**WEEK-1**

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

**1.1: Data Structure:**

Data means the collection of information, structure means the specific shape that is outlined, and the Data structure means organizing the data in a specific order or shape.

The data structures are the organizers and storers of data in an efficient manner so that they can be modified and accessed in the future.

**Two Types of Data Structures**

1. **Linear Data Structures**: In linear data structures, the elements are arranged in sequence.
2. **Non-linear Data Structures**: In non-linear data structures, the elements are arranged in a hierarchical manner where one element will be connected to one or more elements.

|  |  |
| --- | --- |
| Linear DS | Non-linear DS |
| Data items are arranged in sequential order, one after the other | Data items are arranged in non-sequential order (hierarchical order) |
| All the items are present on the same layer | The data items are present at different layers |
| The memory utilization is not efficient | Different data structures utilize memory in different efficient ways depending on the need |
| The time complexity increase with the data size | Time complexity remains the same |
| Example: Arrays, Stack, Queue | Example: Tree, Graph, Map |

Table 1.1: Difference between Linear and Non-linear Data structures

**1.1.1: Python build-in data structure:** These are the data structures that come along with Python and can be implemented same as primitive data types like integers, etc.

Some built-in data structures in python are as follows:

1. **Lists:**

Lists are the containers of values of different data types. These store the data in a continuous manner in the memory.

1. **Tuples**:

A tuple is also a collection of elements of different data types. The difference between a tuple and a list is that lists are mutable whereas tuples are not.

1. **Dictionaries**:

Dictionaries is again a collection of multiple elements in the form of key, value pair as a item. Here the value is represented by its respective key. It does’nt allows duplicate keys in a dictionary.

1. **Sets**:

Sets are the collection of elements of different data types. It does’nt allows duplicate values and the elements are not ordered of a set.

**1.1.2: Python user-defined data structure:** These data structures are the ones built using the built-in data structures and have their own properties.

Some user-defined data structures in python are as follows:

1. **Arrays**:

These are the data structures similar to lists. The only difference is that these are homogeneous, that is, have the elements of the same data type.

1. **Stacks**:

These are the data structures that work following the principle of LIFO (Last In First Out). The elements enter and leave at the same node called the top. We can access only the last entered element using the top node.

1. **Queues**:

These data structures follow the principle of FIFO (First In First Out). The elements enter at one end and leave at the other end. It is similar to the queues we see ATM centres, mess, etc.

1. **Linked lists**:

These are linear data structures that hold the values of the next element using the pointers. A node consists of two values, one is the data and the other is the pointer to the next node.

1. **Hash Maps**:

Hashmaps are the data structures similar to the dictionaries in Python. The only difference is that it maps the keys to the values by taking the hash value of that key.

1. **Trees**:

These are non-linear data structures that form a hierarchy like structure. A tree has a root node, from where the tree starts and which represents the tree. The other nodes are formed by descending to the root. For a node, the preceding node is called the parent and succeeding node is called the child.

1. **Graphs**:

These are also non-linear data structures that store nodes and edges connecting the nodes.  These have the different nodes connecting to the other nodes, representing a real word map like structure.

**1.2:** Design a data structure to handle the customer’s recharge activities

Here we are providing the customer a different recharge plans for airtel Vodafone, and Jio sims, and allowing the user to recharge their phone for their required plan.

We creating three lists each for airtel Vodafone and jio recharge plans with elements of the lists of a corresponding recharge plan with the details of it as its elements.

1.2.1: Algorithm to solve the above situation:

**Step-1**: Start

**Step-2**: Create a list name airtel with the elements of sub-lists that is the plans of airtel, and the details of the respective plan as the elements of the sub-lists.

**Step-3**: Create a list name vi with the elements of sub-lists that is the plans of vodafone, and the details of the respective plan as the elements of the sub-lists.

**Step-4**: Create a list name jio with the elements of sub-lists that is the plans of Jio, and the details of the respective plan as the elements of the sub-lists.

**Step-5**: Define the function named option() and read the option from the user either they want airtel or Vodafone or jio plans.

