Project I: Comparison-based Sorting Algorithms

Implement by: Sumedh Joglekar
Language: Python
Sorting Performed: Insertion Sort, Merge Sort, In-Place Quick Sort, Modified Quick Sort

1. **Insertion Sort** :-

- Algorithm:-
  a. Defined an array n which contains the range of numbers for which sorting needs to be performed.
  b. In order to measure the time required for the execution stating timer just after main for loop.
  c. Array 'arr' will contain the all the randomly generated numbers.
  d. Calling insertionSort(arr) function which will return the sorted array of sorted elements.
  e. In the insertion sort function, setting key = arr[j] where j will be from 1 till the len(arr)
  f. While ((i>=0) and (arr[i] > key)), swapping the array elements.
  g. Updating the a[i+1] with the existing key.

- Time Complexity of Insertion Sort is O(n^2) in worst case.
- Best Case of insertion sort is sorted array and reversely sorted array and time complexity in best case is O(n).

2. **Merge Sort :-**

- Algorithm:-
  a. Merge Sort is recursive sort which divides given array into exactly half and send to the same function recursively until the length of the array is less than or equal to 1.
  b. In the code, merge_sort function is dividing the input recursively into two parts, which keeps on creating internal stack till the last call.
  c. Once the last call to merge_sort is made, program counter goes to merge(left,right), which combines the two part of sorted array.
  d. Sorting is done at the time of merging i.e. in merge function.
  e. Lastly insertion of remaining elements either from left array or right array is done depending upon the size of the array.
  f. At the end, timer is stopped and the difference between stop and start will give the execution time of the entire program.

- Time Complexity of Merge Sort is O(nlogn).
- There is no best or worst case for Merge Sort as we are dividing array in exact half regardless of the size and the position of the elements.

3. **Quick Sort** :-

- Algorithm:-
    a. Array 'arr' contains the randomly generated elements with varying length from 500 to 50000
    b. Function inPlaceQuickSort(arr) calls inPlacePartition which is a recursive function.
    c. Here, randomly number from the range is generated and the it is set as the index of the pivot element. Pivot element is chosen corresponding to the same index value.
    d. In the loop, scanning of i from left to right so long as (array[left] <pivot) and Scan j from right to left so long as(array[right]>pivot).
    e. If left<=right,we are swapping the elements.
    f. We will continue to do the same process until the left and right crosses.
    g. Once left and right cross, exchange pivot with a[right]
    h. Continue the same process recursively on the same array.

- Time complexity of Quick sort is O(nlogn) and worst case will be when all the element in the quick sort are sorted or reversely sorted and we continuously choose the lowest or highest element as the pivot. In such scenario time complexity will be O(n^2).

4. **Modified Quick Sort** :-

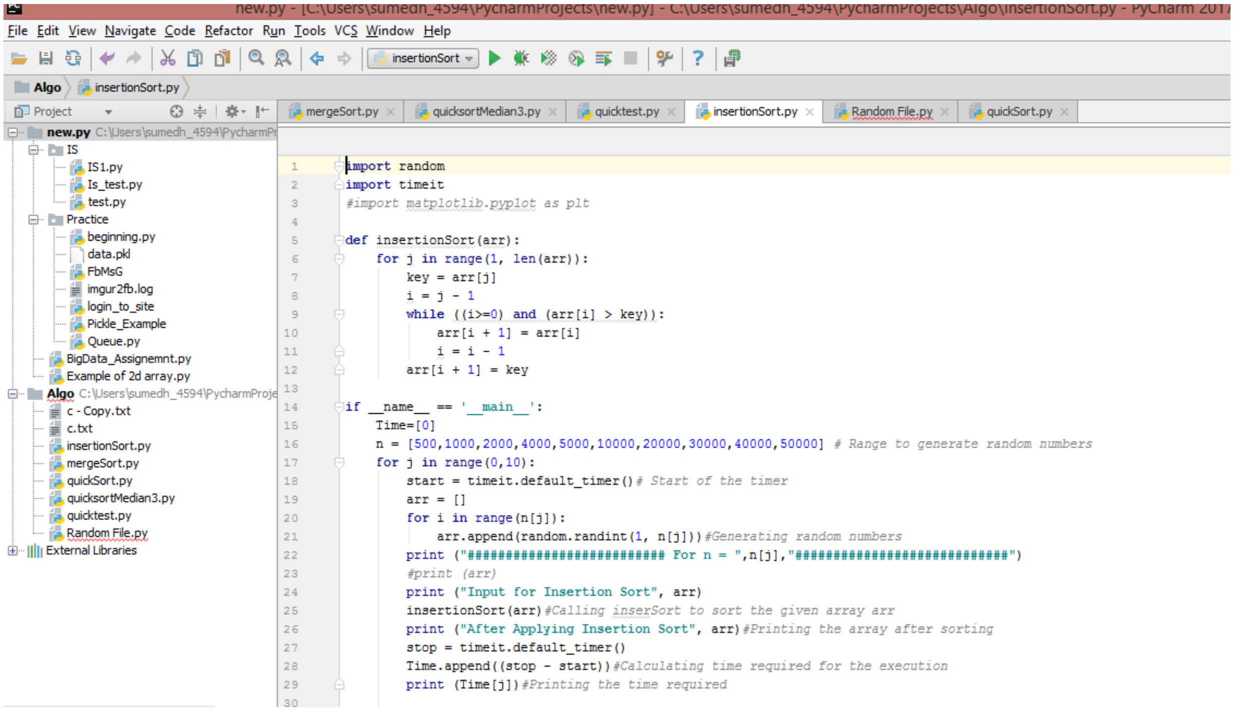
- Algorithm :-

a. Pivot selection in the modified quick sort is done by taking the median of the first, last and middle element of the array.
b. Before selecting the pivot the first, last and middle element is sorted according the value and then the array[center] is selected as the pivot. By this point we have already sorted 3 elements hence the probability of choosing bad pivot decreases.
c. We will continue to perform the normal quick sort with the pivot as median of three elements until the length of the array becomes 10.
d. Insertion sort is better option for the sorting of the elements with the length less than 10 and hence remaining sorting is performed with the help of insertion sort.

Codes and Results :-

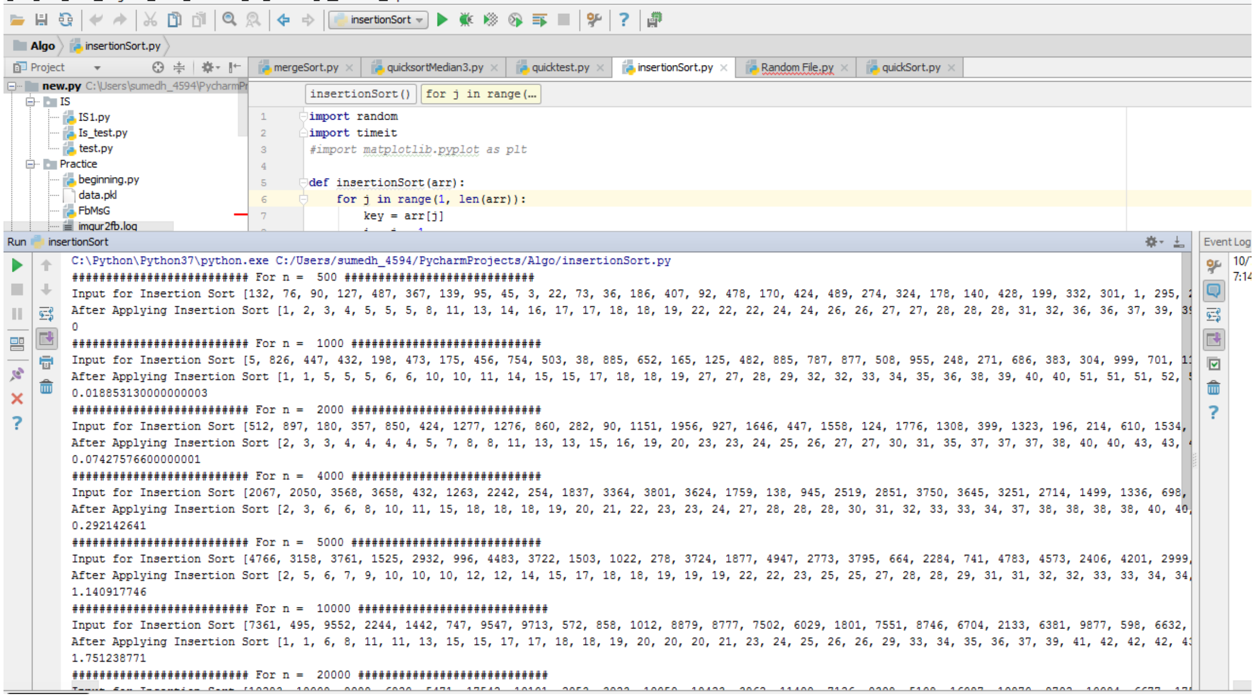
1. Insertion Sort :-

Code :-



```python
import random
import timeit
#import matplotlib.pyplot as plt

def insertionSort(arr):
    for j in range(1, len(arr)):
        key = arr[j]
        i = j - 1
        while ((i>=0) and (arr[i] > key)):
            arr[i + 1] = arr[i]
            i = i - 1
        arr[i + 1] = key

if __name__ == '__main__':
    Time=[0]
    n = [500,1000,2000,4000,5000,10000,20000,30000,40000,50000] # Range to generate random numbers
    for j in range(0,10):
        start = timeit.default_timer() # Start of the timer
        arr = []
        for i in range(n[j]):
            arr.append(random.randint(1, n[j])) #Generating random numbers
        print ("########################## For n = ",n[j],"###########################")
        #print (arr)
        print ("Input for Insertion Sort", arr)
        insertionSort(arr) #Calling inserSort to sort the given array arr
        print ("After Applying Insertion Sort", arr) #Printing the array after sorting
        stop = timeit.default_timer()
        Time.append((stop - start)) #Calculating time required for the execution
        print (Time[j]) #Printing the time required
```

