Assignment no. 5	5
Aim	**Write a python program to store second year percentage of students in array. Write function for sorting array of floating-point numbers in ascending order using a) Insertion sort b) Shell Sort and display top five scores**
Objective	To understand the concept of a sorting method To find the performance of sorting method To apply sorting method to sort the given array/list in ascending order and display top five elements.
Outcome	To design and implement the sorting techniques. To calculate time and space complexity of the given sorting methods To apply the functions of sorting methods on given data for sorting the elements in ascending order and display top five elements.
OS/Programming tools used	(64-Bit) 64-BIT Fedora 17 or latest 64-BIT Update of Equivalent Open-source OS or latest 64-BIT Version and update of Microsoft Windows 7 Operating System onwards Programming Tools (64-Bit) Eclipse with Python plugin or Pycharm IDE

Theory related to assignment:

In this assignment we will implement the given sorting techniques.

For this purpose, we need to accept the student's data of percentages in array or list. The percentages should be accepted as floating-point numbers. Thereafter, apply the following sorting operations to sort the percentages in ascending order.

The program should display the sorted elements in ascending order as well as display the top five scores amongst them.

Sorting is the process of arranging the given data in some pre-defined order or sequence. The order can be either ascending or descending.

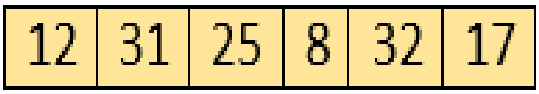
Insertion Sort:

- Insertion sort works similar to the sorting of playing cards in hands.
- It is assumed that the first card is already sorted in the card game, and then we select an unsorted card.

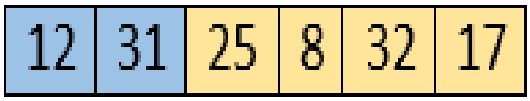
- If the selected unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side. Similarly, all unsorted cards are taken and put in their exact place.
- The same approach is applied in insertion sort. The idea behind the insertion sort is that first take one element, iterate it through the sorted array.
- Although it is simple to use, it is not appropriate for large data sets as the time complexity of insertion sort in the average case and worst case is O(n2), where n is the number of items.
- Insertion sort is less efficient than the other sorting algorithms like heap sort, quick sort, merge sort, etc

Example of Insertion sort

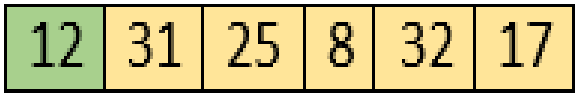
Let the elements of array are –



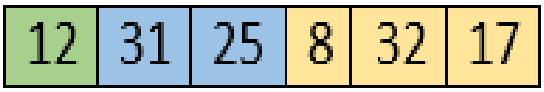
Initially, the first two elements are compared in insertion sort.

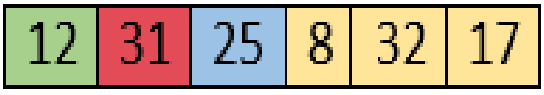


Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.



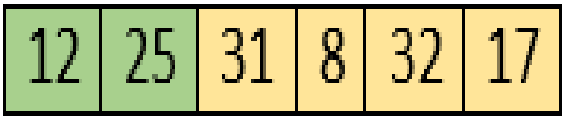
Now, move to the next two elements and compare them.



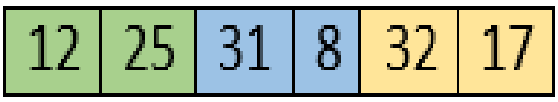


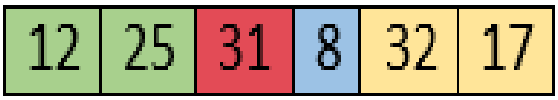
Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.

For now, the sorted array has only one element, i.e. 12. So, 25 is greater than 12. Hence, the sorted array remains sorted after swapping.



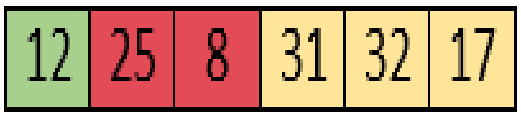
Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.



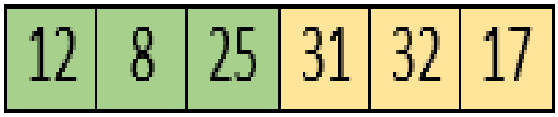


Both 31 and 8 are not sorted. So, swap them.