If the choice is airtel display the plans and details of airtel recharge,

else if the choice is Vodafone display the plans and details of Vodafone recharge,

else if the choice is jio display the plans and details of jio recharge.

**Step-6**: Read the Y or N from user to recharge or not, if the choice is Y read the cost of the plan and mobile number from the user and display the message “Your recharge is successfully done!”, else display a message “Thank You!”.

1.2.2: Program:

airtel=list[]

airtel=[[details of a respective plan],[],[],……plans available in airtel……..,[]]

In the airtel list we are creating a sub-lists with details of respective plan as its elements. The airtel list contains the elements that is lists of different recharge plans available for airtel users.

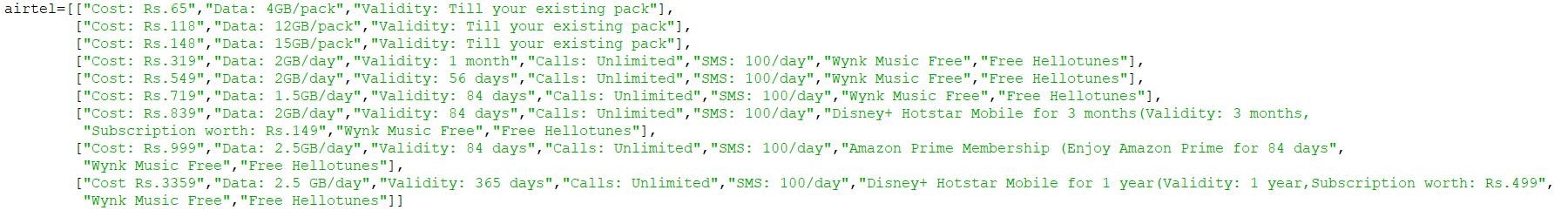


Fig 1.1: Screenshot of airtel list of rec\_plans.py

vi=list[]

vi=[[details of a respective plan],[],[],…plans available in vodafone…….[]]

In the vi list we are creating a sub-lists with details of respective plan as its elements. The vi list contains the elements that is lists of different recharge plans available for vodafone users.

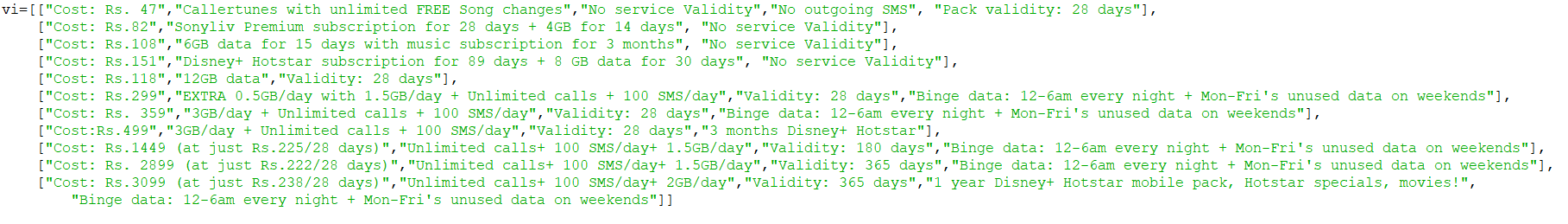


Fig 1.2: Screenshot of vi list of rec\_plans.py

jio=list[]

jio=[[details of a respective plan],[],[],……plans available in jio…….,[]]

In the jio list we are creating a sub-lists with details of respective plan as its elements. The jio list contains the elements that is lists of different recharge plans available for Jio users.