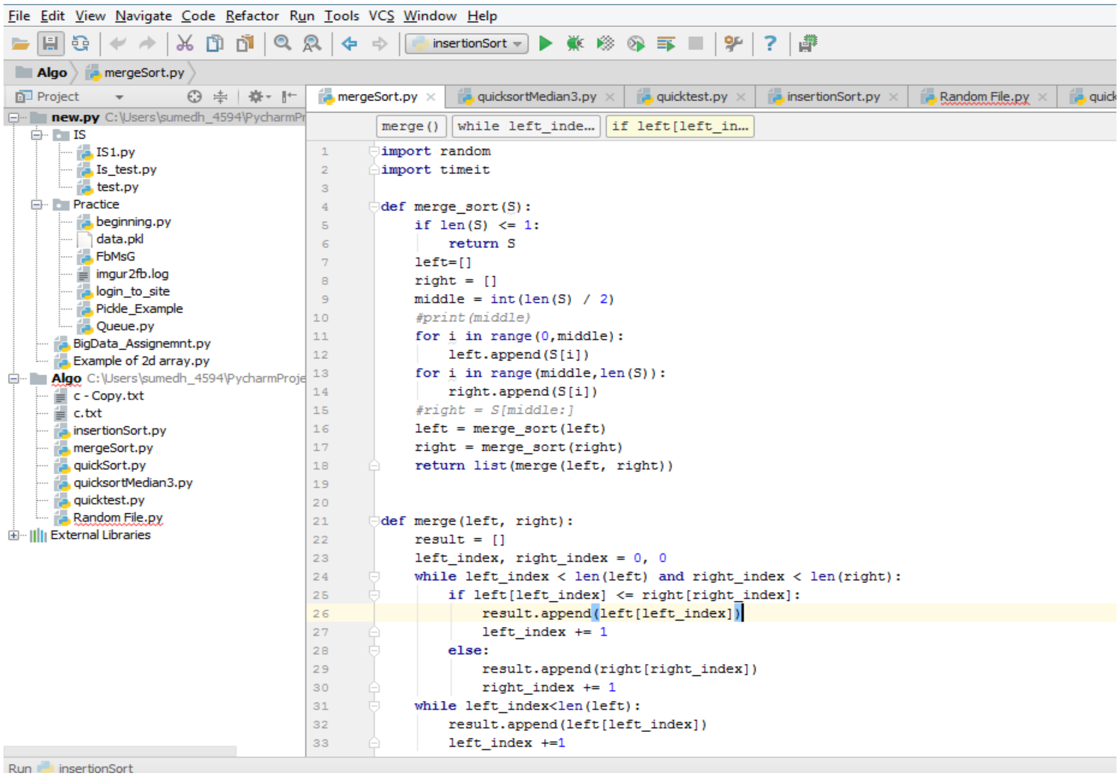
Result :-



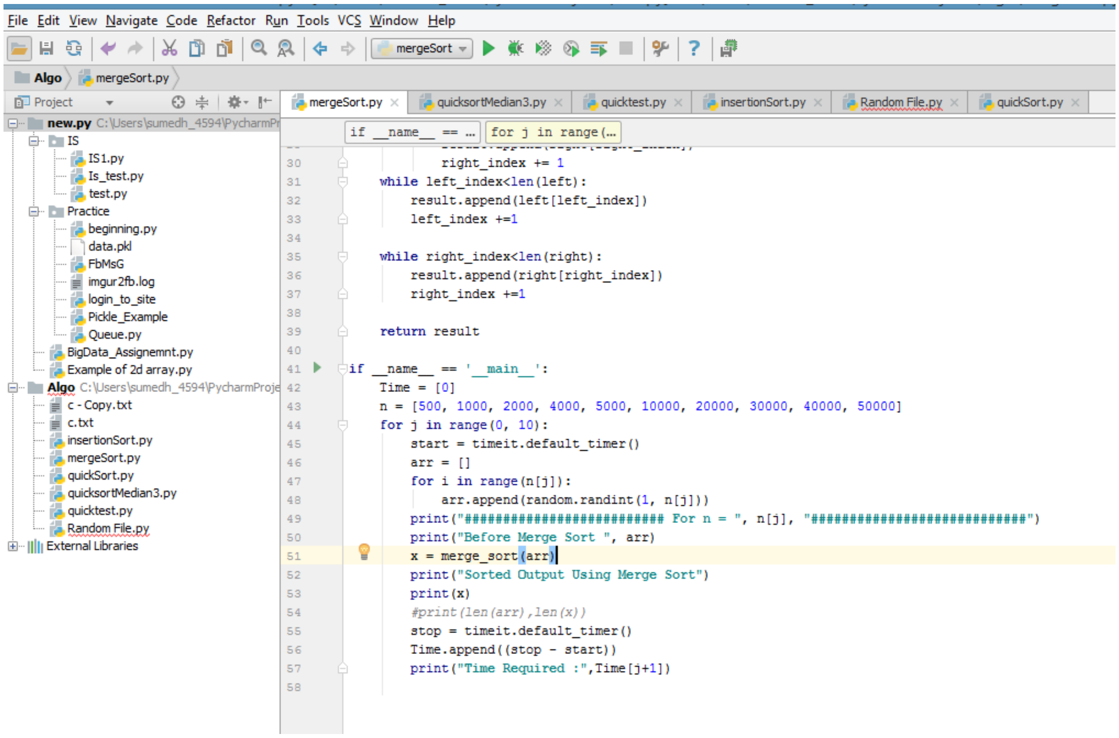
```
C:\Python\Python37\python.exe C:/Users/sumedh_4594/PycharmProjects/Algo/insertionSort.py
########################## For n =  500 ###########################
Input for Insertion Sort [132, 76, 90, 127, 487, 367, 139, 95, 45, 3, 22, 73, 36, 186, 407, 92, 478, 170, 424, 489, 274, 324, 178, 140, 428, 199, 332, 301, 1, 295,
After Applying Insertion Sort [1, 2, 3, 4, 5, 5, 5, 8, 11, 13, 14, 16, 17, 17, 18, 18, 19, 22, 22, 22, 24, 24, 26, 26, 27, 27, 28, 28, 28, 31, 32, 36, 36, 37, 39,
0
########################## For n =  1000 ###########################
Input for Insertion Sort [5, 826, 447, 432, 198, 473, 175, 456, 754, 503, 38, 885, 652, 165, 125, 482, 885, 787, 877, 508, 955, 248, 271, 686, 383, 304, 999, 701,
After Applying Insertion Sort [1, 1, 5, 5, 5, 6, 6, 10, 10, 11, 14, 15, 15, 17, 18, 18, 19, 27, 27, 28, 29, 32, 32, 33, 34, 35, 36, 38, 39, 40, 40, 51, 51, 51, 52,
0.018853130000000003
########################## For n =  2000 ###########################
Input for Insertion Sort [512, 897, 180, 357, 850, 424, 1277, 1276, 860, 282, 90, 1151, 1956, 927, 1646, 447, 1558, 124, 1776, 1308, 399, 1323, 196, 214, 610, 1534,
After Applying Insertion Sort [2, 3, 3, 4, 4, 4, 4, 5, 7, 8, 8, 11, 13, 13, 15, 16, 19, 20, 23, 23, 24, 25, 26, 27, 27, 30, 31, 35, 37, 37, 37, 38, 40, 40, 43, 43,
0.07427576600000001
########################## For n =  4000 ###########################
Input for Insertion Sort [2067, 2050, 3568, 3658, 432, 1263, 2242, 254, 1837, 3364, 3801, 3624, 1759, 138, 945, 2519, 2851, 3750, 3645, 3251, 2714, 1499, 1336, 698,
After Applying Insertion Sort [2, 3, 6, 6, 8, 10, 11, 15, 18, 18, 18, 19, 20, 21, 22, 23, 23, 24, 27, 28, 28, 28, 30, 31, 32, 33, 33, 34, 37, 38, 38, 38, 38, 40, 40,
0.292142641
########################## For n =  5000 ###########################
Input for Insertion Sort [4766, 3158, 3761, 1525, 2932, 996, 4483, 3722, 1503, 1022, 278, 3724, 1877, 4947, 2773, 3795, 664, 2284, 741, 4783, 4573, 2406, 4201, 2999,
After Applying Insertion Sort [2, 5, 6, 7, 9, 10, 10, 10, 12, 12, 14, 15, 17, 18, 18, 19, 19, 19, 22, 22, 23, 25, 25, 27, 28, 28, 29, 31, 31, 32, 32, 33, 33, 34, 34,
1.140917746
########################## For n =  10000 ###########################
Input for Insertion Sort [7361, 495, 9552, 2244, 1442, 747, 9547, 9713, 572, 858, 1012, 8879, 8777, 7502, 6029, 1801, 7551, 8746, 6704, 2133, 6381, 9877, 598, 6632,
After Applying Insertion Sort [1, 1, 6, 8, 11, 11, 13, 15, 15, 17, 17, 18, 18, 19, 20, 20, 20, 21, 23, 24, 25, 26, 26, 29, 33, 34, 35, 36, 37, 39, 41, 42, 42, 42,
1.751238771
########################## For n =  20000 ###########################
```

2. **Merge Sort**:-

Code :-

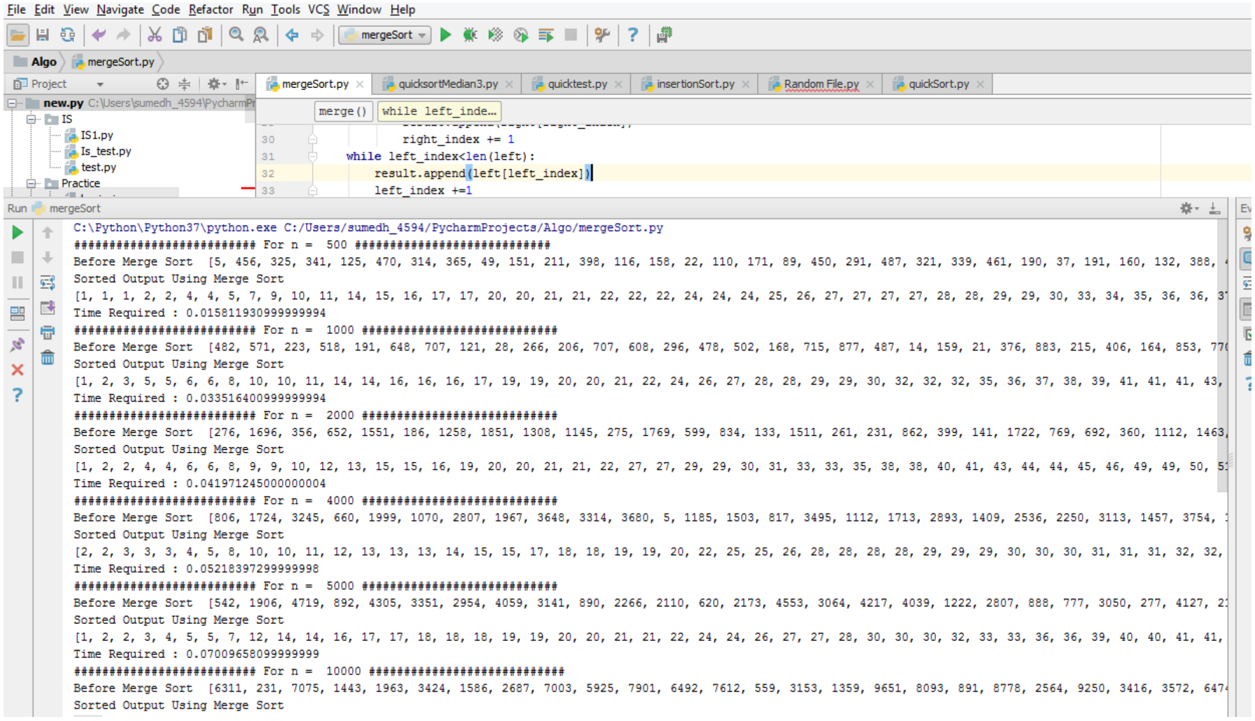


```python
import random
import timeit

def merge_sort(S):
    if len(S) <= 1:
        return S
    left=[]
    right = []
    middle = int(len(S) / 2)
    #print(middle)
    for i in range(0,middle):
        left.append(S[i])
    for i in range(middle,len(S)):
        right.append(S[i])
    #right = S[middle:]
    left = merge_sort(left)
    right = merge_sort(right)
    return list(merge(left, right))


def merge(left, right):
    result = []
    left_index, right_index = 0, 0
    while left_index < len(left) and right_index < len(right):
        if left[left_index] <= right[right_index]:
            result.append(left[left_index])
            left_index += 1
        else:
            result.append(right[right_index])
            right_index += 1
    while left_index<len(left):
        result.append(left[left_index])
        left_index +=1
```



```python
            right_index += 1
    while left_index<len(left):
        result.append(left[left_index])
        left_index +=1

    while right_index<len(right):
        result.append(right[right_index])
        right_index +=1

    return result

if __name__ == '__main__':
    Time = [0]
    n = [500, 1000, 2000, 4000, 5000, 10000, 20000, 30000, 40000, 50000]
    for j in range(0, 10):
        start = timeit.default_timer()
        arr = []
        for i in range(n[j]):
            arr.append(random.randint(1, n[j]))
        print("########################## For n = ", n[j], "############################")
        print("Before Merge Sort ", arr)
        x = merge_sort(arr)
        print("Sorted Output Using Merge Sort")
        print(x)
        #print(len(arr),len(x))
        stop = timeit.default_timer()
        Time.append((stop - start))
        print("Time Required :",Time[j+1])

```

Result :-

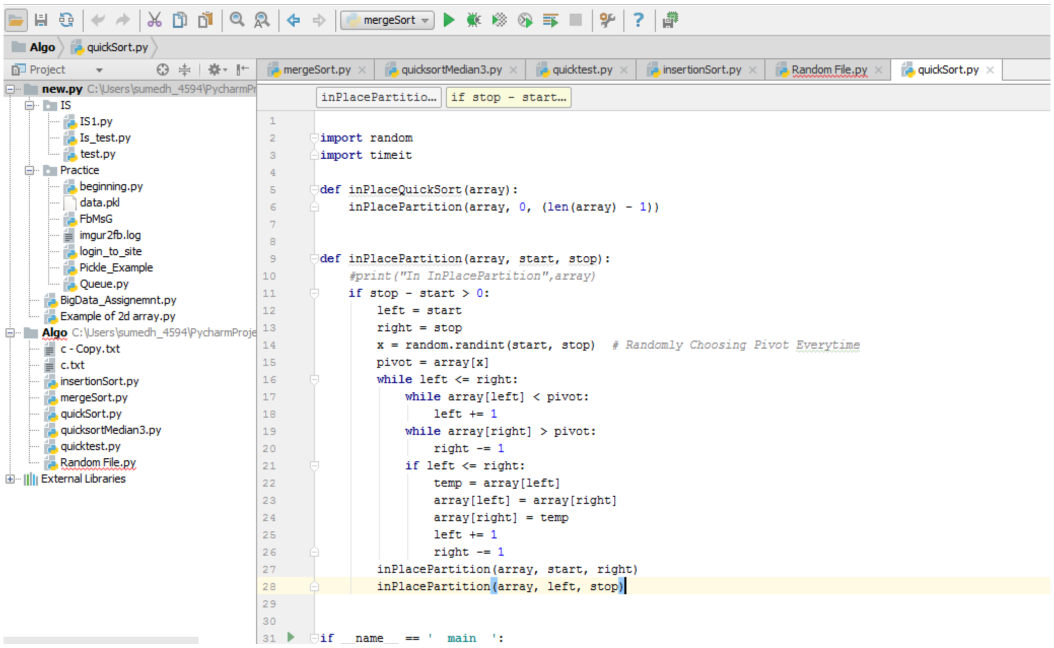


```
            right_index += 1
        while left_index<len(left):
            result.append(left[left_index])
            left_index +=1
```

