**Name**: Ravi Jamanbhai Makwana

**Roll No**: 21BCP418

Div-6, G-12

**Assignment-1**

**Aim:**

To understand the concept and implementation of Insertion sort and Selection sort.

**Theory:**

**Insertion Sort:**

The idea behind the insertion sort is that first take one element, iterate it through the sorted array before it. For all elements before it that are greater than it is shifted one place to the right. This is done for all elements, until we find an element that is smaller than it. When this situation occurs, we stop there and insert our element there. Similarly, this is repeated for all other elements.

In general, the Insertion sort technique compares each element with all of its previous elements and sorts the element to place it in its proper position.

The Insertion sort technique is more feasible for a smaller set of data, and thus arrays with a small number of elements can be sorted using efficiently Insertion sort.

Insertion sort is especially useful in sorting linked list data structures. As we know, linked lists have pointers pointing to its next element (singly linked list) and previous element (double linked list). This makes it easier to keep track of the previous and next elements.

Thus, it is easier to use Insertion sort for sorting linked lists. However, sorting will take a lot of time if the data items are more.

**Selection Sort:**

The **selection sort algorithm** sorts an array by repeatedly finding the minimum element (considering ascending order) from the unsorted part and putting it at the beginning.

The algorithm maintains two subarrays in a given array.

* The subarray which already sorted.
* The remaining subarray was unsorted.

In every iteration of the selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the beginning of unsorted subarray.

After every iteration sorted subarray size increase by one and unsorted subarray size decrease by one.

**Algorithm:**

**Insertion Sort:**

Step 1: Iterate the array from left.

Step 2: Pick the next element (key).

Step 3: Compare that key with all elements that are to the left of it.

Step 4: Shift all the elements in the sorted sub-array that are greater than the key

Step 5: Insert the key.

Step 6: Repeat until the array is sorted.

insertionSort(array)

mark first element as sorted

for each unsorted element X

'extract' the element X

for j <- lastSortedIndex down to 0

if current element j > X

move sorted element to the right by 1

break loop and insert X here

end insertionSort

**Selection Sort:**

Step 1 − Set MIN to location 0

Step 2 − Search the minimum element in the list

Step 3 − Swap with value at location MIN

Step 4 − Increment MIN to point to next element

Step 5 − Repeat until list is sorted

selectionSort(array, size)

repeat (size - 1) times

set the first unsorted element as the minimum

for each of the unsorted elements

if element < currentMinimum

set element as new minimum

swap minimum with first unsorted position

end selectionSort

**Program:**

**Insertion Sort:**

#include <stdio.h>

int main(){

int arr[] = {3,2,5,4,1};

printf("Before sorting: ");

for (int i=0;i<5;i++){

printf("%d ",arr[i]);

}

for (int i=0;i<5;i++){

int item = arr[i];

int counter = i-1;

while (arr[counter] > item){

arr[counter+1] = arr[counter];

//printf("shifted %d to the right\n", arr[counter]);

counter--;

}

arr[counter+1] = item;

}

printf("\nAfter sorting: ");

for (int i=0;i<5;i++){

printf("%d ",arr[i]);

}

}

**Selection Sort:**

#include <stdio.h>

int main(){

int arr[] = {3,5,2,4,1};

printf("Before sorting: ");

for (int i=0;i<5;i++){

printf("%d ",arr[i]);

}

for (int i=0;i<5;i++){

int smallest = arr[i];

int index=i;

for (int j=i+1;j<5;j++){

if (arr[j]<smallest){

smallest = arr[j];

index = j;

}

}

//swap the next element (i'th) with the smallest element (found at index j)

//printf("swapped %d ",arr[index]);

//printf("with %d\n",arr[i]);

int temp = arr[index];

arr[index] = arr[i];

arr[i] = temp;

}

printf("\nAfter sorting: ");

for (int i=0;i<5;i++){

printf("%d ",arr[i]);

}

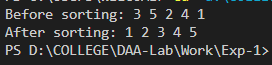
}

**Output:**

**Insertion Sort:**



**Selection Sort:**



**Analysis of Algorithm:**

**Insertion Sort:**

* The worst-case time complexity of Insertion sort is O(N^2).

In Worst Case i.e., when the array is reversely 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 n2 and gives T(n) = C \* ( n 2)

or O( n2 ).

* The average case time complexity of Insertion sort is O(N^2)

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 n2 and gives T(n) = C \* ( n 2)

or O( n2 ).

* The time complexity of the best case is O(N).

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).

* The space complexity is O(1).

**Selection Sort:**

|  |  |
| --- | --- |
| Cycle | Number of Comparison |
| 1st | (n-1) |
| 2nd | (n-2) |
| 3rd | (n-3) |
| ... | ... |
| last | 1 |

Number of comparisons: (n-1) + (n-2) + (n-3) + … + 1 = n(n-1) / 2 nearly equals to n2.

**Complexity** = O(n2)

Also, we can analyse the complexity by simply observing the number of loops. There are 2 loops so the complexity is n\*n = n2.

**Time Complexities:**

* **Worst Case Complexity:** O(n2)   
  If we want to sort in ascending order and the array is in descending order then, the worst case occurs.
* **Best Case Complexity:** O(n2)  
  It occurs when the array is already sorted
* **Average Case Complexity:** O(n2)  
  It occurs when the elements of the array are in jumbled order (neither ascending nor descending).

The time complexity of the selection sort is the same in all cases. At every step, you have to find the minimum element and put it in the right place. The minimum element is not known until the end of the array is not reached.