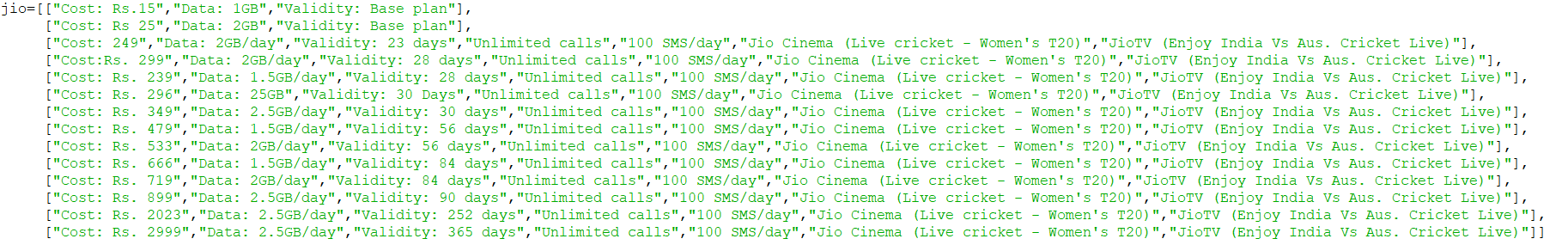


Fig 1.3: Screenshot of jio list of rec\_plans.py

Next we are asking the user to choose the sim that needs to be recharged, and displaying the plans available for the respective sim.

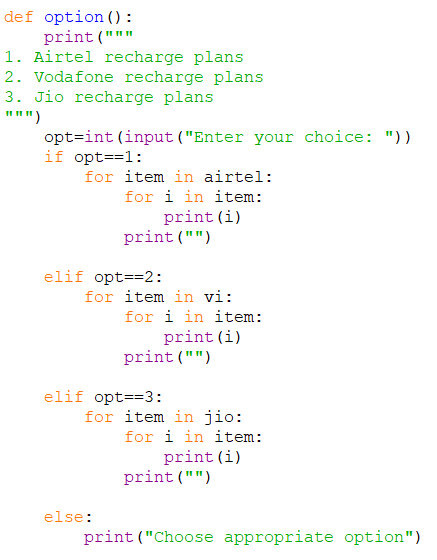


Fig 1.4: Screenshot of reading option of rec\_plans.py

Next we are asking that the user is doing recharge, if yes we are asking the cost of the required plan and mobile number.

cost=int

mob\_no=int

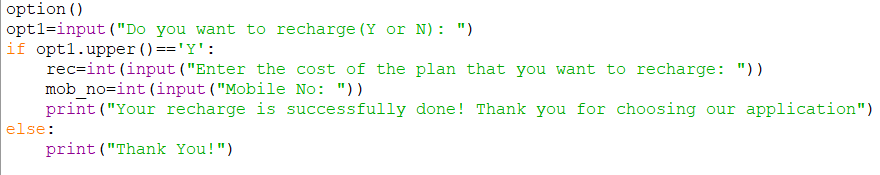


Fig 1.5: Screenshot of reading mob\_no and cost of rec\_plans.py

1.2.3: Output of the above program will be;

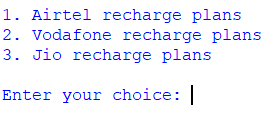
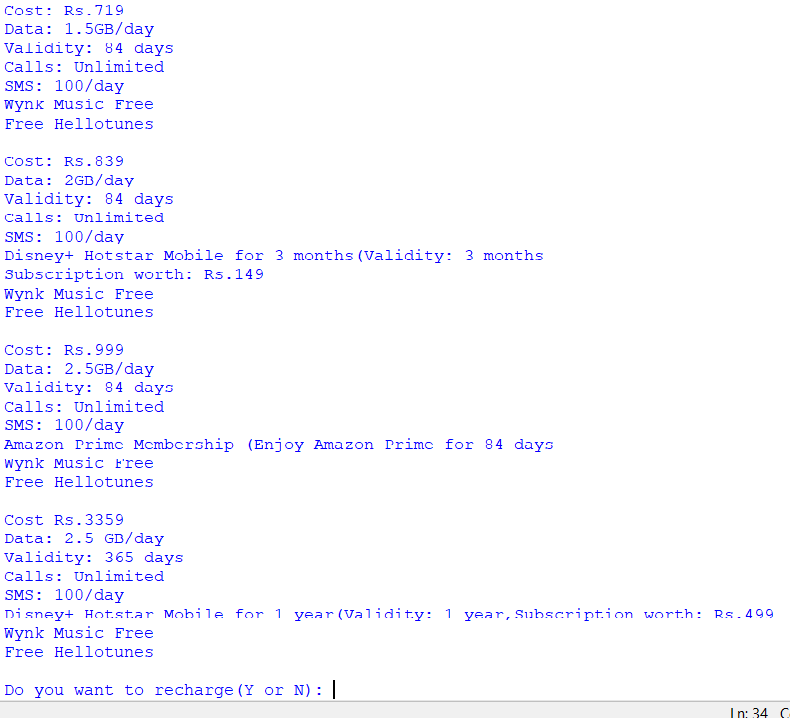