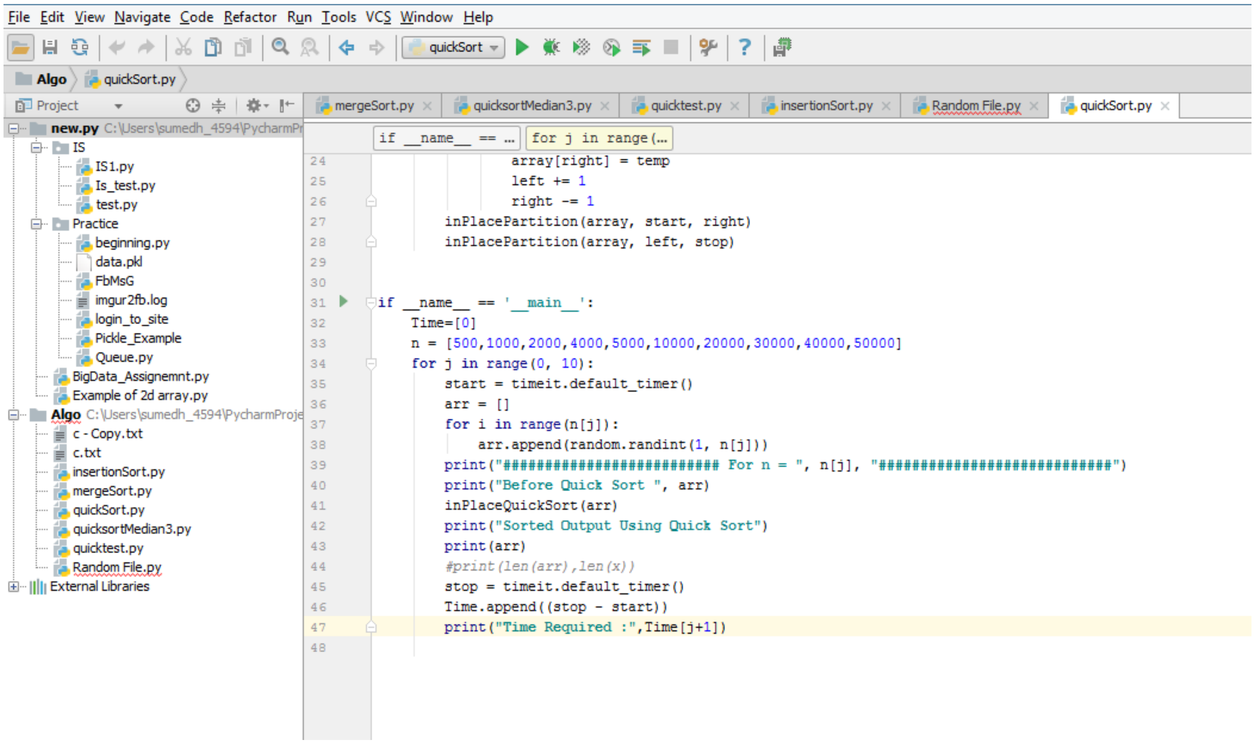
```
C:\Python\Python37\python.exe C:/Users/sumedh_4594/PycharmProjects/Algo/mergeSort.py
########################## For n =  500 ############################
Before Merge Sort  [5, 456, 325, 341, 125, 470, 314, 365, 49, 151, 211, 398, 116, 158, 22, 110, 171, 89, 450, 291, 487, 321, 339, 461, 190, 37, 191, 160, 132, 388,
Sorted Output Using Merge Sort
[1, 1, 1, 2, 2, 4, 4, 5, 7, 9, 10, 11, 14, 15, 16, 17, 17, 20, 20, 21, 21, 22, 22, 22, 24, 24, 24, 25, 26, 27, 27, 27, 27, 28, 28, 29, 29, 30, 33, 34, 35, 36, 36,
Time Required : 0.015811930999999994
########################## For n =  1000 ############################
Before Merge Sort  [482, 571, 223, 518, 191, 648, 707, 121, 28, 266, 206, 707, 608, 296, 478, 502, 168, 715, 877, 487, 14, 159, 21, 376, 883, 215, 406, 164, 853,
Sorted Output Using Merge Sort
[1, 2, 3, 5, 5, 6, 6, 8, 10, 10, 11, 14, 14, 16, 16, 16, 17, 19, 19, 20, 20, 21, 22, 24, 26, 27, 28, 28, 29, 29, 30, 32, 32, 32, 35, 36, 37, 38, 39, 41, 41, 41, 43,
Time Required : 0.033516400999999994
########################## For n =  2000 ############################
Before Merge Sort  [276, 1696, 356, 652, 1551, 186, 1258, 1851, 1308, 1145, 275, 1769, 599, 834, 133, 1511, 261, 231, 862, 399, 141, 1722, 769, 692, 360, 1112, 1463,
Sorted Output Using Merge Sort
[1, 2, 2, 4, 4, 6, 6, 8, 9, 9, 10, 12, 13, 15, 15, 16, 19, 20, 20, 21, 21, 22, 27, 27, 29, 29, 30, 31, 33, 33, 35, 38, 38, 40, 41, 43, 44, 44, 45, 46, 49, 49, 50,
Time Required : 0.041971245000000004
########################## For n =  4000 ############################
Before Merge Sort  [806, 1724, 3245, 660, 1999, 1070, 2807, 1967, 3648, 3314, 3680, 5, 1185, 1503, 817, 3495, 1112, 1713, 2893, 1409, 2536, 2250, 3113, 1457, 3754,
Sorted Output Using Merge Sort
[2, 2, 3, 3, 3, 4, 5, 8, 10, 10, 11, 12, 13, 13, 13, 14, 15, 15, 17, 18, 18, 19, 19, 20, 22, 25, 25, 26, 28, 28, 28, 28, 29, 29, 29, 30, 30, 30, 31, 31, 31, 32, 32,
Time Required : 0.05218397299999998
########################## For n =  5000 ############################
Before Merge Sort  [542, 1906, 4719, 892, 4305, 3351, 2954, 4059, 3141, 890, 2266, 2110, 620, 2173, 4553, 3064, 4217, 4039, 1222, 2807, 888, 777, 3050, 277, 4127,
Sorted Output Using Merge Sort
[1, 2, 2, 3, 4, 5, 5, 7, 12, 14, 14, 16, 17, 17, 18, 18, 18, 19, 19, 20, 20, 21, 21, 22, 24, 24, 26, 27, 27, 28, 30, 30, 30, 32, 33, 33, 36, 36, 39, 40, 40, 41, 41,
Time Required : 0.07009658099999999
########################## For n =  10000 ############################
Before Merge Sort  [6311, 231, 7075, 1443, 1963, 3424, 1586, 2687, 7003, 5925, 7901, 6492, 7612, 559, 3153, 1359, 9651, 8093, 891, 8778, 2564, 9250, 3416, 3572,
Sorted Output Using Merge Sort
```

3. **In-Place Quick Sort** :-

Code :-

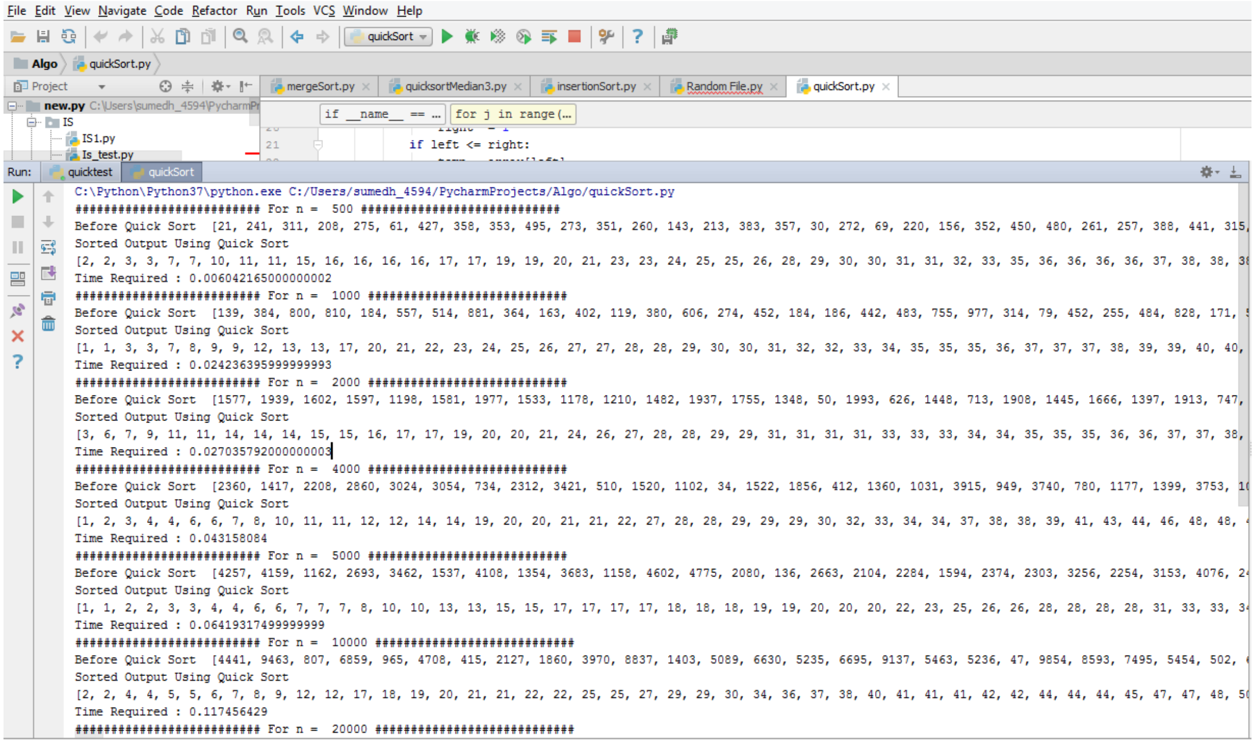


```

import random
import timeit

def inPlaceQuickSort(array):
    inPlacePartition(array, 0, (len(array) - 1))


def inPlacePartition(array, start, stop):
    #print("In InPlacePartition",array)
    if stop - start > 0:
        left = start
        right = stop
        x = random.randint(start, stop)   # Randomly Choosing Pivot Everytime
        pivot = array[x]
        while left <= right:
            while array[left] < pivot:
                left += 1
            while array[right] > pivot:
                right -= 1
            if left <= right:
                temp = array[left]
                array[left] = array[right]
                array[right] = temp
                left += 1
                right -= 1
        inPlacePartition(array, start, right)
        inPlacePartition(array, left, stop)


if __name__ == '__main__':
```



```
                array[right] = temp
                left += 1
                right -= 1
        inPlacePartition(array, start, right)
        inPlacePartition(array, left, stop)


if __name__ == '__main__':
    Time=[0]
    n = [500,1000,2000,4000,5000,10000,20000,30000,40000,50000]
    for j in range(0, 10):
        start = timeit.default_timer()
        arr = []
        for i in range(n[j]):
            arr.append(random.randint(1, n[j]))
        print("######################### For n = ", n[j], "############################")
        print("Before Quick Sort ", arr)
        inPlaceQuickSort(arr)
        print("Sorted Output Using Quick Sort")
        print(arr)
        #print(len(arr),len(x))
        stop = timeit.default_timer()
        Time.append((stop - start))
        print("Time Required :",Time[j+1])
```

