Sorting Algorithm

```
class insertion:
  """
  Class for insertion sort
  """
  def __init__(self, array):
    self.array = array # array to be sorted

  def sort(self):
    for i in range(1, len(self.array)):   # start from the second element
      key = self.array[i]                 #Key is in the right position
      j = i - 1                           # j is the index of the element before key
      while j >= 0 and self.array[j] > key:
        self.array[j + 1] = self.array[j] # move the element to the right
        j -= 1
      self.array[j + 1] = key             # insert key to the right position

  def print(self):
    print(self.array)
```

```
obj=insertion([6, 3, 4, 6, 1, 5])
obj.sort()
obj.print()
```

Output:

[1, 3, 4, 5, 6, 6]

Comparison With Merge Sort

Algorithm	**Best Case Time Complexity**	**Average Case Time Complexity**	**Worst Case Time Complexity**
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)
Merge Sort	Ω(nlogn)	Θ(nlogn)	O(nlogn)

Tim Sorting

Algorithm

1. Define a constant, MIN_RUN, which determines the minimum size of a run (a run is a subsequence of the array that is already sorted).
2. Divide the input array into smaller runs. If the size of the array is less than MIN_RUN, perform an insertion sort on the entire array to create a run. If the size of the array is greater than MIN_RUN, use a combination of insertion sort and merging to create the runs.
3. Merge the runs using a modified merge sort approach. Start by considering the smallest runs and repeatedly merge adjacent runs until only one run remains.
4. During the merge process, ensure that the resulting runs are sorted by utilizing insertion sort or other efficient sorting algorithms for small subarrays.
5. Continue merging the runs until only one sorted array remains.