**WEEK-1**

Design a Data structure for handling Student records- Design a solution, Implementation (Using Basic DS).

Here we are providing the customer a different recharge plans for airtel and Vodafone sims, and allowing the user to recharge their phone for their required plan. We are creating the plans of airtel and Vodafone as each modules.

1.1: Module of airtel recharge plans:

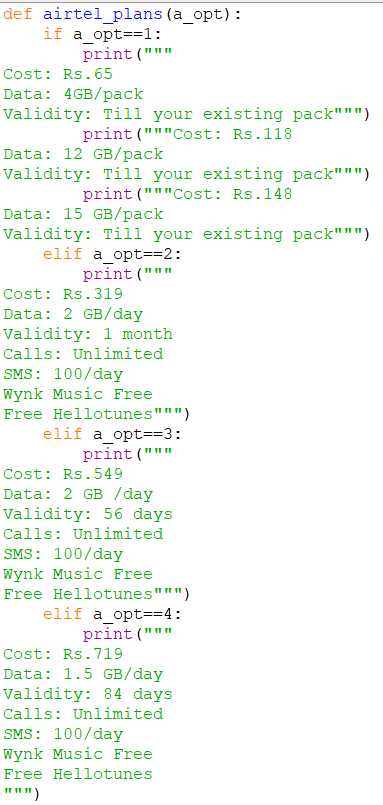


Fig 1.1.1: Screenshot of program of airtel\_plans.py

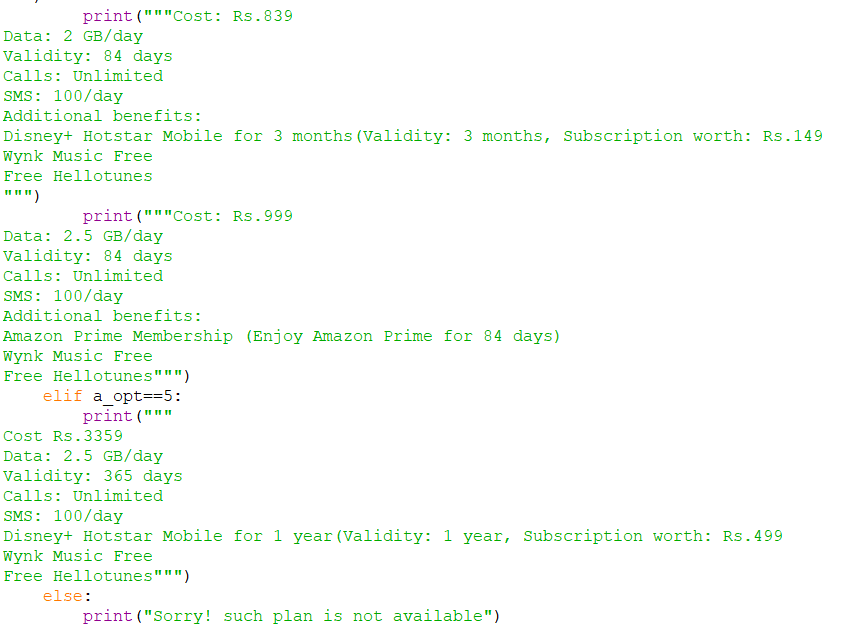


Fig 1.1.2: Screenshot of program airtel\_plans.py

1.2: Module of Vodafone recharge plans:

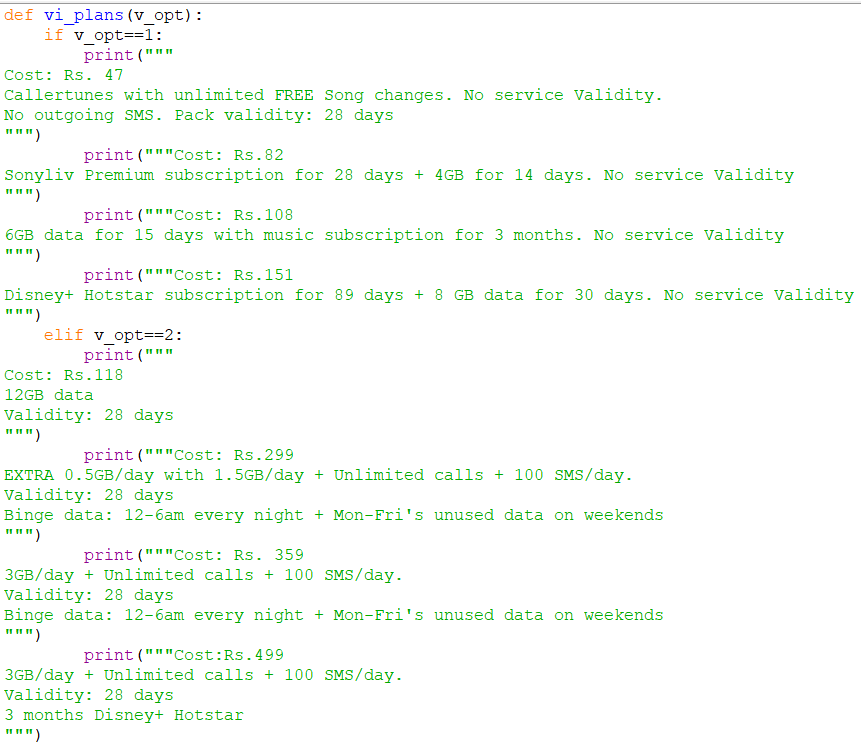


Fig 1.2.1: Screenshot of program of vi\_plans.py

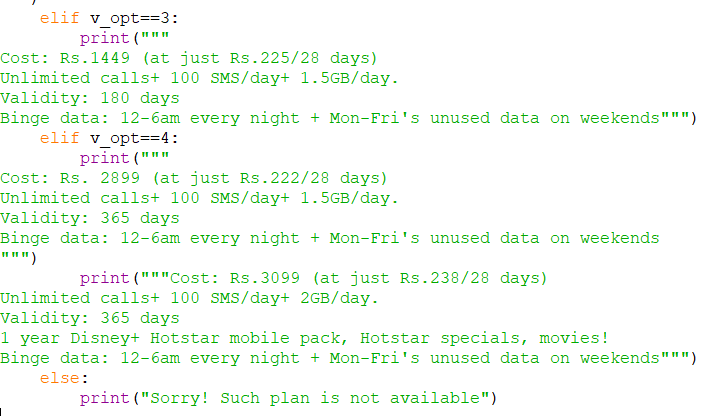


Fig 1.2.2: Screenshot of program of vi\_plans.py

1.3: Module of Jio recharge plans:

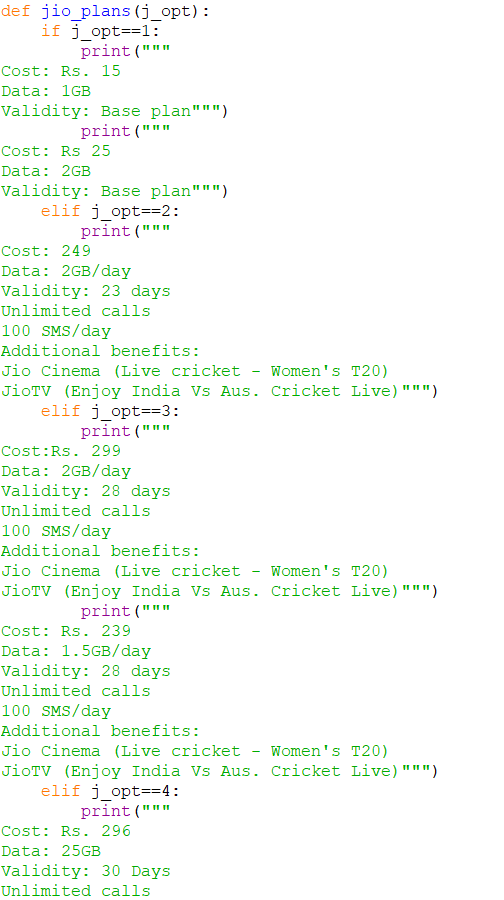


Fig 1.3.1: Screenshot of program of jio\_plans.py

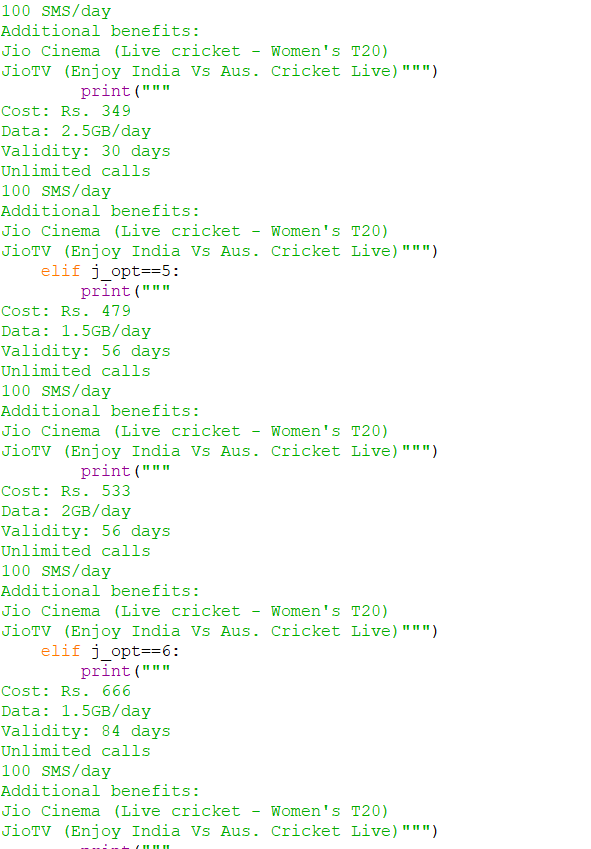


Fig 1.3.2: Screenshot of program of jio\_plans.py

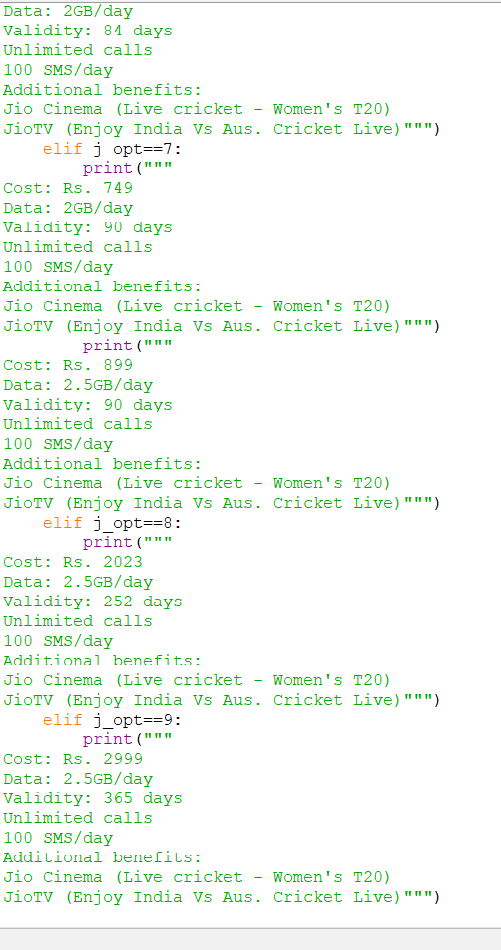


Fig 1.3.3: Screenshot of program of jio\_plans.py

1.4: Program to handle recharging customers:

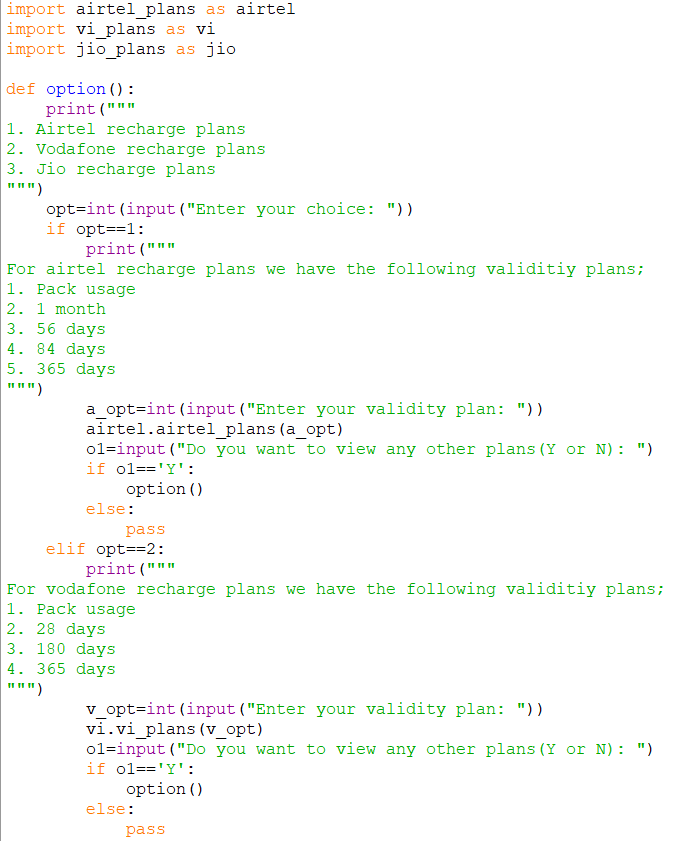


Fig 1.4.1: Screenshot of program of rec-plans

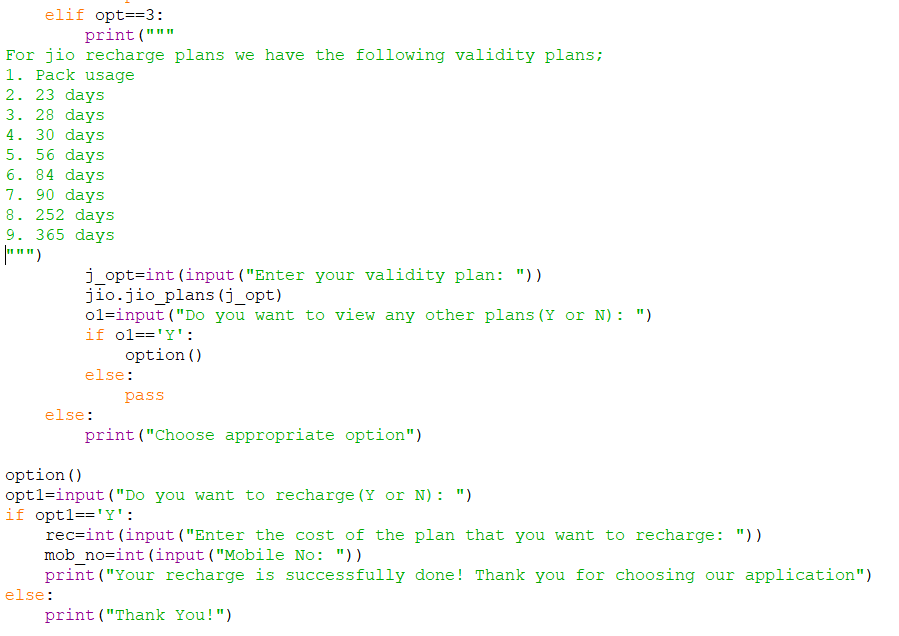


Fig 1.4.2: Screenshot of program of rec\_plans

1.5: Output of the program:

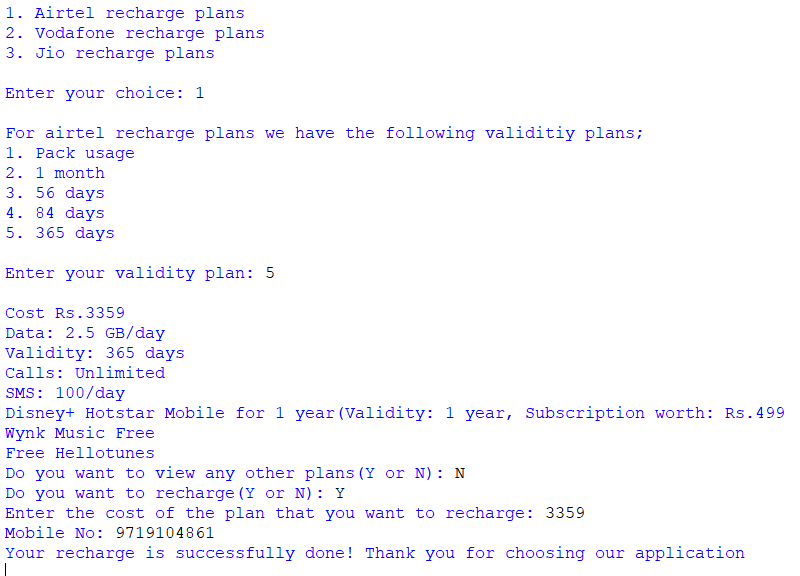


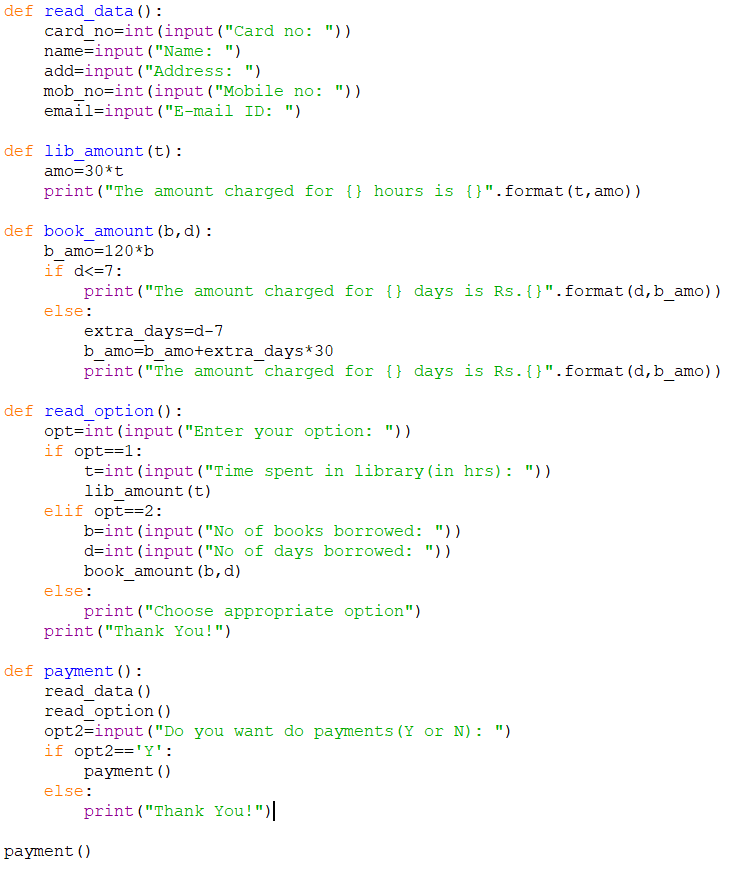
Fig 1.5.1: Screenshot of output of rec\_plans

**WEEK-2**

Design a data structure for handling student records- Designing a solution, Implementation (Using ADT)

Here we are handling the library customer’s payment details. We are charging Rs. 30 for each hour spending in library for reading. And we are offering the customer to borrow the book for 3 days as they have a library card and charging Rs. 30 for each extra days.

2.1: Program:

Fig 2.1.1: Screenshot of program of library.py

2.2: Output:

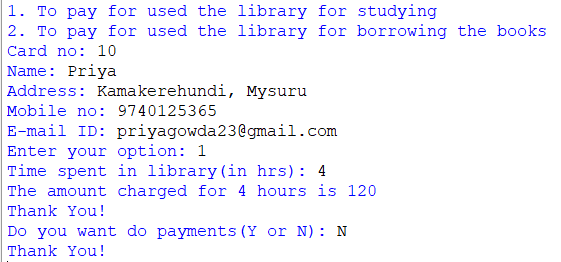


Fig 2.2.1: Screenshot of output of library.py

In the above output, we have demonstrated the working of the program we have handled the payment for used the library for reading.

**WEEK-3**

Optimize your solution (Bubble sort, Selection sort, Insertion sort)

**3.1: Bubble Sort:**

**Bubble sort** is [a sorting algorithm](https://www.programiz.com/dsa/sorting-algorithm) that compares two adjacent elements and swaps them until they are in the intended order.

**3.1.1: Working of Bubble Sort**

Suppose we are trying to sort the elements in ascending order.

1. First Iteration (Compa re and Swap)

1. Starting from the first index, compare the first and the second elements.
2. If the first element is greater than the second element, they are swapped.
3. Now, compare the second and the third elements. Swap them if they are not in order.

The above process goes on until the last element.

**2. Remaining Iteration**

The same process goes on for the remaining iterations.

After each iteration, the largest element among the unsorted elements is placed at the end.

In each iteration, the comparison takes place up to the last unsorted element.

The array is sorted when all the unsorted elements are placed at their correct positions.