Result :-

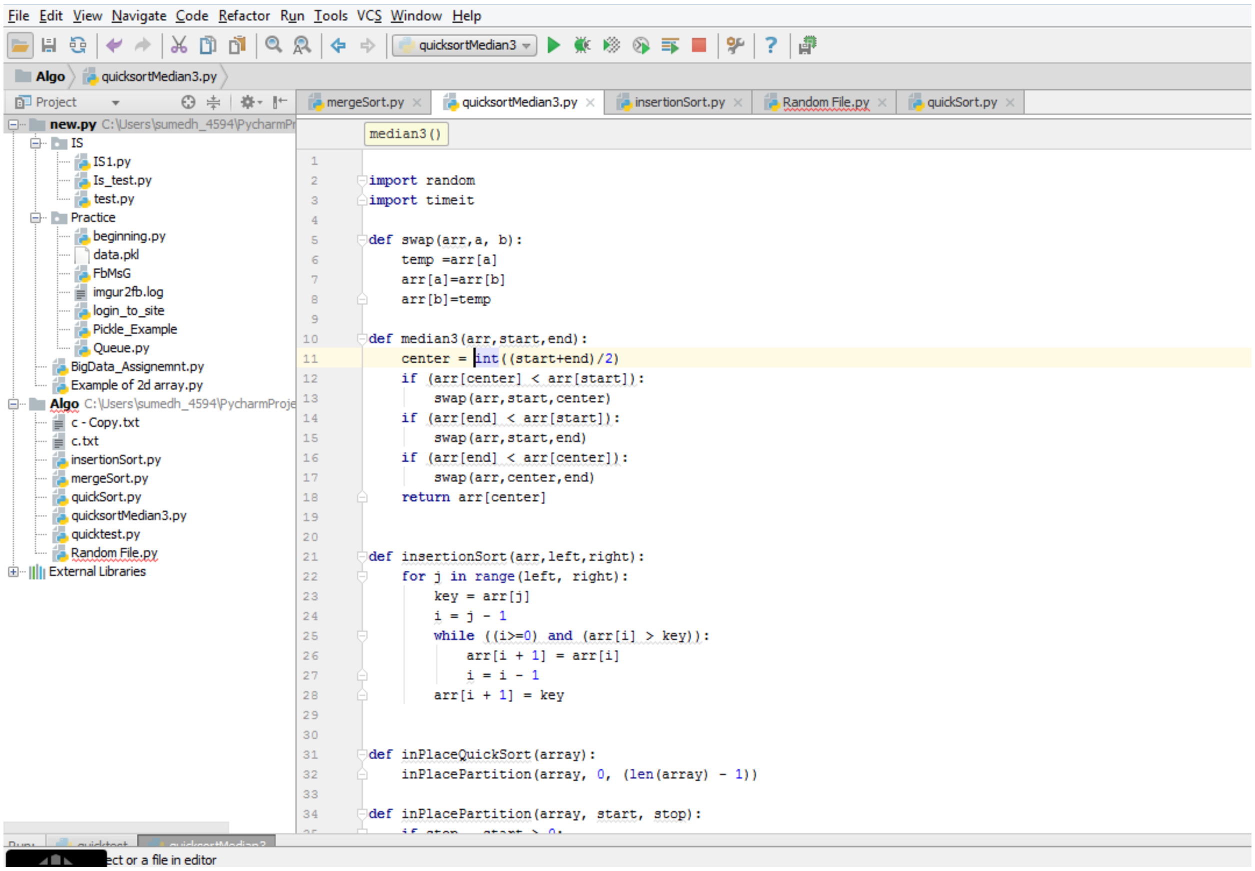


```
        if left <= right:
```

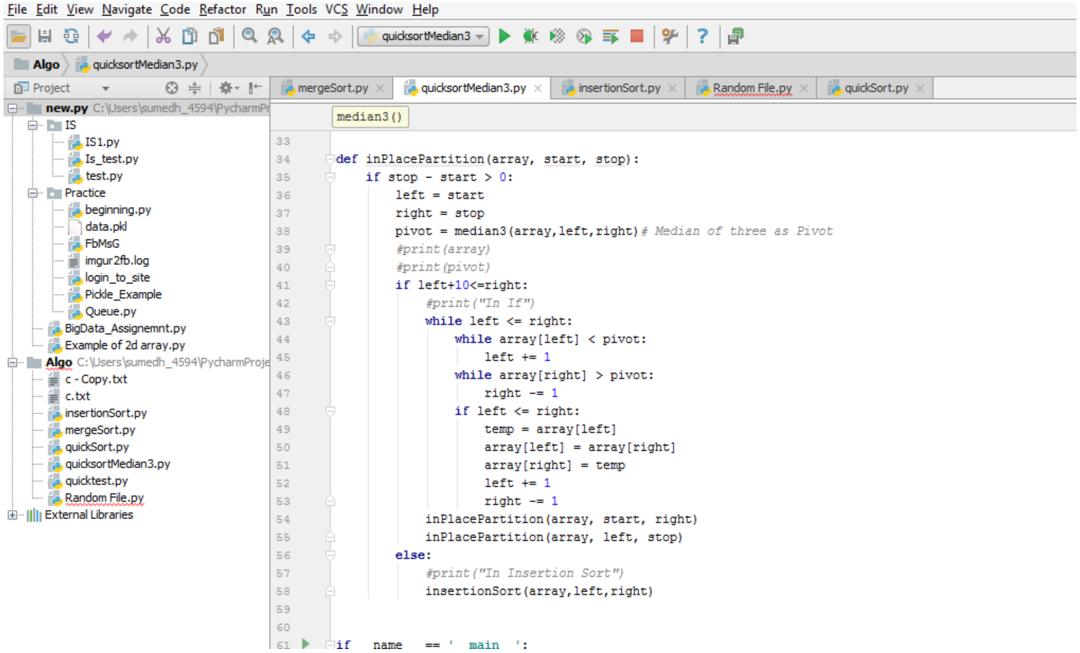
```
C:\Python\Python37\python.exe C:/Users/sumedh_4594/PycharmProjects/Algo/quickSort.py
######################### For n =  500 ############################
Before Quick Sort  [21, 241, 311, 208, 275, 61, 427, 358, 353, 495, 273, 351, 260, 143, 213, 383, 357, 30, 272, 69, 220, 156, 352, 450, 480, 261, 257, 388, 441, 315,
Sorted Output Using Quick Sort
[2, 2, 3, 3, 7, 7, 10, 11, 11, 15, 16, 16, 16, 16, 17, 17, 19, 19, 20, 21, 23, 23, 24, 25, 25, 26, 28, 29, 30, 30, 31, 31, 32, 33, 35, 36, 36, 36, 36, 37, 38, 38, 3
Time Required : 0.006042165000000002
######################### For n =  1000 ############################
Before Quick Sort  [139, 384, 800, 810, 184, 557, 514, 881, 364, 163, 402, 119, 380, 606, 274, 452, 184, 186, 442, 483, 755, 977, 314, 79, 452, 255, 484, 828, 171,
Sorted Output Using Quick Sort
[1, 1, 3, 3, 7, 8, 9, 9, 12, 13, 13, 17, 20, 21, 22, 23, 24, 25, 26, 27, 27, 28, 28, 29, 30, 30, 31, 32, 32, 33, 34, 35, 35, 35, 36, 37, 37, 37, 38, 39, 39, 40, 40,
Time Required : 0.024236395999999993
######################### For n =  2000 ############################
Before Quick Sort  [1577, 1939, 1602, 1597, 1198, 1581, 1977, 1533, 1178, 1210, 1482, 1937, 1755, 1348, 50, 1993, 626, 1448, 713, 1908, 1445, 1666, 1397, 1913, 747,
Sorted Output Using Quick Sort
[3, 6, 7, 9, 11, 11, 14, 14, 14, 15, 15, 16, 17, 17, 19, 20, 20, 21, 24, 26, 27, 28, 28, 29, 29, 31, 31, 31, 31, 33, 33, 33, 34, 34, 35, 35, 35, 36, 36, 37, 37, 38,
Time Required : 0.027035792000000003
######################### For n =  4000 ############################
Before Quick Sort  [2360, 1417, 2208, 2860, 3024, 3054, 734, 2312, 3421, 510, 1520, 1102, 34, 1522, 1856, 412, 1360, 1031, 3915, 949, 3740, 780, 1177, 1399, 3753, 1
Sorted Output Using Quick Sort
[1, 2, 3, 4, 4, 6, 6, 7, 8, 10, 11, 11, 12, 12, 14, 14, 19, 20, 20, 21, 21, 22, 27, 28, 28, 29, 29, 29, 30, 32, 33, 34, 34, 37, 38, 38, 39, 41, 43, 44, 46, 48, 48,
Time Required : 0.043158084
######################### For n =  5000 ############################
Before Quick Sort  [4257, 4159, 1162, 2693, 3462, 1537, 4108, 1354, 3683, 1158, 4602, 4775, 2080, 136, 2663, 2104, 2284, 1594, 2374, 2303, 3256, 2254, 3153, 4076, 2
Sorted Output Using Quick Sort
[1, 1, 2, 2, 3, 3, 4, 4, 6, 6, 7, 7, 7, 8, 10, 10, 13, 13, 15, 15, 17, 17, 17, 17, 18, 18, 18, 19, 19, 20, 20, 20, 22, 23, 25, 26, 26, 28, 28, 28, 28, 31, 33, 33, 3
Time Required : 0.06419317499999999
######################### For n =  10000 ############################
Before Quick Sort  [4441, 9463, 807, 6859, 965, 4708, 415, 2127, 1860, 3970, 8837, 1403, 5089, 6630, 5235, 6695, 9137, 5463, 5236, 47, 9854, 8593, 7495, 5454, 502,
Sorted Output Using Quick Sort
[2, 2, 4, 4, 5, 5, 6, 7, 8, 9, 12, 12, 17, 18, 19, 20, 21, 21, 22, 22, 25, 25, 27, 29, 29, 30, 34, 36, 37, 38, 40, 41, 41, 41, 42, 42, 44, 44, 44, 45, 47, 47, 48, 5
Time Required : 0.117456429
######################### For n =  20000 ############################
```

4. **Modified Quick Sort**:-

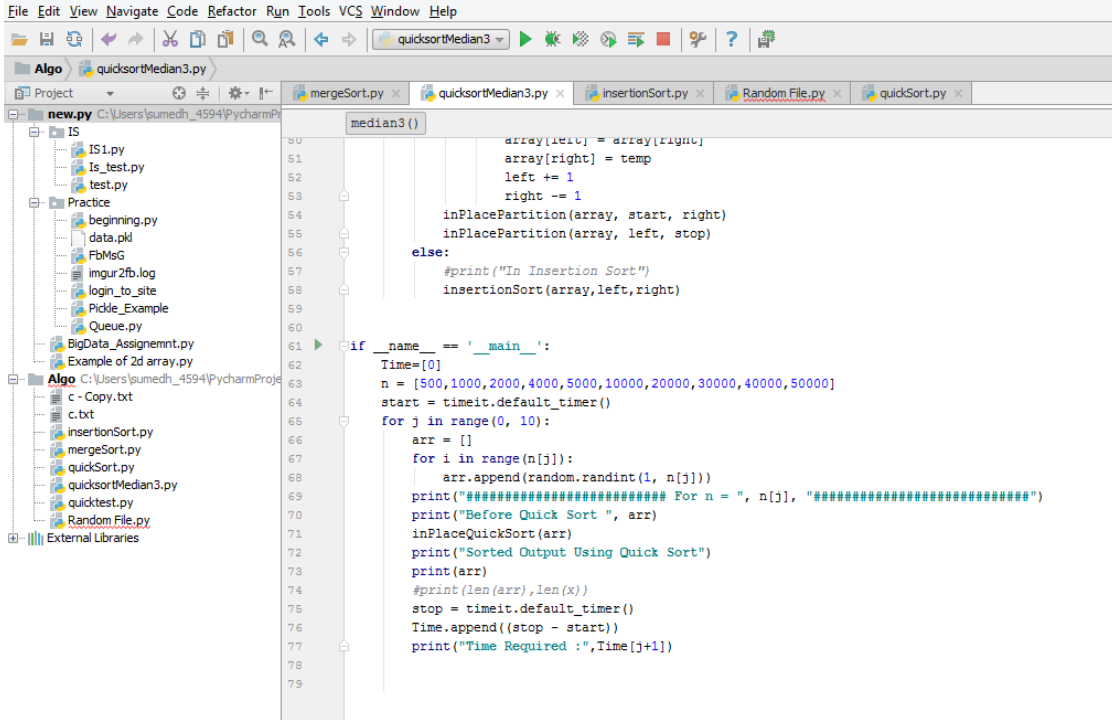
Code :-



```

import random
import timeit

def swap(arr,a, b):
    temp =arr[a]
    arr[a]=arr[b]
    arr[b]=temp

def median3(arr,start,end):
    center = int((start+end)/2)
    if (arr[center] < arr[start]):
        swap(arr,start,center)
    if (arr[end] < arr[start]):
        swap(arr,start,end)
    if (arr[end] < arr[center]):
        swap(arr,center,end)
    return arr[center]


def insertionSort(arr,left,right):
    for j in range(left, right):
        key = arr[j]
        i = j - 1
        while ((i>=0) and (arr[i] > key)):
            arr[i + 1] = arr[i]
            i = i - 1
        arr[i + 1] = key


def inPlaceQuickSort(array):
    inPlacePartition(array, 0, (len(array) - 1))

def inPlacePartition(array, start, stop):
```

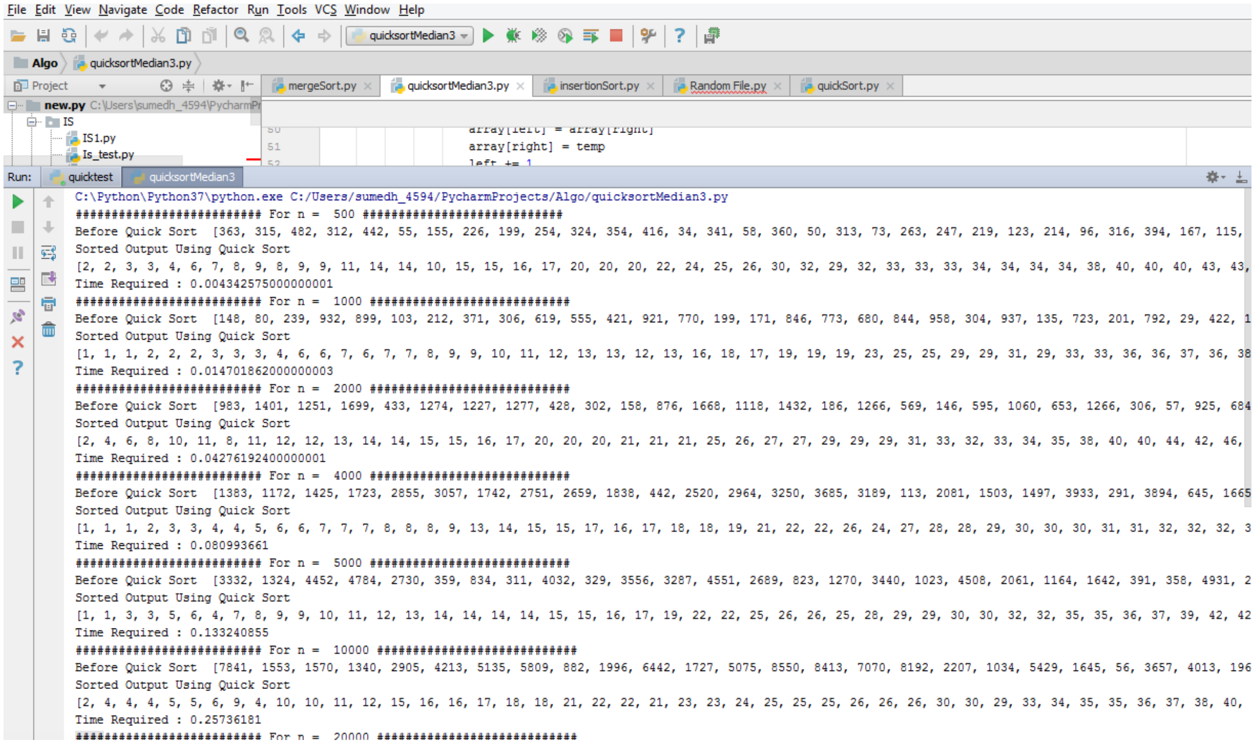


```

def inPlacePartition(array, start, stop):
    if stop - start > 0:
        left = start
        right = stop
        pivot = median3(array,left,right)# Median of three as Pivot
        #print(array)
        #print(pivot)
        if left+10<=right:
            #print("In If")
            while left <= right:
                while array[left] < pivot:
                    left += 1
                while array[right] > pivot:
                    right -= 1
                if left <= right:
                    temp = array[left]
                    array[left] = array[right]
                    array[right] = temp
                    left += 1
                    right -= 1
            inPlacePartition(array, start, right)
            inPlacePartition(array, left, stop)
        else:
            #print("In Insertion Sort")
            insertionSort(array,left,right)


if __name__ == '__main__':
```



```
                    array[left] = array[right]
                    array[right] = temp
                    left += 1
                    right -= 1
            inPlacePartition(array, start, right)
            inPlacePartition(array, left, stop)
        else:
            #print("In Insertion Sort")
            insertionSort(array,left,right)


if __name__ == '__main__':
    Time=[0]
    n = [500,1000,2000,4000,5000,10000,20000,30000,40000,50000]
    start = timeit.default_timer()
    for j in range(0, 10):
        arr = []
        for i in range(n[j]):
            arr.append(random.randint(1, n[j]))
        print("########################## For n = ", n[j], "############################")
        print("Before Quick Sort ", arr)
        inPlaceQuickSort(arr)
        print("Sorted Output Using Quick Sort")
        print(arr)
        #print(len(arr),len(x))
        stop = timeit.default_timer()
        Time.append((stop - start))
        print("Time Required :",Time[j+1])
```