Fig 1.6: Screenshot of output of rec\_plans.py

Enter your choice according to your requirements, For example let us choose 1 for airtel;

  
Fig 1.7: Screenshot of output of rec\_plans.py

It displays the recharge plans available for airtel, and asks you that you recharge or not. Enter ‘Y’ or ‘y’ for recharging or else enter ‘N’ for exiting.

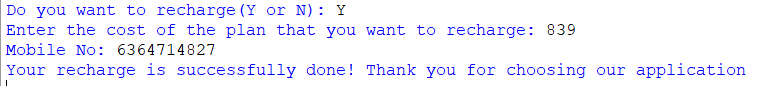


Fig 1.7: Screenshot of output of rec\_plans.py

Enter the cost of required recharge plan and mobile number.

**WEEK-2**

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

**2.1: Abstract Data Type (ADT):**

Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of values and a set of operations. The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called “abstract” because it gives an implementation-independent view.

The process of providing only the essentials and hiding the details is known as abstraction.

The built-in modules are the best examples for ADT as we use it for making several operations by importing the module and calling the different functions, and we know only what operation does the function do but don’t know how these operations are making.

**2.1.1: Key Features of ADT**:

* **Abstraction:**The user does not need to know the implementation of the data structure only essentials are provided.
* **Better Conceptualization:**ADT gives us a better conceptualization of the real world.
* **Robust:**The program is robust and has the ability to catch errors.
* **Encapsulation**: ADTs hide the internal details of the data and provide a public interface for users to interact with the data. This allows for easier maintenance and modification of the data structure.
* **Data Abstraction**: ADTs provide a level of abstraction from the implementation details of the data. Users only need to know the operations that can be performed on the data, not how those operations are implemented.
* **Data Structure Independence**: ADTs can be implemented using different data structures, such as arrays or linked lists, without affecting the functionality of the ADT.
* **Information Hiding:** ADTs can protect the integrity of the data by allowing access only to authorized users and operations. This helps prevent errors and misuse of the data.
* **Modularity**: ADTs can be combined with other ADTs to form larger, more complex data structures. This allows for greater flexibility and modularity in programming.

**2.1.2: Disadvantages:**

* **Overhead**: Implementing ADTs can add overhead in terms of memory and processing, which can affect performance.
* **Complexity**: ADTs can be complex to implement, especially for large and complex data structures.
* **Learning**Curve: Using ADTs requires knowledge of their implementation and usage, which can take time and effort to learn.
* **Limited Flexibility:** Some ADTs may be limited in their functionality or may not be suitable for all types of data structures.
* **Cost**: Implementing ADTs may require additional resources and investment, which can increase the cost of development.

**2.2:** Design a data structure for handling recharge activities (using ADT)

Here we are performing some operations on the recharge plan records like inserting new plans and deleting the existing plans.

As in the first activity the plans are inserted in the list in the form of a list that contains the details of the respective plan. So as in week-1 activity we have created 3 lists namely airtel, vi and jio with the plans as their elements.

2.2.1: Algorithm to solve the above program:

**Step-1**: Start

**Step-2**: Design a module named modify with two functions namely insert() and delete(), to make insertions and deletions of the plans

**Step-3**: Import the module modify to the file

**Step-4**: Create the 3 lists airtel, vi and jio

**Step-5**: Read the user input either he insert a new plan or delete an existing plan, If he choose insert ask to choose the sim to which the plan should be inserted, and call the function insert() with the parameters opt1, airtel, vi, jio; where opt1 is the choice of the sim.

Else if he choose delete ask to choose the sim from which the plan should be deleted, and call the function delete() with the parameters opt1, airtel, vi, jio; where opt1 is the choice of the sim.

