              count2++;

           }

       }

    }

    for (int i = 0; i < n; i++)

    {

        cout<<arr[i]<<" ";

    }

    cout<<endl;

    cout<<count1<<" "<<count2<<endl;

    cout<<count1 + count2<<endl;

    return 0;

}

**Input/Output: -**

**INSERTION SORT**

**Text

Description automatically generated with medium confidence**

U = {1,2,3,4,5,6}

Text

Description automatically generated

V = {6,5,4,3,2,1}

**Text

Description automatically generated**

W = {1,3,4,5,2,6}

**SELECTION SORT**

Text

Description automatically generated

U = {1,2,3,4,5,6}

A picture containing text, meter, device, control panel

Description automatically generated

V = {6,5,4,3,2,1}

**A picture containing text, device, meter, close

Description automatically generated**

W = {1,3,4,5,2,6}

1. **Does insertion sort run faster on the array U or the array V?**

**U = {1,2,3,4,5,6} V = {6,5,4,3,2,1}**

**Justification** – As, Insertion Sort algorithm is adaptive means if array is already sorted then its time complexity is O(n) i.e., best case but if array is sorted in reverse order, then its time complexity is O(n^2) i.e., worst case. So, we can prove this in above insertion Sort Program code using C++ language.

1. **Does selection sort run faster on the array U or the array V?**

**U = {1,2,3,4,5,6} V = {6,5,4,3,2,1}**

**Justification** – As, Selection Sort algorithm is not-adaptive means its time complexity is O(n^2) in all cases whether the given array is sorted or unsorted it will check each condition every time. So, we can prove this in above Selection Sort Program code using C++ language.

1. **Does insertion/selection sorting run faster on the array W?**

**W = {1,3,4,5,2,6}**

**Justification** – Insertion Sort runs faster than Selection Sort for array W because some elements already at their correct position. But in Selection sort it checked every element irrespective of their position. So, we can prove this in above Program code using C++ language.

**4. Derive Time Complexity for Insertion Sort and Selection Sort algorithm**

**(i) Derivation of Time Complexity of Insertion Sort**

**• Running Time**

Running Time of an algorithm is execution time of each line of algorithm As stated, Running Time for any algorithm depends on the number of operations executed. We could see in the Pseudocode that there are precisely 7 operations under this algorithm. So, our task is to find the Cost or Time Complexity of each and trivially sum of these will be the Total Time Complexity of our Algorithm. We assume Cost of each i operation as C i where i ∈ {1,2,3,4,5,6,8} and compute the number of times these are executed. Therefore, the Total Cost for one such operation would be the product of Cost of one operation and the number of times it is executed.

Total Running Time of Insertion sort: -

(T(n)) = C1\*n + (C2 + C3)\*(n - 1) + C4\*Σ n - 1 j = 1(t j) + (C5 + C6)\*Σ n - 1 j = 1(t j) + C8\*(n - 1)

**Best Case Analysis –**

In Best Case i.e., when the array is already sorted, tj = 1 Therefore,

T(n) = C1\*n + (C2 + C3)\*(n - 1) + C4\*(n - 1) + (C5 + C6)\*(n - 2) + C8\*(n - 1) which when further simplified has dominating factor of n and gives

**T(n) = C\*( n ) or O(n)**

**Worst Case Analysis –**

In Worst Case i.e., when the array is reversly sorted (in descending order), tj = j Therefore, T(n) = C1\*n + (C2+C3)\*(n-1) + C4\*(n-1) (n)/2 + (C5+C6)\*( (n-1) (n)/2-1) + C8\*(n-1) which when further simplified has dominating factor of n 2 and gives

**T(n) = C\*(n2 ) or O(n2 )**

**Average Case Analysis –**

Let's assume that tj = (j-1)/2 to calculate the average case Therefore, T(n) = C1\*n + (C2 + C3)\*(n-1) + C4/2\*(n-1) (n)/2 + (C5+C6)/2\*( (n-1)(n)/2-1) + C8\*(n-1) which when further simplified has dominating factor of n 2 and gives

**T(n) = C\*(n2 ) or O(n2 )**

➢ Worst Case Time Complexity is: O(N2 )

➢ Average Case Time Complexity is: O(N2 )

➢ Best Case Time Complexity is: O(N)

**(ii)Derivation of Time Complexity of Selection Sort**

At the beginning, the size of sorted sub-array (say S1) is 0 and the size of unsorted sub-array (say S2) is N.