Result:-

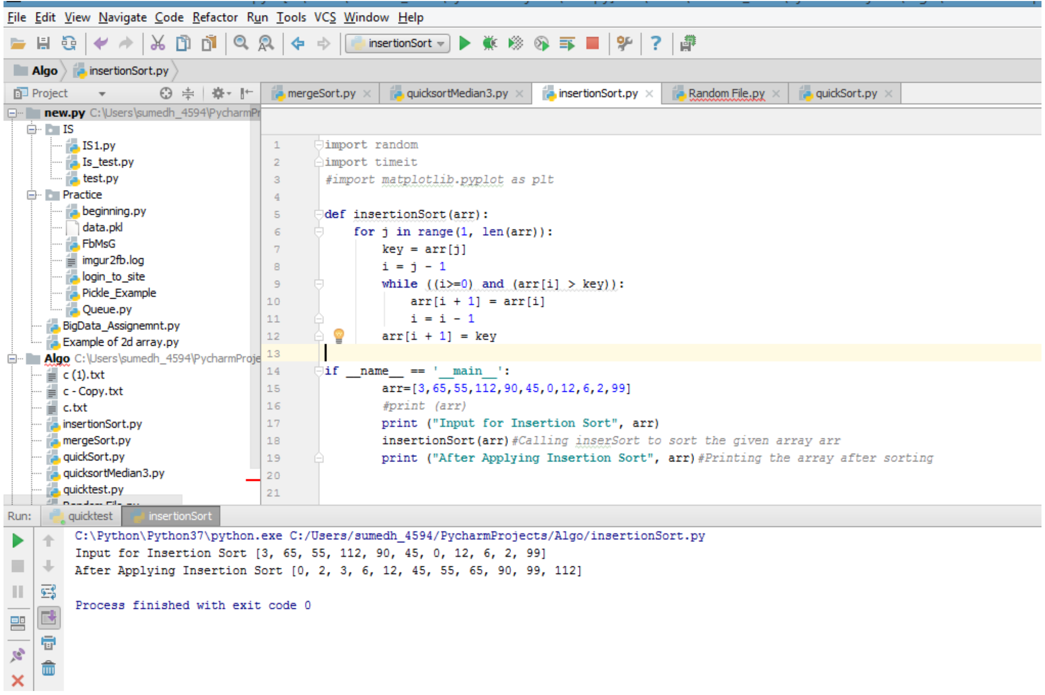


```
C:\Python\Python37\python.exe C:/Users/sumedh_4594/PycharmProjects/Algo/quicksortMedian3.py
########################## For n =  500 ############################
Before Quick Sort  [363, 315, 482, 312, 442, 55, 155, 226, 199, 254, 324, 354, 416, 34, 341, 58, 360, 50, 313, 73, 263, 247, 219, 123, 214, 96, 316, 394, 167, 115,
Sorted Output Using Quick Sort
[2, 2, 3, 3, 4, 6, 7, 8, 9, 8, 9, 9, 11, 14, 14, 10, 15, 15, 16, 17, 20, 20, 20, 22, 24, 25, 26, 30, 32, 29, 32, 33, 33, 33, 34, 34, 34, 34, 38, 40, 40, 40, 43, 43,
Time Required : 0.004342575000000001
########################## For n =  1000 ############################
Before Quick Sort  [148, 80, 239, 932, 899, 103, 212, 371, 306, 619, 555, 421, 921, 770, 199, 171, 846, 773, 680, 844, 958, 304, 937, 135, 723, 201, 792, 29, 422, 1
Sorted Output Using Quick Sort
[1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 6, 6, 7, 6, 7, 7, 8, 9, 9, 10, 11, 12, 13, 13, 12, 13, 16, 18, 17, 19, 19, 19, 23, 25, 25, 29, 29, 31, 29, 33, 33, 36, 36, 37, 36, 38
Time Required : 0.014701862000000003
########################## For n =  2000 ############################
Before Quick Sort  [983, 1401, 1251, 1699, 433, 1274, 1227, 1277, 428, 302, 158, 876, 1668, 1118, 1432, 186, 1266, 569, 146, 595, 1060, 653, 1266, 306, 57, 925, 684
Sorted Output Using Quick Sort
[2, 4, 6, 8, 10, 11, 8, 11, 12, 12, 13, 14, 14, 15, 15, 16, 17, 20, 20, 20, 21, 21, 21, 25, 26, 27, 27, 29, 29, 29, 31, 33, 32, 33, 34, 35, 38, 40, 40, 44, 42, 46,
Time Required : 0.04276192400000001
########################## For n =  4000 ############################
Before Quick Sort  [1383, 1172, 1425, 1723, 2855, 3057, 1742, 2751, 2659, 1838, 442, 2520, 2964, 3250, 3685, 3189, 113, 2081, 1503, 1497, 3933, 291, 3894, 645, 1665
Sorted Output Using Quick Sort
[1, 1, 1, 2, 3, 3, 4, 4, 5, 6, 6, 7, 7, 7, 8, 8, 8, 9, 13, 14, 15, 15, 17, 16, 17, 18, 18, 19, 21, 22, 22, 26, 24, 27, 28, 28, 29, 30, 30, 30, 31, 31, 32, 32, 32, 3
Time Required : 0.080993661
########################## For n =  5000 ############################
Before Quick Sort  [3332, 1324, 4452, 4784, 2730, 359, 834, 311, 4032, 329, 3556, 3287, 4551, 2689, 823, 1270, 3440, 1023, 4508, 2061, 1164, 1642, 391, 358, 4931, 2
Sorted Output Using Quick Sort
[1, 1, 3, 3, 5, 6, 4, 7, 8, 9, 9, 10, 11, 12, 13, 14, 14, 14, 14, 15, 15, 16, 17, 19, 22, 22, 25, 26, 26, 25, 28, 29, 29, 30, 30, 32, 32, 35, 35, 36, 37, 39, 42, 42
Time Required : 0.133240855
########################## For n =  10000 ############################
Before Quick Sort  [7841, 1553, 1570, 1340, 2905, 4213, 5135, 5809, 882, 1996, 6442, 1727, 5075, 8550, 8413, 7070, 8192, 2207, 1034, 5429, 1645, 56, 3657, 4013, 196
Sorted Output Using Quick Sort
[2, 4, 4, 4, 5, 5, 6, 9, 4, 10, 10, 11, 12, 15, 16, 16, 17, 18, 18, 21, 22, 22, 21, 23, 23, 24, 25, 25, 25, 26, 26, 26, 30, 30, 29, 33, 34, 35, 35, 36, 37, 38, 40,
Time Required : 0.25736181
########################## For n =  20000 ############################
```

Different Outputs :-

Insertion Sort

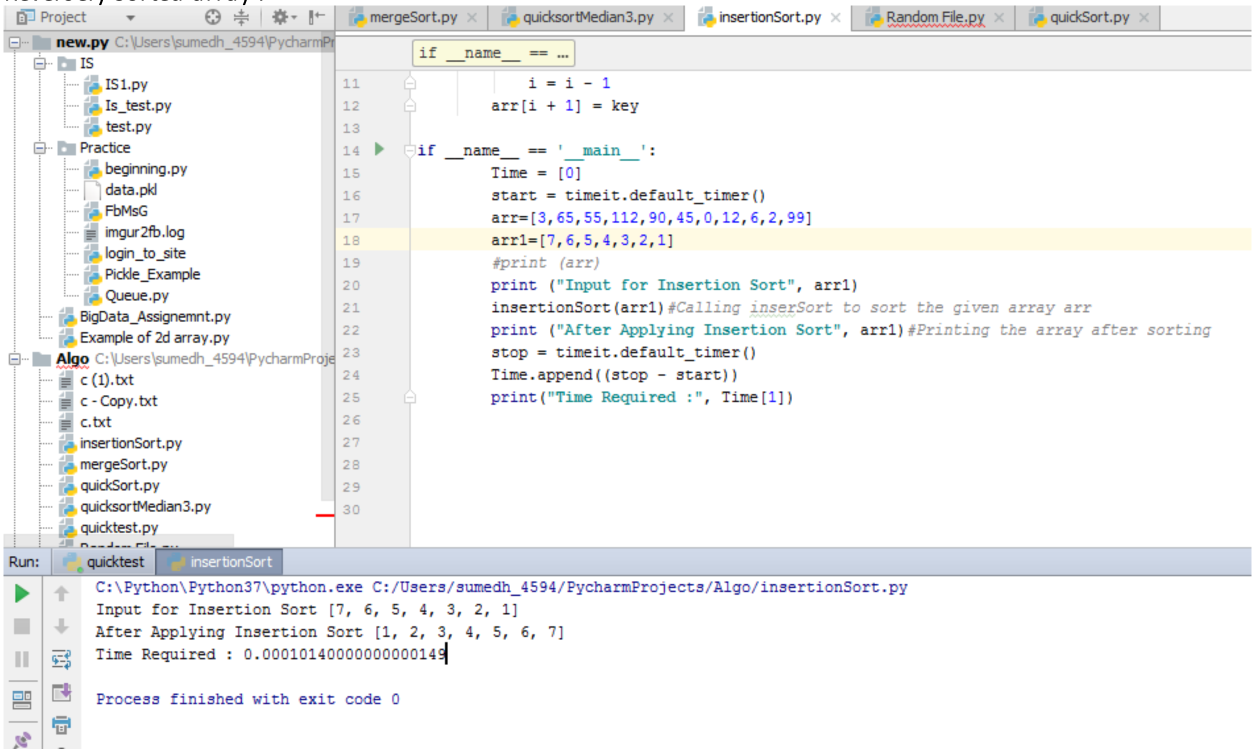
a. Random array :-



```
import random
import timeit
#import matplotlib.pyplot as plt

def insertionSort(arr):
    for j in range(1, len(arr)):
        key = arr[j]
        i = j - 1
        while ((i>=0) and (arr[i] > key)):
            arr[i + 1] = arr[i]
            i = i - 1
        arr[i + 1] = key

if __name__ == '__main__':
        arr=[3,65,55,112,90,45,0,12,6,2,99]
        #print (arr)
        print ("Input for Insertion Sort", arr)
        insertionSort(arr)#Calling inserSort to sort the given array arr
        print ("After Applying Insertion Sort", arr)#Printing the array after sorting
```

```
C:\Python\Python37\python.exe C:/Users/sumedh_4594/PycharmProjects/Algo/insertionSort.py
Input for Insertion Sort [3, 65, 55, 112, 90, 45, 0, 12, 6, 2, 99]
After Applying Insertion Sort [0, 2, 3, 6, 12, 45, 55, 65, 90, 99, 112]

Process finished with exit code 0
```

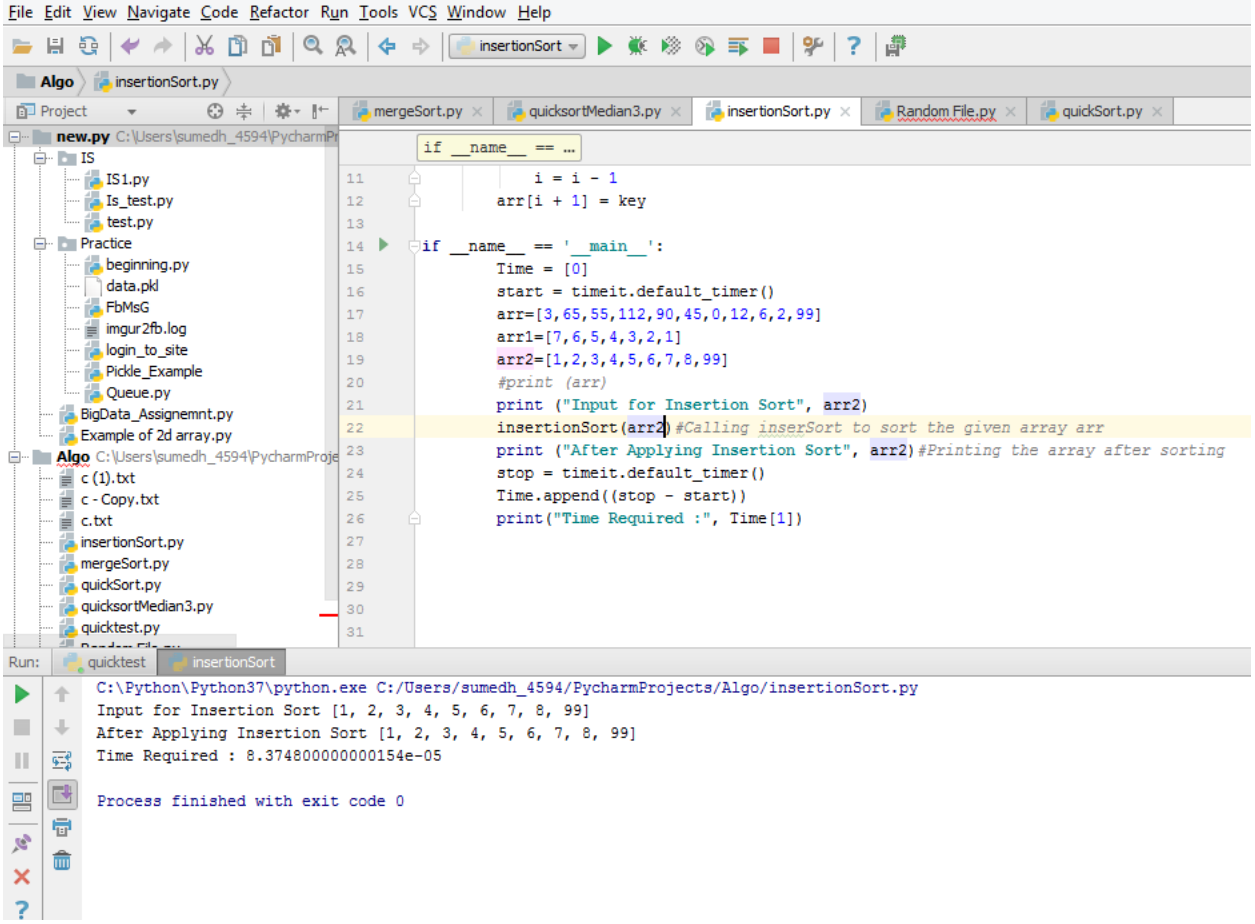
b. Reversely Sorted array :-



```
            i = i - 1
        arr[i + 1] = key

if __name__ == '__main__':
        Time = [0]
        start = timeit.default_timer()
        arr=[3,65,55,112,90,45,0,12,6,2,99]
        arr1=[7,6,5,4,3,2,1]
        #print (arr)
        print ("Input for Insertion Sort", arr1)
        insertionSort(arr1)#Calling inserSort to sort the given array arr
        print ("After Applying Insertion Sort", arr1)#Printing the array after sorting
        stop = timeit.default_timer()
        Time.append((stop - start))
        print("Time Required :", Time[1])
```

```
C:\Python\Python37\python.exe C:/Users/sumedh_4594/PycharmProjects/Algo/insertionSort.py
Input for Insertion Sort [7, 6, 5, 4, 3, 2, 1]
After Applying Insertion Sort [1, 2, 3, 4, 5, 6, 7]
Time Required : 0.00010140000000000149

Process finished with exit code 0
```

c. Sorted Array :