3.1.3. The process of bubble sort continues until it passes one complete pass without any swaps, then the algorithm gets to know that the list or array is got sorted.

**3.1.2: Bubble sort algorithm is as follows:**

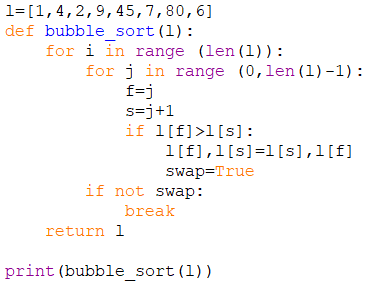


Fig 3.1.2.1: Screenshot of program of bubble\_sort.py

**3.1.3: Output:**



Fig 3.1.3.1: Screenshot of output of bubble\_sort.py

**3.1.4: 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(n)  
  If the array is already sorted, then there is no need for sorting.
* **Average Case Complexity: O(n2)**  
  It occurs when the elements of the array are in jumbled order (neither ascending nor descending).

**3.1.5: Space Complexity:**

* Space complexity is O(1) because an extra variable is used for swapping.
* In the **optimized bubble sort algorithm**, two extra variables are used. Hence, the space complexity will be O(2).

**3.2: Selection Sort:**

Selection sort is [a sorting algorithm](https://www.programiz.com/dsa/sorting-algorithm) that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

**3.2.1: Working of Selection Sort**

1) Set the first element as minimum.

Compare minimum with the second element. If the second element is smaller than minimum, assign the second element as minimum.  
  
2) Compare minimum with the third element. Again, if the third element is smaller, then assign minimum to the third element otherwise do nothing. The process goes on until the last element.

3) After each iteration, minimum is placed in the front of the unsorted list.

For each iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions.

**3.2.2: Selection sort algorithm:**

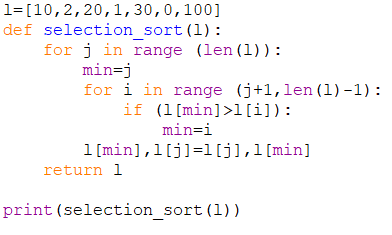


Fig 3.2.2.1: Screenshot of program of selection sort.py

**3.2.3: Output:**



Fig 3.2.3.1: Screenshot of output of selection sort.py

Complexity = 0(n)2

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

**3.2.4: Time Complexities:**

* Worst Case Complexity: O(n)2  
  If we want to sort in ascending order and the array is in descending order then, the worst case occurs.
* Best Case Complexity: O(n)2  
  It occurs when the array is already sorted
* Average Case Complexity: O(n)2  
  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.

**3.2.5: Space Complexity:**

Space complexity is O(1) because an extra variable temp is used.

**3.3: Insertion Sort:**

Insertion sort is [a sorting algorithm](https://www.programiz.com/dsa/sorting-algorithm) that places an unsorted element at its suitable place in each iteration.

Insertion sort works similarly as we sort cards in our hand in a card game.

We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put in their right place.

A similar approach is used by insertion sort.

## 3.3.1: Working of Insertion Sort

Suppose we need to sort the following array.

1. The first element in the array is assumed to be sorted. Take the second element and store it separately in key.  
     
   Compare key with the first element. If the first element is greater than key, then key is placed in front of the first element. If the first element is greater than key, then key is placed in front of the first element.
2. Now, the first two elements are sorted.  
     
   Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.
3. Similarly, place every unsorted element at its correct position.

**3.3.2: Insertion Sorting algorithm:**

**3.3.3: Output:**

**3.3.4: Time Complexities:**

* Worst Case Complexity: O(n2)  
  Suppose, an array is in ascending order, and you want to sort it in descending order. In this case, worst case complexity occurs.

Each element has to be compared with each of the other elements so, for every nth element, (n-1) number of comparisons are made.  
Thus, the total number of comparisons = n\*(n-1) ~n2

* Best Case Complexity: O(n)  
  When the array is already sorted, the outer loop runs for n number of times whereas the inner loop does not run at all. So, there are only n number of comparisons. Thus, complexity is linear.
* Average Case Complexity: O(n2)  
  It occurs when the elements of an array are in jumbled order (neither ascending nor descending).

**3.3.5: Space Complexity:**

Space complexity is O(1) because an extra variable key is used.