**Step-6**: Stop

2.2.2: Program:

Firstly we are defining the functions insert and delete in a module named modify

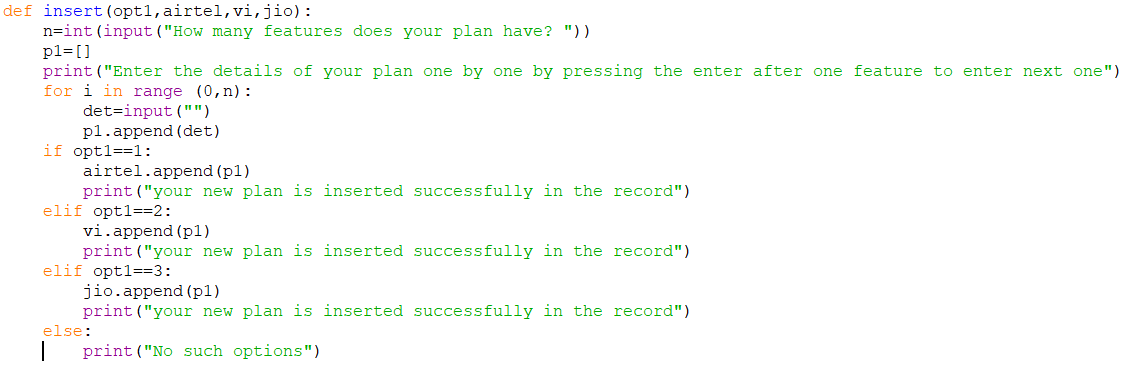


Fig 2.1: Screenshot of insert function of modify.py

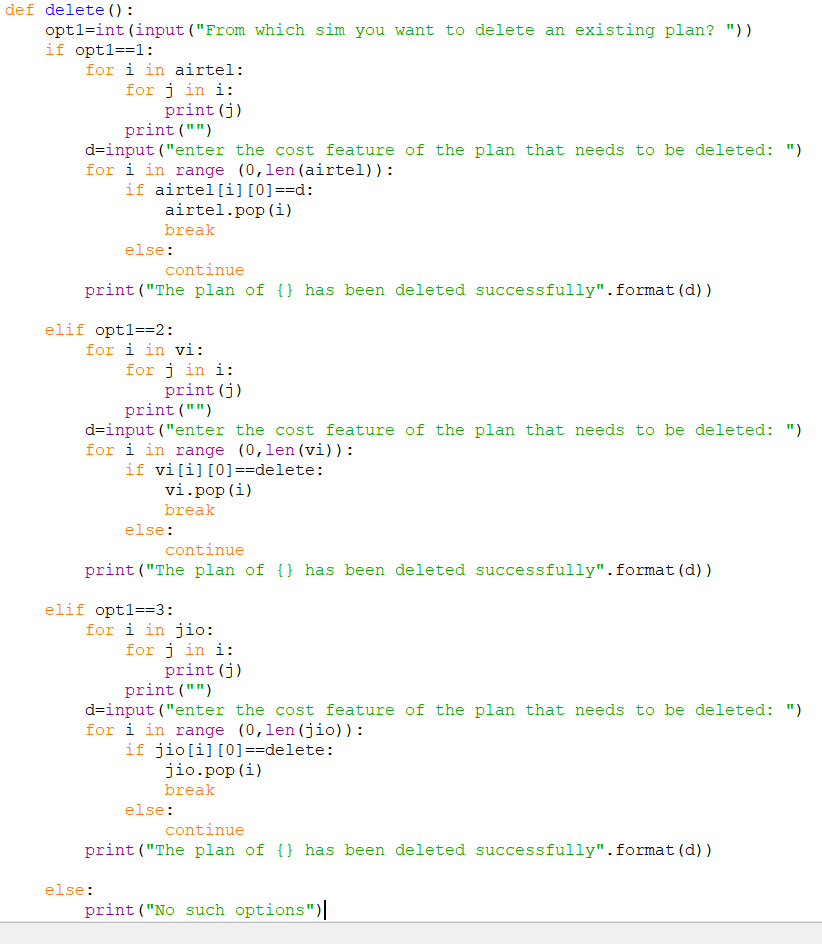


Fig 2.2: Screenshot of delete function of modify.py

Next we need to import the module named modify to the file



Fig 2.3: Screenshot of plan\_handling.py

Next we need to create the lists with the elements.