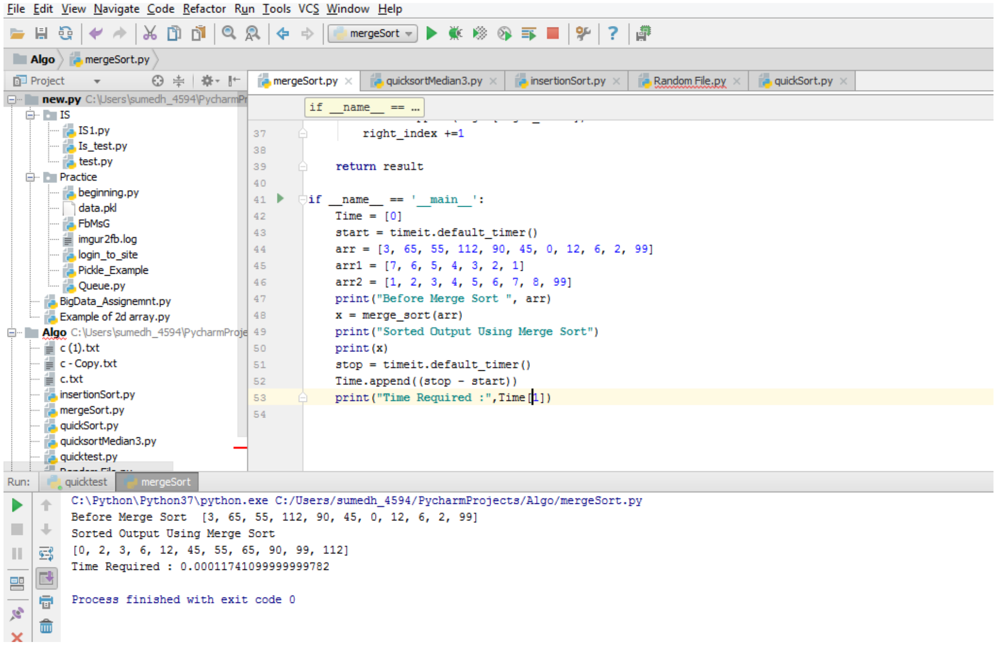
```
            i = i - 1
        arr[i + 1] = key

if __name__ == '__main__':
        Time = [0]
        start = timeit.default_timer()
        arr=[3,65,55,112,90,45,0,12,6,2,99]
        arr1=[7,6,5,4,3,2,1]
        arr2=[1,2,3,4,5,6,7,8,99]
        #print (arr)
        print ("Input for Insertion Sort", arr2)
        insertionSort(arr2)#Calling inserSort to sort the given array arr
        print ("After Applying Insertion Sort", arr2)#Printing the array after sorting
        stop = timeit.default_timer()
        Time.append((stop - start))
        print("Time Required :", Time[1])
```

```
C:\Python\Python37\python.exe C:/Users/sumedh_4594/PycharmProjects/Algo/insertionSort.py
Input for Insertion Sort [1, 2, 3, 4, 5, 6, 7, 8, 99]
After Applying Insertion Sort [1, 2, 3, 4, 5, 6, 7, 8, 99]
Time Required : 8.374800000000154e-05

Process finished with exit code 0
```

- Hence it can be conclude that the execution of the insertion sort is very fast for the sorted array as it does not go into the inner while loop of the code and as a result the time complexity of the algorithm reduces to O(n).

Merge Sort:-

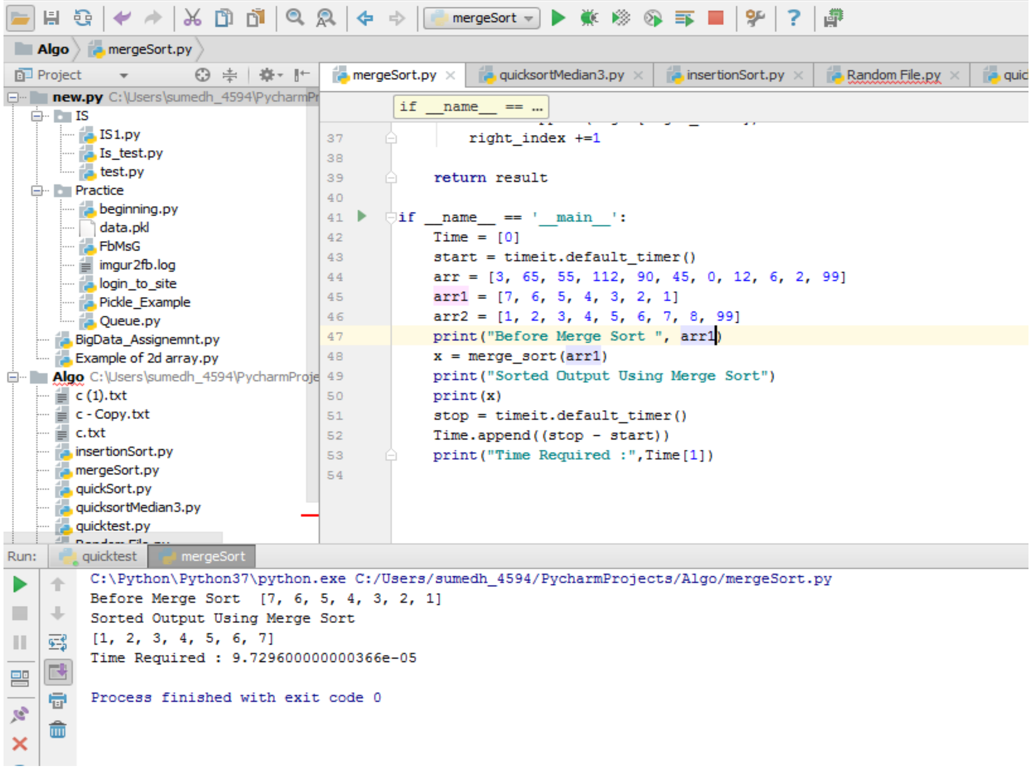
a. Random Input array:



```
        right_index +=1

    return result

if __name__ == '__main__':
    Time = [0]
    start = timeit.default_timer()
    arr = [3, 65, 55, 112, 90, 45, 0, 12, 6, 2, 99]
    arr1 = [7, 6, 5, 4, 3, 2, 1]
    arr2 = [1, 2, 3, 4, 5, 6, 7, 8, 99]
    print("Before Merge Sort ", arr)
    x = merge_sort(arr)
    print("Sorted Output Using Merge Sort")
    print(x)
    stop = timeit.default_timer()
    Time.append((stop - start))
    print("Time Required :",Time[1])
```

```
C:\Python\Python37\python.exe C:/Users/sumedh_4594/PycharmProjects/Algo/mergeSort.py
Before Merge Sort  [3, 65, 55, 112, 90, 45, 0, 12, 6, 2, 99]
Sorted Output Using Merge Sort
[0, 2, 3, 6, 12, 45, 55, 65, 90, 99, 112]
Time Required : 0.00011741099999999782

Process finished with exit code 0
```

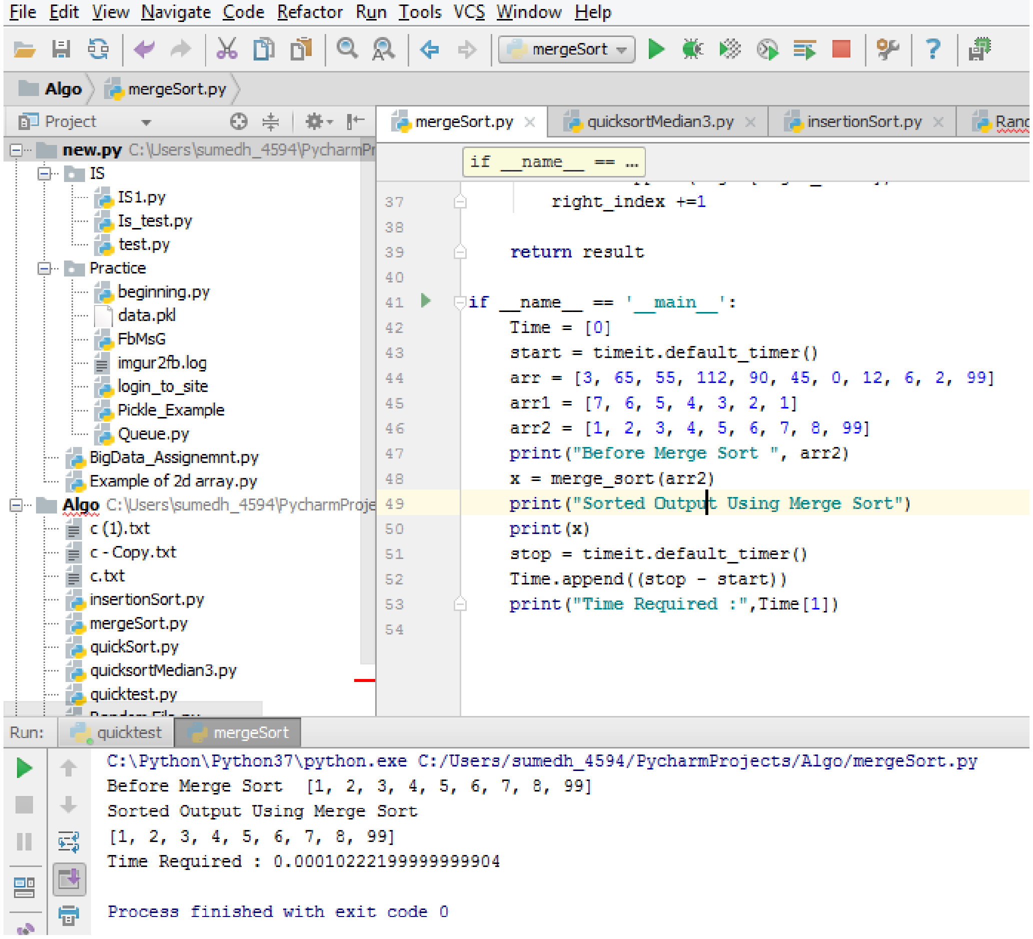
b. Reversely Sorted Array:



```
        right_index +=1

    return result

if __name__ == '__main__':
    Time = [0]
    start = timeit.default_timer()
    arr = [3, 65, 55, 112, 90, 45, 0, 12, 6, 2, 99]
    arr1 = [7, 6, 5, 4, 3, 2, 1]
    arr2 = [1, 2, 3, 4, 5, 6, 7, 8, 99]
    print("Before Merge Sort ", arr1)
    x = merge_sort(arr1)
    print("Sorted Output Using Merge Sort")
    print(x)
    stop = timeit.default_timer()
    Time.append((stop - start))
    print("Time Required :",Time[1])
```

```
C:\Python\Python37\python.exe C:/Users/sumedh_4594/PycharmProjects/Algo/mergeSort.py
Before Merge Sort  [7, 6, 5, 4, 3, 2, 1]
Sorted Output Using Merge Sort
[1, 2, 3, 4, 5, 6, 7]
Time Required : 9.729600000000366e-05

Process finished with exit code 0
```

c. Sorted Array :



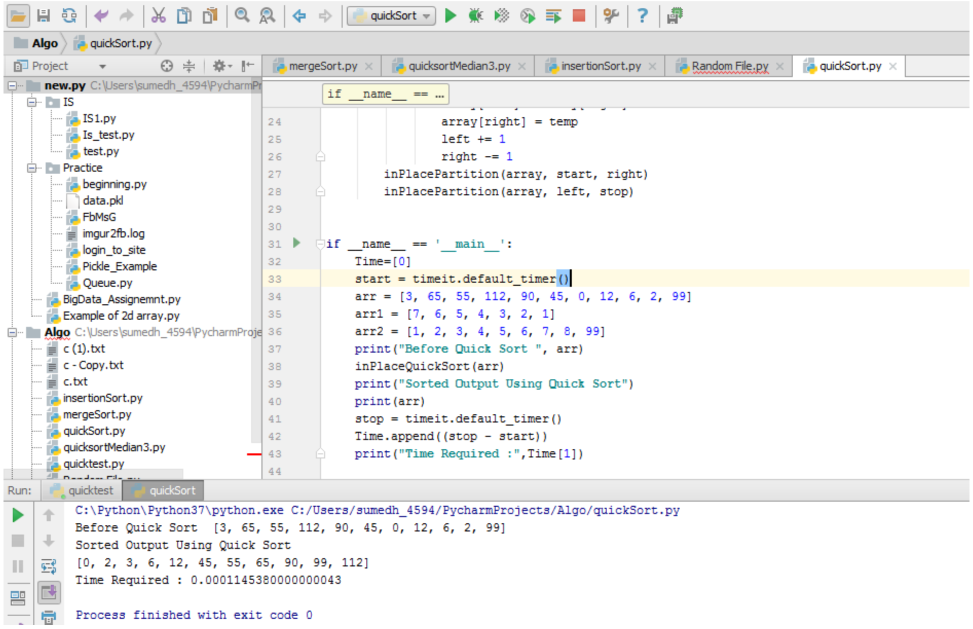
```
        right_index +=1

    return result

if __name__ == '__main__':
    Time = [0]
    start = timeit.default_timer()
    arr = [3, 65, 55, 112, 90, 45, 0, 12, 6, 2, 99]
    arr1 = [7, 6, 5, 4, 3, 2, 1]
    arr2 = [1, 2, 3, 4, 5, 6, 7, 8, 99]
    print("Before Merge Sort ", arr2)
    x = merge_sort(arr2)
    print("Sorted Output Using Merge Sort")
    print(x)
    stop = timeit.default_timer()
    Time.append((stop - start))
    print("Time Required :",Time[1])
```

```
C:\Python\Python37\python.exe C:/Users/sumedh_4594/PycharmProjects/Algo/mergeSort.py
Before Merge Sort  [1, 2, 3, 4, 5, 6, 7, 8, 99]
Sorted Output Using Merge Sort
[1, 2, 3, 4, 5, 6, 7, 8, 99]
Time Required : 0.00010222199999999904

Process finished with exit code 0
```

Merge sort is invariant of the value of the elements in an array. It will take perform same amount of work in order to sort the given array.