At each step, the size of sorted sub-array increases by 1 and size of unsorted sub-array decreases by

1. Hence, for a few steps are as follows:

• Step 1: S1: 0, S2: N

• Step 2: S1: 1, S2: N-1

• Step 3: S1: 2, S2: N-2 and so on till S1 = N.

Hence, there will be N+1 steps. Hence, S2 = N - S1 The Time Complexity of finding the smallest element in a list of M elements is O(M). This is constant for all worst case, average case and best case. The time required for finding the smallest element is the size of unsorted sub-array that is O(S2). The exact value of S2 is dependent of the step. For step I, S1 will be I-1 and S2 will be N-S1 = N-I+1. So, the time complexity for step I will be:

• O(N-I+1) for find the smallest element

• O(1) for swapping the smallest element to the front of unsorted subarray I will range from 1 to N+1. Hence, the sum of time complexity of all operations will be as follows:

Sum [O(N-I+1) + O(1) ] for I from 1 to N+1 = Sum [O(N-I+1)] + Sum[O(1)] ... Equation 1

Sum [ O(1) ] = 1 + 1 + ... + 1 [(N+1) times] = N+1 = O(N)

Sum [O(N-I+1)] = N + (N-1) + ... + 1 + 0 =

= 1 + 2 + ... + N = N \* (N+1) / 2

= (N^2 + N) / 2 = O(N^2) + O(N)

= O(N^2) [as N^2 is dominant term]

Therefore, from Equation 1,

we get: Sum [O(N-I+1)] + Sum[O(1)] = O(N^2) + O(N) = O(N^2)

**Worst Case Analysis** **–**

The worst case is the case when the array is already sorted (with one swap) but the smallest element is the last element. For example, if the sorted number as a1, a2, ..., aN, then: a2, a3, ..., aN, a1 will be the worst case for our particular implementation of Selection Sort.

The cost in this case is that at each step, a swap is done. This is because the smallest element will always be the last element and the swapped element which is kept at the end will be the second smallest element that is the smallest element of the new unsorted sub-array.

Hence, the worst case has:

• N \* (N+1) / 2 comparisons

• N swaps Hence, the time complexity is O(N^2)

**Best Case Analysis –**

The best case is the case when the array is already sorted. For example, if the sorted number as a1, a2, ..., aN, then: a1, a2, a3, ..., aN will be the best case for our particular implementation of Selection Sort. This is the best case as we can avoid the swap at each step but the time spend to find the smallest element is still O(N). Hence, the best case has:

• N \* (N+1) / 2 comparisons

• 0 swaps Note only the number of swaps has changed. Hence, the time complexity is O(N^2).

**Average Case Analysis –**

Based on the worst case and best case, we know that the number of comparisons will be the same for every case and hence, for average case as well, the number of comparisons will be constant. Number of comparisons = N \* (N+1) / 2 Therefore, the time complexity will be O(N^2). To find the number of swaps,

• There are N! different combination of N elements

• Only for one combination (sorted order) there is 0 swaps.

• In the worst case, a combination will have N swaps. There are several such combinations.

• Number of ways to select 2 elements to swap = nC2 = N \* (N-1) / 2

• From sorted array, this will result in O(N^2) combinations which need 1 swap. So,

0 swap = 1 combination

1 swap = O(N^2) combinations

2 swaps = O(N^4) combinations ...

N swaps = O(N) combinations

Hence, the average number of swaps will be N that is O((N+1)!) / O(N!).

Hence, the average case has:

• N \* (N+1) / 2 comparisons

• N swaps

➢ Worst Case Time Complexity is: O(N2 )

➢ Average Case Time Complexity is: O(N2 )

➢ Best Case Time Complexity is: O(N2 )

**Conclusion –**

**Hence, I executed all programs of different problem statements and matched their result successfully.**