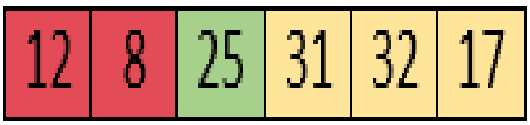
After swapping, elements 25 and 8 are unsorted.



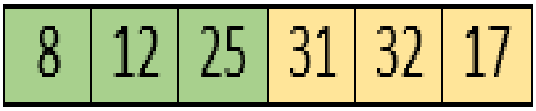
So, swap them.



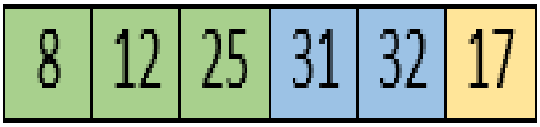
Now, elements 12 and 8 are unsorted.



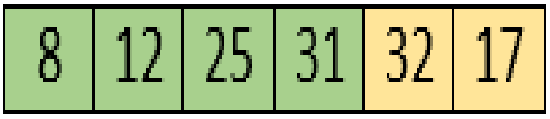
So, swap them too.



Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.



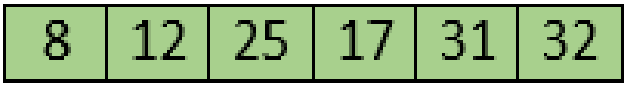
Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.

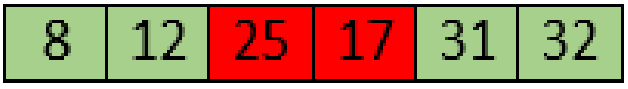


Move to the next elements that are 32 and 17.

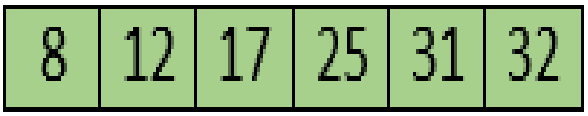
17 is smaller than 32. So, swap them.

Swapping makes 31 and 17 unsorted. So, swap them too.





Now, swapping makes 25 and 17 unsorted. So, perform swapping again.



Now, the array is completely sorted.

Pseudocode for Insertion Sort:

```
Algorithm insertionSort(A [] ) {
        for i = 1 to length(A) inclusive do:
        /* select value to be inserted */
        valueToInsert = A[i]
        holePosition = i
        /*locate hole position for the element to be inserted */
        while holePosition > 0 and A[holePosition-1] > valueToInsert do:
                A[holePosition] = A[holePosition-1]
                holePosition = holePosition -1
        end while
        /* insert the number at hole position */
        A[holePosition] = valueToInsert
        end for
end procedure
```

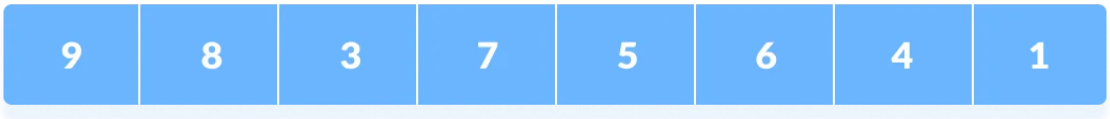
Time Complexity:
Worst Case – $O(n^2)$
Average Case – $O(n^2)$

Shell Sort:

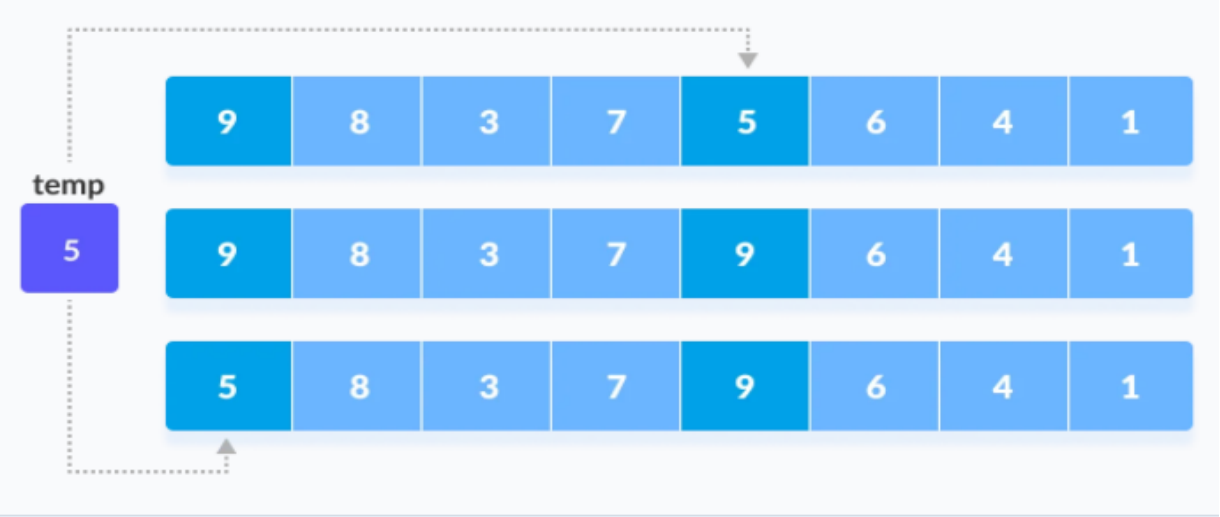
- Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm.
- This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.
- This algorithm uses insertion sort on a widely spread elements, first to sort them and then sorts the less widely spaced elements. This spacing is termed as interval.
- This algorithm is quite efficient for medium-sized data sets as its average and worst-case complexity of this algorithm depends on the gap sequence the best known is O(n), where n is the number of items. And the worst case space complexity is O(n).

Example of Shell Sort

Suppose, we need to sort the following array.

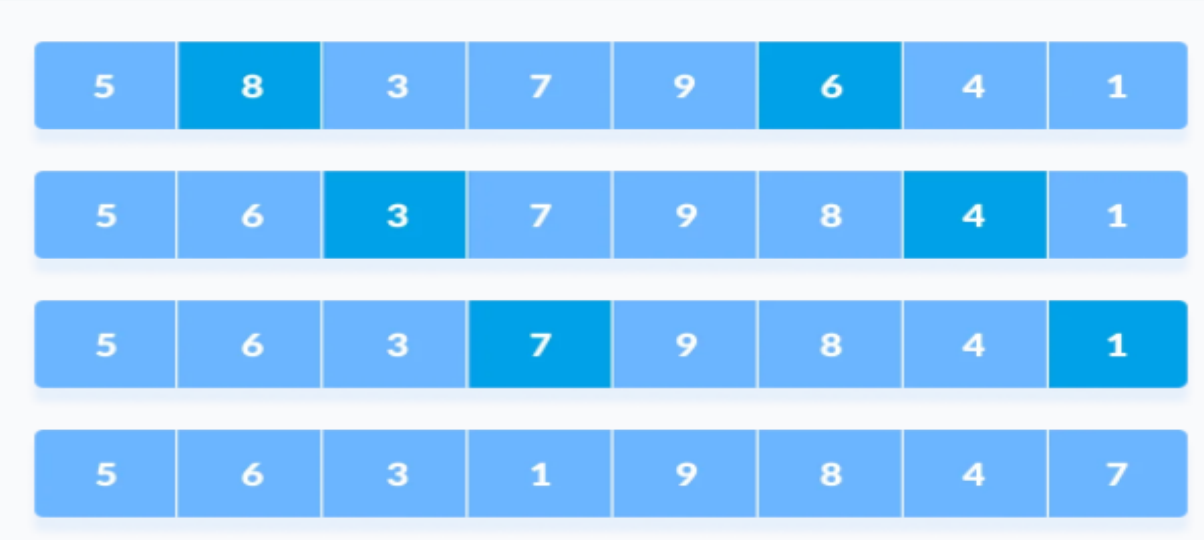


- We are using the shell's original sequence (N/2, N/4, ...1) as intervals in our algorithm.
- In the first loop, if the array size is N = 8 then, the elements lying at the interval of N/2 = 4 are compared and swapped if they are not in order.
- The 0th element is compared with the 4th element.
- If the 0th element is greater than the 4th one then, the 4th element is first stored in temp variable and the 0th element (ie. greater element) is stored in the 4th position and the element stored in temp is stored in the 0th position.