Quick Sort:

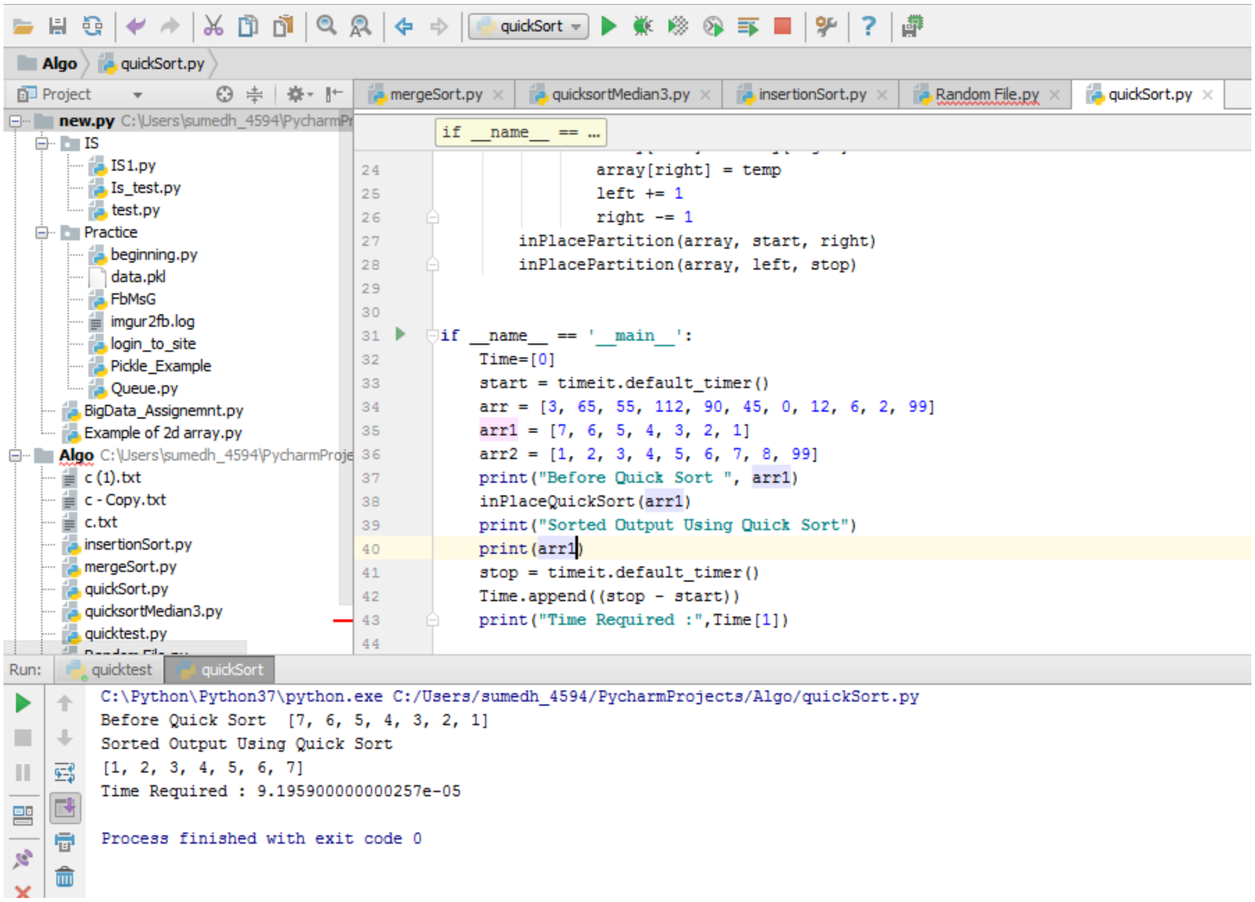
a. Random Input:-



```
                array[right] = temp
                left += 1
                right -= 1
        inPlacePartition(array, start, right)
        inPlacePartition(array, left, stop)


if __name__ == '__main__':
    Time=[0]
    start = timeit.default_timer()
    arr = [3, 65, 55, 112, 90, 45, 0, 12, 6, 2, 99]
    arr1 = [7, 6, 5, 4, 3, 2, 1]
    arr2 = [1, 2, 3, 4, 5, 6, 7, 8, 99]
    print("Before Quick Sort ", arr)
    inPlaceQuickSort(arr)
    print("Sorted Output Using Quick Sort")
    print(arr)
    stop = timeit.default_timer()
    Time.append((stop - start))
    print("Time Required :",Time[1])
```

```
C:\Python\Python37\python.exe C:/Users/sumedh_4594/PycharmProjects/Algo/quickSort.py
Before Quick Sort  [3, 65, 55, 112, 90, 45, 0, 12, 6, 2, 99]
Sorted Output Using Quick Sort
[0, 2, 3, 6, 12, 45, 55, 65, 90, 99, 112]
Time Required : 0.0001145380000000043

Process finished with exit code 0
```

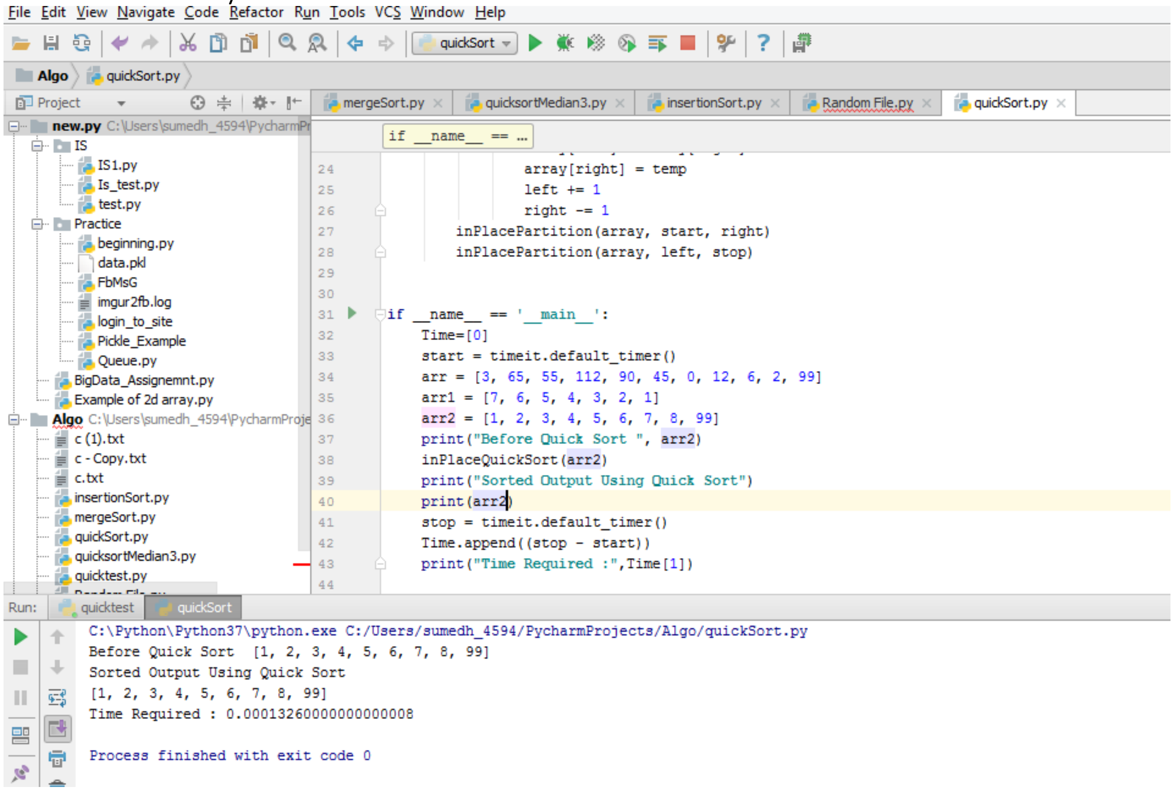
b. Reversely Sorted Array :-



```
                array[right] = temp
                left += 1
                right -= 1
        inPlacePartition(array, start, right)
        inPlacePartition(array, left, stop)


if __name__ == '__main__':
    Time=[0]
    start = timeit.default_timer()
    arr = [3, 65, 55, 112, 90, 45, 0, 12, 6, 2, 99]
    arr1 = [7, 6, 5, 4, 3, 2, 1]
    arr2 = [1, 2, 3, 4, 5, 6, 7, 8, 99]
    print("Before Quick Sort ", arr1)
    inPlaceQuickSort(arr1)
    print("Sorted Output Using Quick Sort")
    print(arr1)
    stop = timeit.default_timer()
    Time.append((stop - start))
    print("Time Required :",Time[1])
```

```
C:\Python\Python37\python.exe C:/Users/sumedh_4594/PycharmProjects/Algo/quickSort.py
Before Quick Sort  [7, 6, 5, 4, 3, 2, 1]
Sorted Output Using Quick Sort
[1, 2, 3, 4, 5, 6, 7]
Time Required : 9.195900000000257e-05

Process finished with exit code 0
```

c. Sorted Array :-



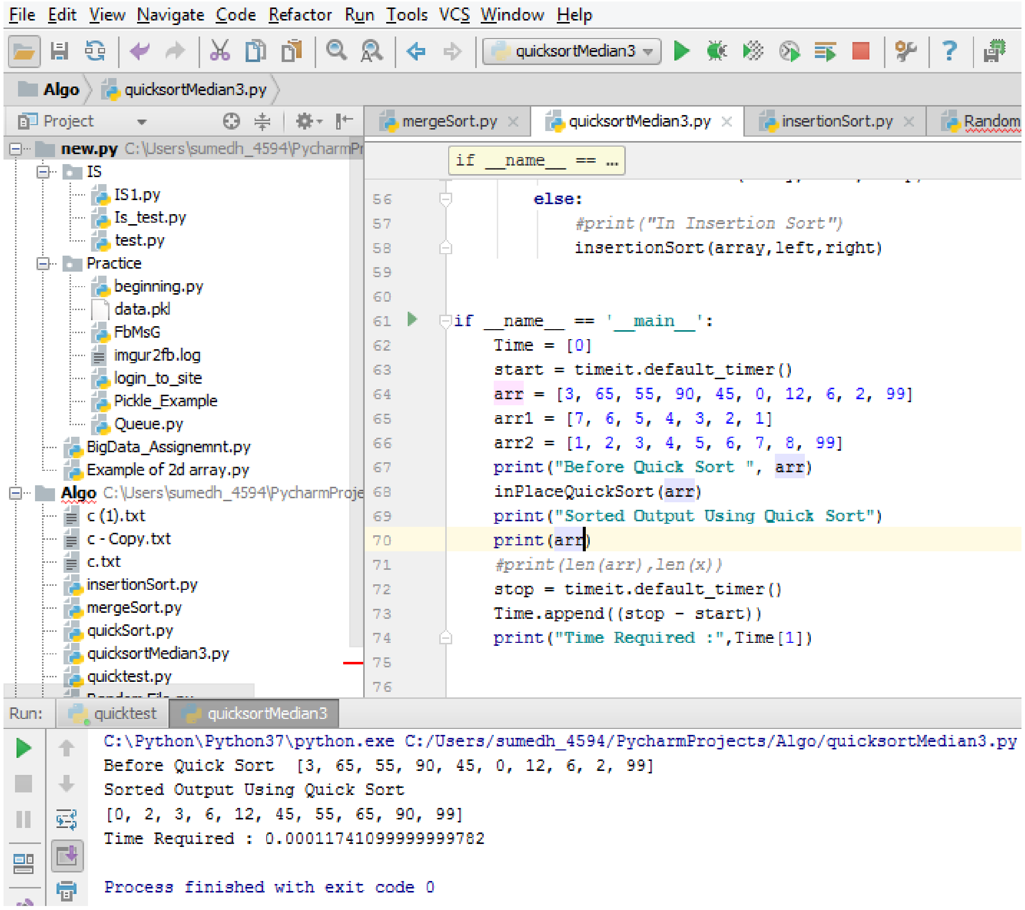
```
                    array[right] = temp
                    left += 1
                    right -= 1
            inPlacePartition(array, start, right)
            inPlacePartition(array, left, stop)


if __name__ == '__main__':
    Time=[0]
    start = timeit.default_timer()
    arr = [3, 65, 55, 112, 90, 45, 0, 12, 6, 2, 99]
    arr1 = [7, 6, 5, 4, 3, 2, 1]
    arr2 = [1, 2, 3, 4, 5, 6, 7, 8, 99]
    print("Before Quick Sort ", arr2)
    inPlaceQuickSort(arr2)
    print("Sorted Output Using Quick Sort")
    print(arr2)
    stop = timeit.default_timer()
    Time.append((stop - start))
    print("Time Required :",Time[1])
```

```
C:\Python\Python37\python.exe C:/Users/sumedh_4594/PycharmProjects/Algo/quickSort.py
Before Quick Sort  [1, 2, 3, 4, 5, 6, 7, 8, 99]
Sorted Output Using Quick Sort
[1, 2, 3, 4, 5, 6, 7, 8, 99]
Time Required : 0.00013260000000000008

Process finished with exit code 0
```

- Quick Sort performs worse for the sorted and reversely sorted array, if we choose pivot as minimum or maximum value. As it divides the array into 3 sub arrays with the length as (0, pivot, length(array)-1) or (length(array)-1, pivot, 0) repetitively and as a result time complexity reaches to O(n^2).

