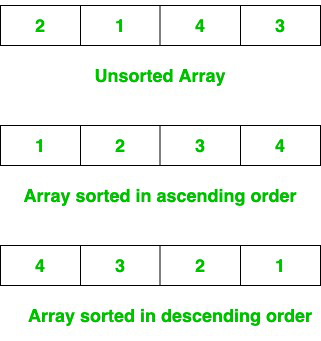
# Sorting Algorithms in Python

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.



Some of the real-life examples of sorting are:

* **Telephone Directory:**  It is a book that contains telephone numbers and addresses of people in alphabetical order.
* **Dictionary:** It is a huge collection of words along with their meanings in alphabetical order.
* **Contact List:** It is a list of contact numbers of people in alphabetical order on a mobile phone.

Before discussing the different algorithms used to sort the data given to us, we should think about the operations which can be used for the analysis of a sorting process. First, we need to compare the values to see which one is smaller and which one is larger so that they can be sorted into an order, it will be necessary to have an organized way to compare values to see that if they are in order.

**The different types of 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 increasing 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.

### Sorting Techniques

The different implementations of sorting techniques in Python are:

* Bubble Sort
* Selection Sort
* Insertion Sort

### Bubble Sort

Bubble Sort is a simple sorting algorithm. This sorting algorithm repeatedly compares two adjacent elements and swaps them if they are in the wrong order. It is also known as the sinking sort. It has a time complexity of O(n2) in the average and worst cases scenarios and O(n) in the best-case scenario. Bubble sort can be visualized as a queue where people arrange themselves by swapping with each other so that they all can stand in ascending order of their heights. Or in other words, we compare two adjacent elements and see if their order is wrong, if the order is wrong we swap them. (i.e arr[i] > arr[j] for 1 <= i < j <= s; where s is the size of the array, if array is to be in ascending order, and vice-versa).

**Example**

Here we sort the following sequence using bubble sort

Sequence: 2, 23, 10, 1

**First Iteration**

(**2, 23**, 10, 1) –> (**2, 23**, 10, 1), Here the first 2 elements are compared and remain the same because they are already in ascending order.

(2, **23, 10**, 1) –> (2, **10, 23**, 1), Here 2nd and 3rd elements are compared and swapped(10 is less than 23) according to ascending order.

(2, 10, **23, 1**) –> (2, 10, **1, 23**), Here 3rd and 4th elements are compared and swapped(1 is less than 23) according to ascending order

At the end of the first iteration, the largest element is at the rightmost position which is sorted correctly.

**Second Iteration**

(**2, 10**, 1, 23) –> (**2, 10,**1, 23), Here again, the first 2 elements are compared and remain the same because they are already in ascending order.

(2, **10, 1**, 23) –> (2, **1, 10**, 23), Here 2nd and 3rd elements are compared and swapped(1 is less than 10) in ascending order.

At the end of the second iteration, the second largest element is at the adjacent position to the largest element.

**Third Iteration**

(**2, 1**, 10, 23) –> (**1, 2**, 10, 23), Here the first 2 elements are compared and swap according to ascending order.

The remaining elements are already sorted in the first and second Iterations. After the three iterations, the given array is sorted in ascending order. So the final result is 1, 2, 10, 23.

**Implementation of Bubble Sort:**

* Python3

|  |
| --- |
| # Python3 program for Bubble Sort Algorithm Implementation  def bubbleSort(arr):        n = len(arr)        # For loop to traverse through all      # element in an array      for i in range(n):          for j in range(0, n - i - 1):                # Range of the array is from 0 to n-i-1              # Swap the elements if the element found              #is greater than the adjacent element              if arr[j] > arr[j + 1]:                  arr[j], arr[j + 1] = arr[j + 1], arr[j]    # Driver code    # Example to test the above code  arr = [ 2, 1, 10, 23 ]    bubbleSort(arr)    print("Sorted array is:")  for i in range(len(arr)):      print("%d" % arr[i]) |

**Output:**

Sorted array is:

1

2

10

23

### Selection Sort

This sorting technique repeatedly finds the minimum element and sort it in order. Bubble Sort does not occupy any extra memory space. 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.

**Example**

Here we sort the following sequence using the selection sort

Sequence: 7, 2, 1, 6

(7, 2**, 1**, 6) –> (**1**, 7, 2, 6), In the first traverse it finds the minimum element(i.e., 1) and it is placed at 1st position.

(1, 7, **2**, 6) –> (1, **2**, 7, 6), In the second traverse it finds the 2nd minimum element(i.e., 2) and it is placed at 2nd position.

(1, 2**, 7, 6**) –> (1, 2, **6, 7**), In the third traverse it finds the next minimum element(i.e., 6) and it is placed at 3rd position.

After the above iterations, the final array is in sorted order, i.e., 1, 2, 6, 7.

**Implementation of Selection Sort**

* Python3

|  |
| --- |
| # Selection Sort algorithm in Python  def selectionSort(array, size):        for s in range(size):          min\_idx = s            for i in range(s + 1, size):                # For sorting in descending order              # for minimum element in each loop              if array[i] < array[min\_idx]:                  min\_idx = i            # Arranging min at the correct position          (array[s], array[min\_idx]) = (array[min\_idx], array[s])    # Driver code  data = [ 7, 2, 1, 6 ]  size = len(data)  selectionSort(data, size)    print('Sorted Array in Ascending Order is :')  print(data) |

**Output:**

Sorted Array in Ascending Order is :

[1, 2, 6, 7]

### 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.

**Example**

Here we sort the following sequence using the insertion sort

Sequence: 7, 2, 1, 6

(**7, 2**, 1, 6) –> (**2, 7**, 1, 6), In the first iteration, the first 2 elements are compared, here 2 is less than 7 so insert 2 before 7.

(2, **7, 1**, 6) –> (2, **1, 7**, 6), In the second iteration the 2nd and 3rd elements are compared, here 1 is less than 7 so insert 1 before 7.

(**2, 1**, 7, 6) –> (**1, 2**, 7, 6), After the second iteration (1, 7) elements are not in ascending order so first these two elements are arranged. So, insert 1 before 2.

(1, 2, **7, 6**) –> (1, 2, **6, 7**), During this iteration the last 2 elements are compared and swapped after all the previous elements are swapped.

**Implementation of Insertion Sort**

* Python3

|  |
| --- |
| # Creating a function for insertion sort algorithm  def insertion\_sort(list1):            # Outer loop to traverse on len(list1)          for i in range(1, len(list1)):                a = list1[i]                # Move elements of list1[0 to i-1],              # which are greater to one position              # ahead of their current position              j = i - 1                while j >= 0 and a < list1[j]:                  list1[j + 1] = list1[j]                  j -= 1                list1[j + 1] = a            return list1    # Driver code  list1 = [ 7, 2, 1, 6 ]  print("The unsorted list is:", list1)  print("The sorted new list is:", insertion\_sort(list1)) |

**Output:**

The unsorted list is: [7, 2, 1, 6]

The sorted new list is: [1, 2, 6, 7]