airtel=list[]

airtel=[[details of the respective plan],[],[],…plans available for airtel…..,[]]

vi=list[]

vi=[[details of respective plan],[],………plans available for vodafone…….,[]]

jio=list[]

jio=[[details of respective plan],[],[],……plans available for jio……….,[]]

Next we are reading the option to insert or delete a plan

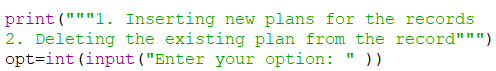


Fig 2.3: Screenshot of plan\_handling.py

If the option is inserting we are calling the insert function and doing changes to the record

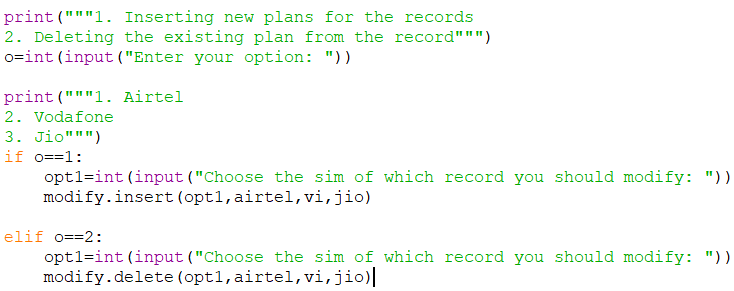


Fig 2.4: Screenshot of plan\_handling.py

**2.2.3: Output of the above program will be;**

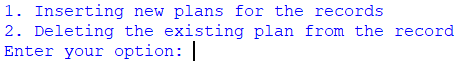


Fig 2.5: Screenshot of output of plan\_handling.py

Enter your choice, let us try to insert a new plan to the airtel plans record

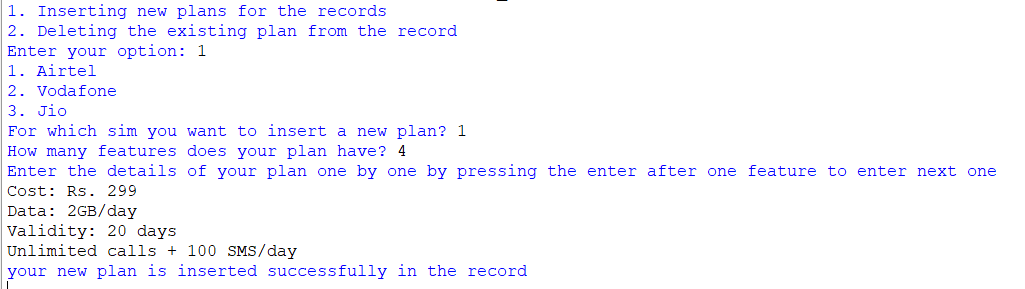


Fig 2.6: Screenshot of output of plan\_handling.py

Then we should select the sim for which we need to insert a plan, and it asks how much features does our new plan consists and allows us to entry the features one after one.

In the program we can say **insertion and deletion is simple ADT** as the process of insertion and deletion is known but don’t know how actually it inserts or deletes the plans.

**WEEK-3**

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

**3.1: Sorting:**

Sorting is defined as an arrangement of data in a certain order. Sorting techniques are used to arrange data (mostly numerical) in an ascending or descending order. It is a method used for the representation of data in a more comprehensible format. It is an important area of Computer Science. Sorting a large amount of data can take a substantial amount of computing resources if the methods we use to sort the data are inefficient. The efficiency of the algorithm is proportional to the number of items it is traversing. For a small amount of data, a complex sorting method may be more trouble than it is worth. On the other hand, for larger amounts of data, we want to increase the efficiency and speed as far as possible. We will now discuss the several sorting techniques and compare them with respect to their time complexity.