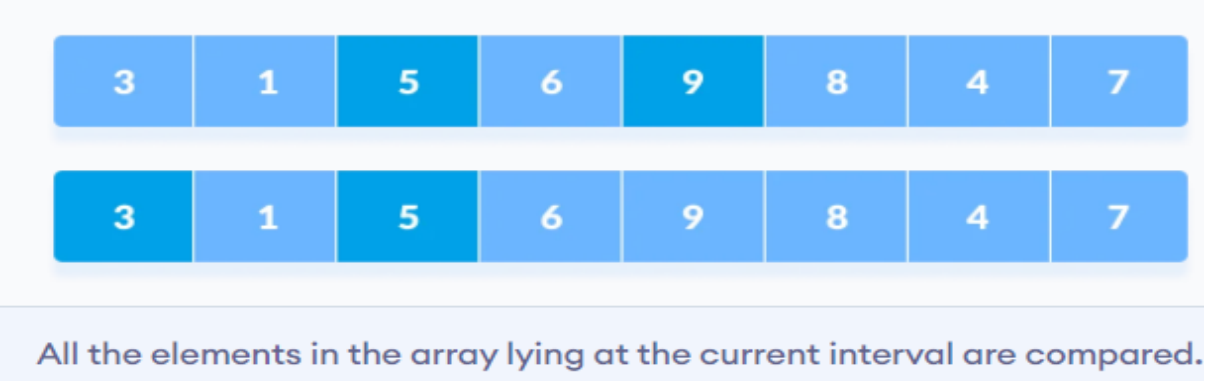
Rearrange the elements at n/2 interval

- This process goes on for all the remaining elements.

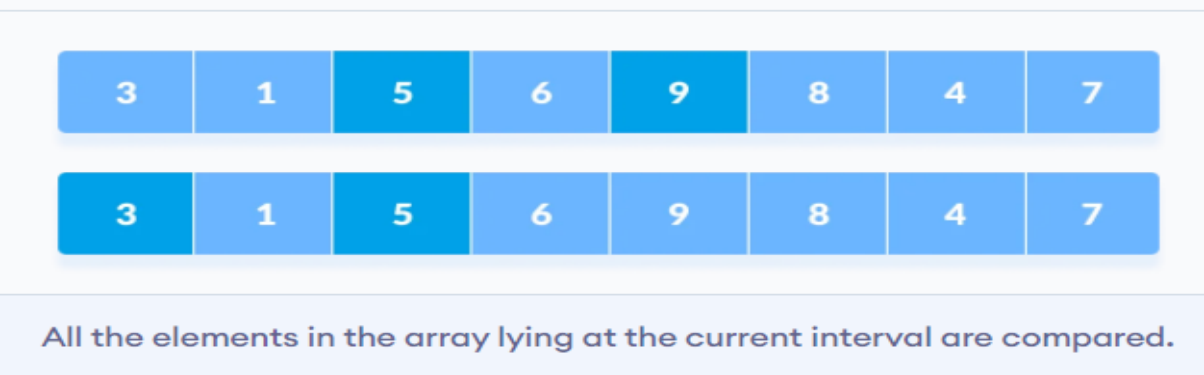


Rearrange all the elements at n/2 interval

- In the second loop, an interval of N/4 = 8/4 = 2 is taken and again the elements lying at these intervals are sorted.



- The elements at 4th and 2nd position are compared. The elements at 2nd and 0th position are also compared. All the elements in the array lying at the current interval are compared.
- The same process goes on for remaining elements.



- Finally, when the interval is N/8 = 8/8 =1 then the array elements lying at the interval of 1 are sorted. The array is now completely sorted.

Pseudocode Shell Sort

```
Algorithm shellSort(A[]) {
```

```
  /* calculate interval*/
  while interval < A.length /3 {
    interval = interval * 3 + 1
}

  while interval > 0 {

    for outer = interval to A.length; {

    /* select value to be inserted */
    valueToInsert = A[outer]
    inner = outer;

      /*shift element towards right*/
      while inner > interval -1 && A[inner - interval] >= valueToInsert {
            A[inner] = A[inner - interval]
            inner = inner - interval
}

    /* insert the number at hole position */
    A[inner] = valueToInsert

} //end for

  /* calculate interval*/
  interval = (interval -1) /3;
} //  end while

end procedure
```

Time Complexity:
Worst case, Average case: O(n)

Conclusion:

The functions for different sorting methods are implemented successfully on student percentage data with efficient time and space complexity and displayed top five scores amongst them.

Review Questions:

1. Explain the concept of sorting?
1. Define and explain insertion sort?
2. Define and explain shell sort?
3. How many passes are required in insertion and shell sort?
4. What is the time complexity of insertion sort?
5. What is the time complexity of shell sort?
6. Trace the step-by-step procedure of insertion sort on a sample array of elements.
7. Trace the step-by-step procedure of shell sort on a sample array of elements.