Unit2

Program for insertion sort

Aim:

- To implement insertion sort

Theory:

- 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.

Insertion Sort Logic

- Consider you have 10 cards out of a deck of cards in your hand. And they are sorted, or arranged in the ascending order of their numbers.

- If I give you another card, and ask you to insert the card in just the right position, so that the cards in your hand are still sorted. What will you do?

- you will have to go through each card from the starting or the back and find the right position for the new card, comparing it's value with each card.
- Once you find the right position, you will insert the card there.

- Similarly, if more new cards are provided to you, you can easily repeat the same process and insert the new cards and keep the cards sorted too.

- This is exactly how insertion sort works.
- It starts from the index 1(not 0), and each index starting from index 1 is like a new card, that you have to place at the right position in the sorted subarray on the left.

Following are some of the important characteristics of Insertion Sort:

- It is efficient for smaller data sets, but very inefficient for larger lists.
- Insertion Sort is adaptive, that means it reduces its total number of steps if a partially sorted array is provided as input, making it efficient.
- It is better than Selection Sort and Bubble Sort algorithms.

- Its space complexity is less. Like bubble Sort, insertion sort also requires a single additional memory space.
- It is a stable sorting technique, as it does not change the relative order of elements which are equal.

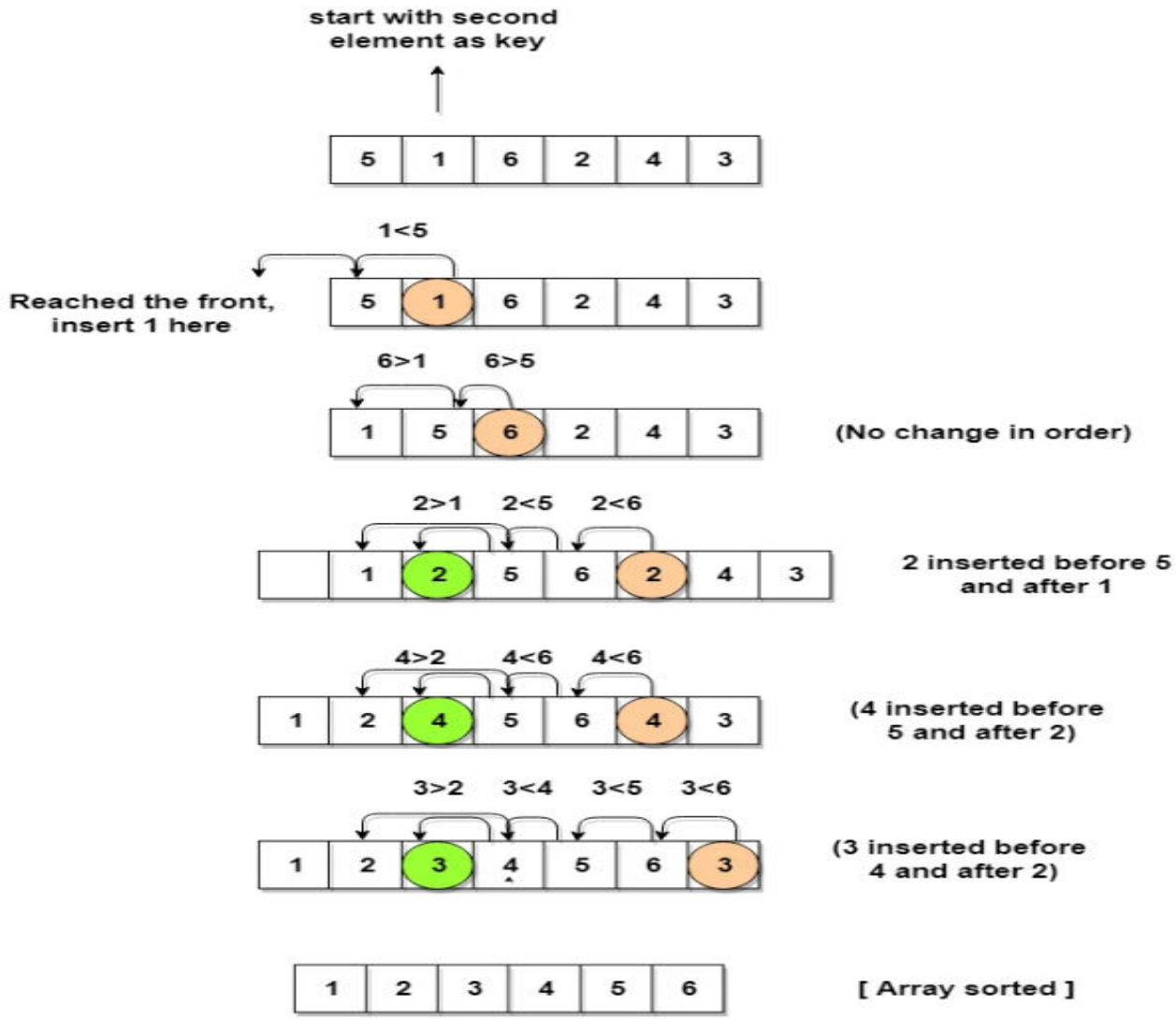
How Insertion Sort Works?