**3.2:** **The different types of sorting order are:**

* **Increasing Order:**A set of values are said to be increasing order when every successive element is greater than its previous element. For example: 1, 2, 3, 4, 5. Here, the given sequence is in increasing order.
* **Decreasing Order:**A set of values are said to be in decreasing order when the successive element is always less than the previous one. For Example: 5, 4, 3, 2, 1. Here the given sequence is in decreasing order.
* **Non-Increasing Order:** A set of values are said to be in non-increasing order if every ith element present in the sequence is greater than or equal to its (i-1)th element. This order occurs whenever there are numbers that are being repeated. For Example: 1, 2, 2, 3, 4, 5. Here 2 repeated two times.
* **Non-Decreasing Order:** A set of values are said to be in non-decreasing order if every ith element present in the sequence is less than or equal to its (i-1)th element. This order occurs whenever there are numbers that are being repeated. For Example: 5, 4, 3, 2, 2, 1. Here 2 repeated two times.

### **3.3: Sorting Techniques:**

### The different implementations of sorting techniques in Python are:

* Bubble Sort
* Selection Sort
* Insertion Sort

**3.3.1: 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.

**3.3.1.1: Working of Bubble Sort**

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

1. First Iteration (Compare 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. Second iteration**

The same process goes on for the second and remaining iterations.

The iteration continues until one iteration is processed without any swaps.

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

**Step-1:** Start

**Step-2**: Create a list ‘l’ with some elements

**Step-3**: In the first pass compare the first and second elements and swap the adjacent elements if the first element is greater than second element, and then compare second and third element and swap if second is greater than third compare the adjacent elements like this and swap if needed until the last element.

**Step-4**: Continue the iteration until one iteration is processed without any swaps

**Step-5**: print the sorted list

**Step-6**: Stop

**3.3.1.3: Optimized Bubble sort program:**

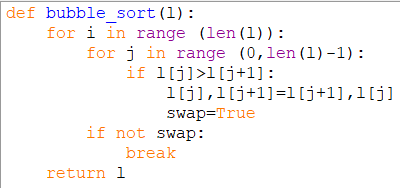


Fig 3.1: Screenshot of bubble\_sort function of sort.py

The above bubble sort program is an optimized program as we assigned True to swap if the swap is done and can optimize the time complexity by breaking from the loop if swap is not true.

**3.3.1.4: 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).

**3.3.1.5: 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).

**3.3.2: 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. During the execution of this algorithm, two subarrays are maintained, the subarray which is already sorted, and the remaining subarray which is unsorted. During the execution of Selection Sort for every iteration, the minimum element of the unsorted subarray is arranged in the sorted subarray. Selection Sort is a more efficient algorithm than bubble sort. Sort has a Time-Complexity of O(n2) in the average, worst, and in the best cases.

**3.3.2.1: 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.

**3.3.2.2: Selection sort algorithm:**

**Step-1**: Start

**Step-2**: In the first pass assign the elements as minimum, and compare the minimum with second if second is minimum than first, assign second to minimum. Then compare the minimum with third and find the minimum among two and assign it to minimum. Like this compare the minimum with all elements and assign the minimum element to minimum.

**Step-3**: In the second pass leave the first element as it is the minimum element of the list, and compare the second and third and assign value for minimum and compare the minimum with third and continue the process.

**Step-4**: Continue the process until the last element.

**Step-5**: Print the sorted list

**Step-6**: Stop

**3.3.2.3: Selection sort program:**

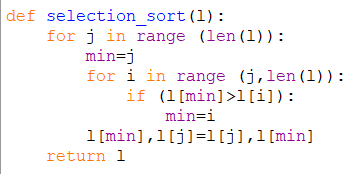


Fig 3.2: Screenshot of selection\_sort function of sort.py

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

**3.3.2.4: 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)  
  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.

**3.3.2.5: Space Complexity:**

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

**3.3.3: 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. This sorting algorithm maintains a sub-array that is always sorted. Values from the unsorted part of the array are placed at the correct position in the sorted part. It is more efficient in practice than other algorithms such as selection sort or bubble sort. Insertion Sort has a Time-Complexity of O(n2) in the average and worst case, and O(n) in the best case.