- Following are the steps involved in insertion sort:
- We start by making the second element of the given array, i.e. element at index 1, the key. The key element here is the new card that we need to add to our existing sorted set of cards(remember the example with cards above).
- We compare the key element with the element(s) before it, in this case, element at index 0:
- If the key element is less than the first element, we insert the key element before the first element.

- If the key element is greater than the first element, then we insert it after the first element.
- Then, we make the third element of the array as key and will compare it with elements to it's left and insert it at the right position.
- And we go on repeating this, until the array is sorted.

- Let's consider an array with values {5, 1, 6, 2, 4, 3}

Below, we have a pictorial representation



- As you can see in the diagram above, after picking a key, we start iterating over the elements to the left of the key.

- We continue to move towards left if the elements are greater than the key element and stop when we find the element which is less than the key element.

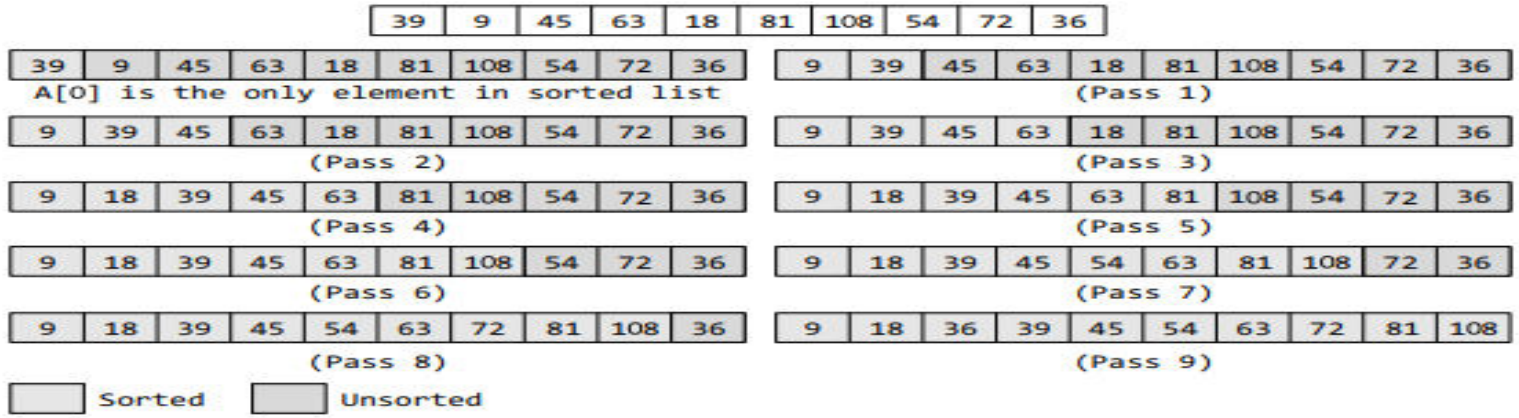
- And, insert the key element after the element which is less than the key element.

Characteristics of Insertion Sort:

- This algorithm is one of the simplest algorithm with simple implementation
- Basically, Insertion sort is efficient for small data values
- Insertion sort is adaptive in nature, i.e. it is appropriate for data sets which are already partially sorted.

Example 14.3 Consider an array of integers given below. We will sort the values in the array using insertion sort.

Solution



Initially, A[0] is the only element in the sorted set. In Pass 1, A[1] will be placed either before or after A[0], so that the array A is sorted. In Pass 2, A[2] will be placed either before A[0], in between A[0] and A[1], or after A[1]. In Pass 3, A[3] will be placed in its proper place. In Pass N–1, A[N–1] will be placed in its proper place to keep the array sorted.

To insert an element A[K] in a sorted list A[0], A[1], . . ., A[K–1], we need to compare A[K] with A[K–1], then with A[K–2], A[K–3], and so on until we meet an element A[J] such that A[J] <= A[K]. In order to insert A[K] in its correct position, we need to move elements A[K–1], A[K–2], . . ., A[J] by one position and then A[K] is inserted at the (J+1)th location. The algorithm for insertion sort is given in Fig. 14.7.

```
INSERTION-SORT (ARR, N)

Step 1: Repeat Steps 2 to 5 for K = 1 to N - 1
Step 2:      SET TEMP = ARR[K]
Step 3:      SET J = K - 1
Step 4:      Repeat while TEMP <= ARR[J]
                   SET ARR[J + 1] = ARR[J]
                   SET J = J - 1
             [END OF INNER LOOP]
Step 5:      SET ARR[J + 1] = TEMP
        [END OF LOOP]
Step 6: EXIT
```

Figure 14.7 Algorithm for insertion sort

In the algorithm, Step 1 executes a for loop which will be repeated for each element in the array. In Step 2, we store the value of the Kth element in TEMP. In Step 3, we set the Jth index in the array. In Step 4, a for loop is executed that will create space for the new element from the unsorted list to be stored in the list of sorted elements. Finally, in Step 5, the element is stored at the (J+1)th location.

Advantages of Insertion Sort

The advantages of this sorting algorithm are as follows:

- It is easy to implement and efficient to use on small sets of data.
- It can be efficiently implemented on data sets that are already substantially sorted.
- It performs better than algorithms like selection sort and bubble sort. Insertion sort algorithm is simpler than shell sort, with only a small trade-off in efficiency. It is over twice as fast as the bubble sort and almost 40 per cent faster than the selection sort.
- It requires less memory space (only $o(1)$ of additional memory space).
- It is said to be online, as it can sort a list as and when it receives new elements.