## 3.3.3.1: Working of Insertion Sort

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.

**3.3.3.2: Insertion Sorting algorithm:**

**Step-1**: Start

**Step-2**: In the first pass assume first element as the minimum element and compare it with the second one, if the second is less than first then swap the elements. Then compare the third element with first and second elements and insert the third element to the right position according to the comparison.

**Step-3**: Next compare the fourth element with the previous ones and place it in the right position. Continue this process until the last position.

**Step-4**: Display the sorted list

**Step-5**: Stop

**3.3.3.3: Insertion sort program:**

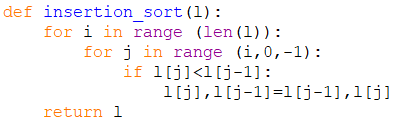


Fig 3.3: Screenshot insertion\_sort function of sort.py

**3.3.3.4: 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).

**3.3.3.5: Space Complexity:**

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

**3.3.4: Menu driven Program to sort the user defined list using the required sorting technique**

The functions insertion\_sort(), selection\_sort(), and insertion\_sort() are defined in the program as in the above screenshots, the main program is as below;

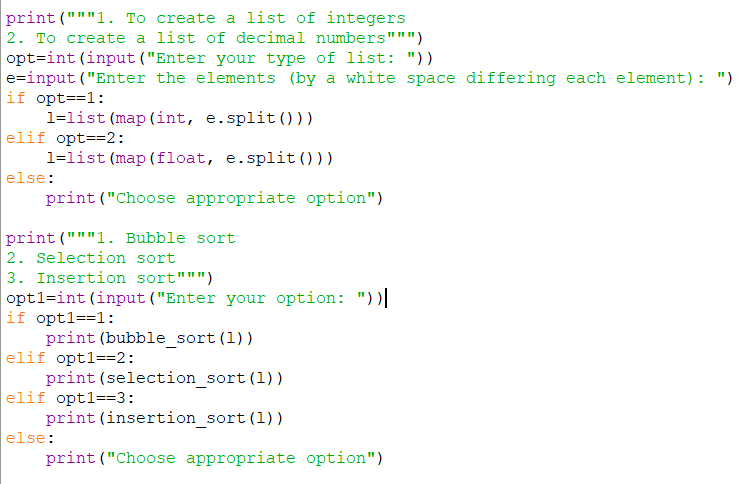


Fig 3.4: Screenshot of main program of sort.py

**The output of the above program will be;**

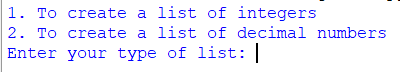


Fig 3.5: Screenshot of output of sort.py

To create and sort the list of integers enter 1, and enter 2 to create and sort the list of decimal numbers; For example let us create a list of integers

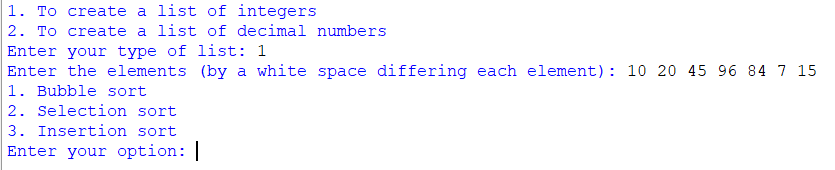


Fig 3.6: Screenshot of output of sort.py

Next enter the elements of the list by giving a whitespace between each element, Next choose which sorting technique do you want to use to sort your list. For now let us use the selection sorting technique to sort our list and the output will be;

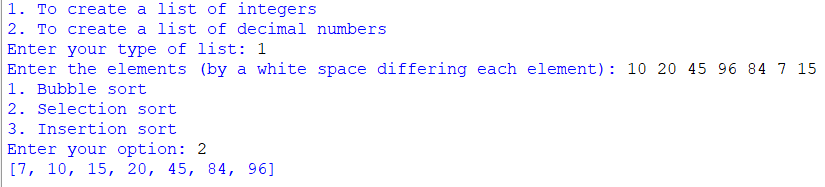


Fig 3.7: Screenshot of output of sort.py