Complexity of Insertion Sort

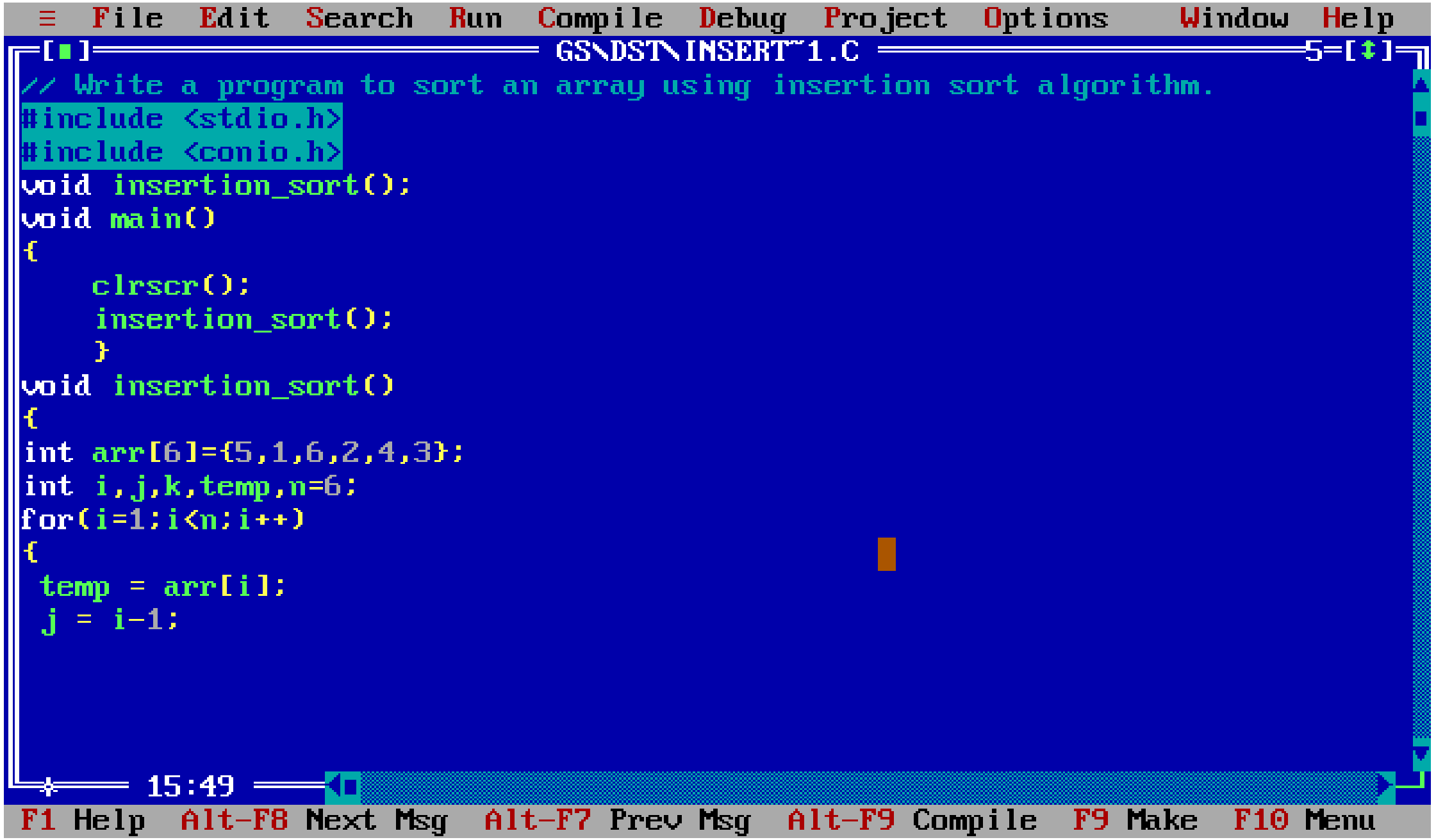
- For insertion sort, the best case occurs when the array is already sorted.
- In this case, the running time of the algorithm has a linear running time (i.e., O(n)).
- This is because, during each iteration, the first element from the unsorted set is compared only with the last element of the sorted set of the array.

- Similarly, the worst case of the insertion sort algorithm occurs when the array is sorted in the reverse order.
- In the worst case, the first element of the unsorted set has to be compared with almost every element in the sorted set.

- Furthermore, every iteration of the inner loop will have to shift the elements of the sorted set of the array before inserting the next element.
- Therefore, in the worst case, insertion sort has a quadratic running time (i.e., $O(n^2)$).
- Even in the average case, the insertion sort algorithm will have to make at least (K–1)/2 comparisons.
- Thus, the average case also has a quadratic running time.

- Worst Case Time Complexity [Big-O]: $O(n^2)$
- Best Case Time Complexity [Big-omega]: O(n)
- Average Time Complexity [Big-theta]: $O(n^2)$
- Space Complexity: O(1)

Source Code


File Edit Search Run Compile Debug Project Options Window Help
GS\DST\INSERT~1.C
// Write a program to sort an array using insertion sort algorithm.
#include <stdio.h>
#include <conio.h>
void insertion_sort();
void main()
{
clrscr();
insertion_sort();
}
void insertion_sort()
{
int arr[6]={5,1,6,2,4,3};
int i,j,k,temp,n=6;
for(i=1;i<n;i++)
{
temp = arr[i];
j = i-1;
15:49
F1 Help Alt-F8 Next Msg Alt-F7 Prev Msg Alt-F9 Compile F9 Make F10 Menu

```
 while((temp < arr[j]) && (j>=0))
 {
 arr[j+1] = arr[j];
 j--;
 }
 arr[j+1] = temp;
printf("Array after pass %d\n",i);
   for (k = 0; k < n; k++)
   printf("%d\n", arr[k]);
   getch();
}
printf("\n The sorted array is: \n");
for(i=0;i<n;i++)
printf(" %d\n", arr[i]);
}
```

38:2

Output

```
Array after pass 1
1
5
6
2
4
3
Array after pass 2
1
5
6
2
4
3
Array after pass 3
1
2
5
6
4
3
```

```
Array after pass 4
1
2
4
5
6
3
Array after pass 5
1
2
3
4
5
6

 The sorted array is:
 1
 2
 3
 4
 5
 6
_
```