Modified Quick Sort :

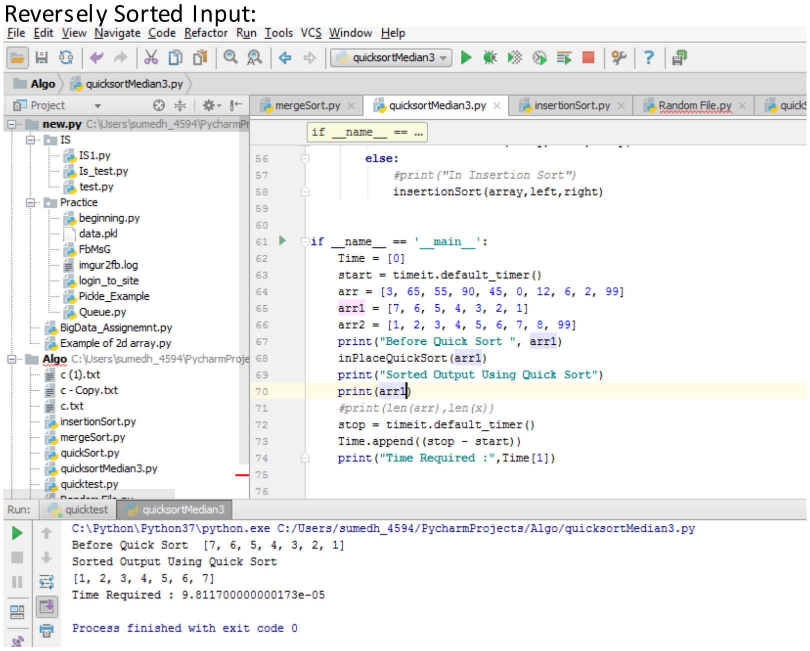
1. Random Input:



```
        else:
            #print("In Insertion Sort")
            insertionSort(array,left,right)


if __name__ == '__main__':
    Time = [0]
    start = timeit.default_timer()
    arr = [3, 65, 55, 90, 45, 0, 12, 6, 2, 99]
    arr1 = [7, 6, 5, 4, 3, 2, 1]
    arr2 = [1, 2, 3, 4, 5, 6, 7, 8, 99]
    print("Before Quick Sort ", arr)
    inPlaceQuickSort(arr)
    print("Sorted Output Using Quick Sort")
    print(arr)
    #print(len(arr),len(x))
    stop = timeit.default_timer()
    Time.append((stop - start))
    print("Time Required :",Time[1])
```

```
C:\Python\Python37\python.exe C:/Users/sumedh_4594/PycharmProjects/Algo/quicksortMedian3.py
Before Quick Sort  [3, 65, 55, 90, 45, 0, 12, 6, 2, 99]
Sorted Output Using Quick Sort
[0, 2, 3, 6, 12, 45, 55, 65, 90, 99]
Time Required : 0.00011741099999999782

Process finished with exit code 0
```

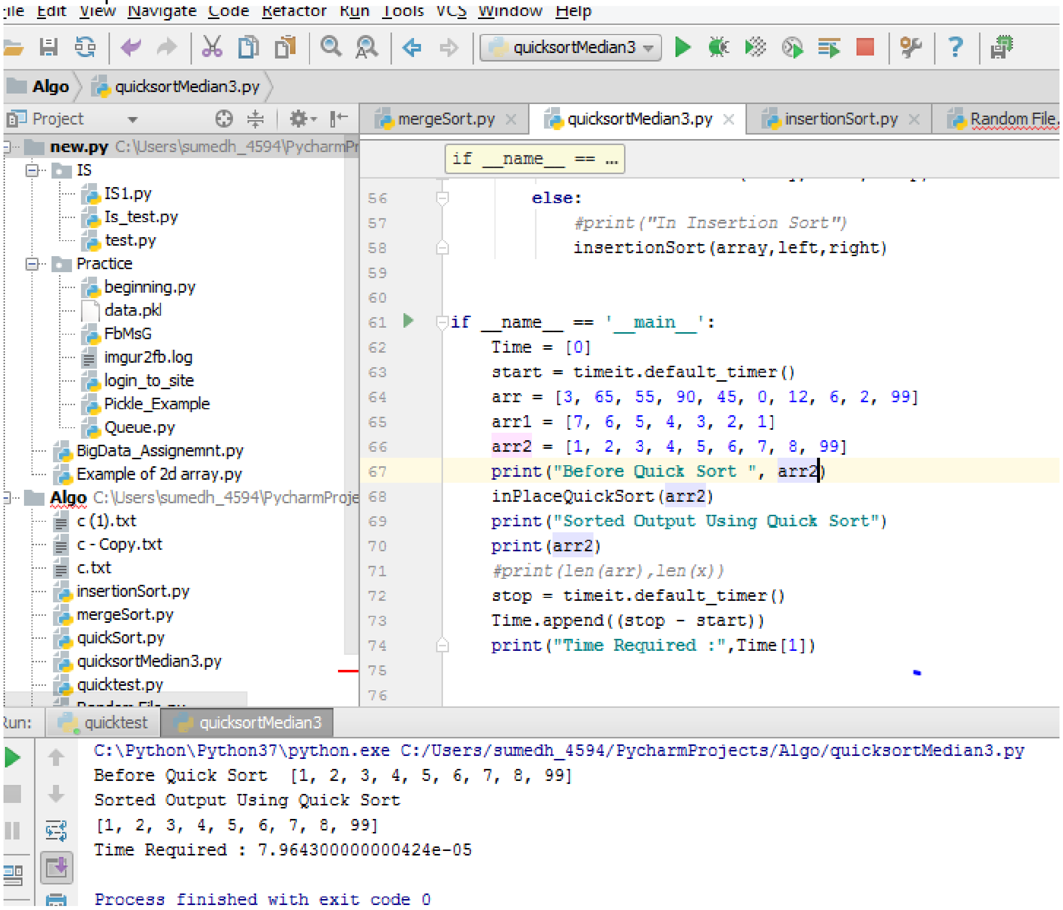
2. Reversely Sorted Input:



```
        else:
            #print("In Insertion Sort")
            insertionSort(array,left,right)


if __name__ == '__main__':
    Time = [0]
    start = timeit.default_timer()
    arr = [3, 65, 55, 90, 45, 0, 12, 6, 2, 99]
    arr1 = [7, 6, 5, 4, 3, 2, 1]
    arr2 = [1, 2, 3, 4, 5, 6, 7, 8, 99]
    print("Before Quick Sort ", arr1)
    inPlaceQuickSort(arr1)
    print("Sorted Output Using Quick Sort")
    print(arr1)
    #print(len(arr),len(x))
    stop = timeit.default_timer()
    Time.append((stop - start))
    print("Time Required :",Time[1])
```

```
C:\Python\Python37\python.exe C:/Users/sumedh_4594/PycharmProjects/Algo/quicksortMedian3.py
Before Quick Sort  [7, 6, 5, 4, 3, 2, 1]
Sorted Output Using Quick Sort
[1, 2, 3, 4, 5, 6, 7]
Time Required : 9.811700000000173e-05

Process finished with exit code 0
```

3. Sorted Input:



```
        else:
            #print("In Insertion Sort")
            insertionSort(array,left,right)


if __name__ == '__main__':
    Time = [0]
    start = timeit.default_timer()
    arr = [3, 65, 55, 90, 45, 0, 12, 6, 2, 99]
    arr1 = [7, 6, 5, 4, 3, 2, 1]
    arr2 = [1, 2, 3, 4, 5, 6, 7, 8, 99]
    print("Before Quick Sort ", arr2)
    inPlaceQuickSort(arr2)
    print("Sorted Output Using Quick Sort")
    print(arr2)
    #print(len(arr),len(x))
    stop = timeit.default_timer()
    Time.append((stop - start))
    print("Time Required :",Time[1])
```

```
C:\Python\Python37\python.exe C:/Users/sumedh_4594/PycharmProjects/Algo/quicksortMedian3.py
Before Quick Sort  [1, 2, 3, 4, 5, 6, 7, 8, 99]
Sorted Output Using Quick Sort
[1, 2, 3, 4, 5, 6, 7, 8, 99]
Time Required : 7.964300000000424e-05

Process finished with exit code 0
```

- **Output of the different sorting algorithms for large number of inputs.**

1. Code for this particular problem is present in the Random File.py.
2. Here I have imported all the files of individual sorting technique as a library files and have used random and time library from python to generate sequence of random numbers and to calculate time respectively.
3. Time_Merge, Time_Insertion, Time_Quick, Time_Quick_Median correspond to the array where I am storing the time required for the n inputs in one iteration. Also, avg_Merge, avg_Insertion, avg_Quick, avg_Quick_Median stores the average time required for particular algorithm for different input size.
4. Attached Result.txt contains the result of each iterations corresponding to n input size.
5. Time required is calculated just before the execution related to any sorting begins and it stops once the sorting is completed.
6. Below is the Final output, table and graph captured by the n number of inputs where n varies from 500 to 50,000.

Output :-

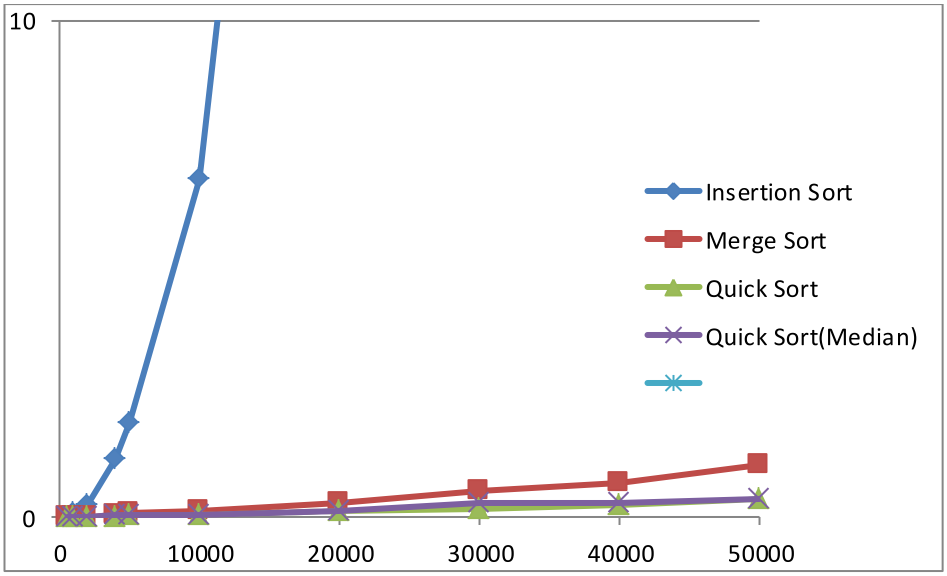
```
……………………..Final  Output……………………….
('Avg Time for Different Sorting Techniques for n=', 500)
('Insertion Sort :-', 0.018789546666666667)
('Merge  Sort :-', 0.008804693333333334)
('Quick Sort :-', 0.002946559999999999)
('Quick With Median Sort :-', 0.0021009066666666625)

……………………..Final  Output……………………….
('Avg Time for Different Sorting Techniques for n=', 1000)
('Insertion Sort :-', 0.07864447999999999)
('Merge  Sort :-', 0.01576277333333333)
('Quick Sort :-', 0.0054677333333333356)
('Quick With Median Sort :-', 0.0029964799999999996)

……………………..Final  Output……………………….
('Avg Time for Different Sorting Techniques for n=', 2000)
('Insertion Sort :-', 0.2559488)
('Merge  Sort :-', 0.04808362666666666)
('Quick Sort :-', 0.01817727999999999)
('Quick With Median Sort :-', 0.009412693333333333)

……………………..Final  Output……………………….
('Avg Time for Different Sorting Techniques for n=', 4000)
('Insertion Sort :-', 1.0254890666666667)
('Merge  Sort :-', 0.05067477333333331)
('Quick Sort :-', 0.013908906666666665)
('Quick With Median Sort :-', 0.022728533333333356)

……………………..Final  Output……………………….
('Avg Time for Different Sorting Techniques for n=', 5000)
('Insertion Sort :-', 1.4276339200000001)
('Merge  Sort :-', 0.10775765333333354)
('Quick Sort :-', 0.0294080000000001)
('Quick With Median Sort :-', 0.02731306666666633)

……………………..Final  Output……………………….
('Avg Time for Different Sorting Techniques for n=', 10000)
('Insertion Sort :-', 4.958820266666667)
('Merge  Sort :-', 0.15564629333333313)
('Quick Sort :-', 0.04750591999999987)
('Quick With Median Sort :-', 0.07373610666666686)

……………………..Final  Output……………………….
('Avg Time for Different Sorting Techniques for n=', 20000)
('Insertion Sort :-', 20.423711146666665)
('Merge  Sort :-', 0.4408512000000009)
('Quick Sort :-', 0.09950122666666594)
('Quick With Median Sort :-', 0.07246250666666754)
```

```
……………………..Final  Output……………………….
('Avg Time for Different Sorting Techniques for n=', 30000)
('Insertion Sort :-', 44.106855253333336)
('Merge  Sort :-', 0.4268753066666662)
('Quick Sort :-', 0.11471786666666617)
('Quick With Median Sort :-', 0.10657578666666723

……………………..Final  Output……………………….
('Avg Time for Different Sorting Techniques for n=', 40000)
('Insertion Sort :-', 67.41122048000001)
('Merge  Sort :-', 0.5740659199999953)
('Quick Sort :-', 0.15427498666666395)
('Quick With Median Sort :-', 0.14127872000000252)

……………………..Final  Output……………………….
('Avg Time for Different Sorting Techniques for n=', 50000)
('Insertion Sort :-', 111.26918570666666)
('Merge  Sort :-', 0.7814920533333236)
('Quick Sort :-', 0.2346056533333467)
('Quick With Median Sort :-', 0.2257088000000067)
```

Comparison Table 1:-

Length	Insertion Sort	Merge Sort	Quick Sort	Quick Sort(Median)
500	0.026698338	0.00789652	0.0042301	0.003987878
1000	0.09692793	0.02272154	0.0098892	0.005051559
2000	0.264120348	0.02934748	0.0177036	0.009878558
4000	1.190095826	0.05107431	0.0213951	0.030490804
5000	1.882631835	0.10584133	0.0403078	0.056903824
10000	6.800517431	0.14449838	0.0544977	0.056903824
20000	31.66799623	0.30603124	0.1194684	0.127404365
30000	62.43589262	0.5135413	0.1890847	0.282311294
40000	127.7116383	0.70684828	0.2449298	0.282311294
50000	206.3577304	1.05067364	0.367198	0.372183874

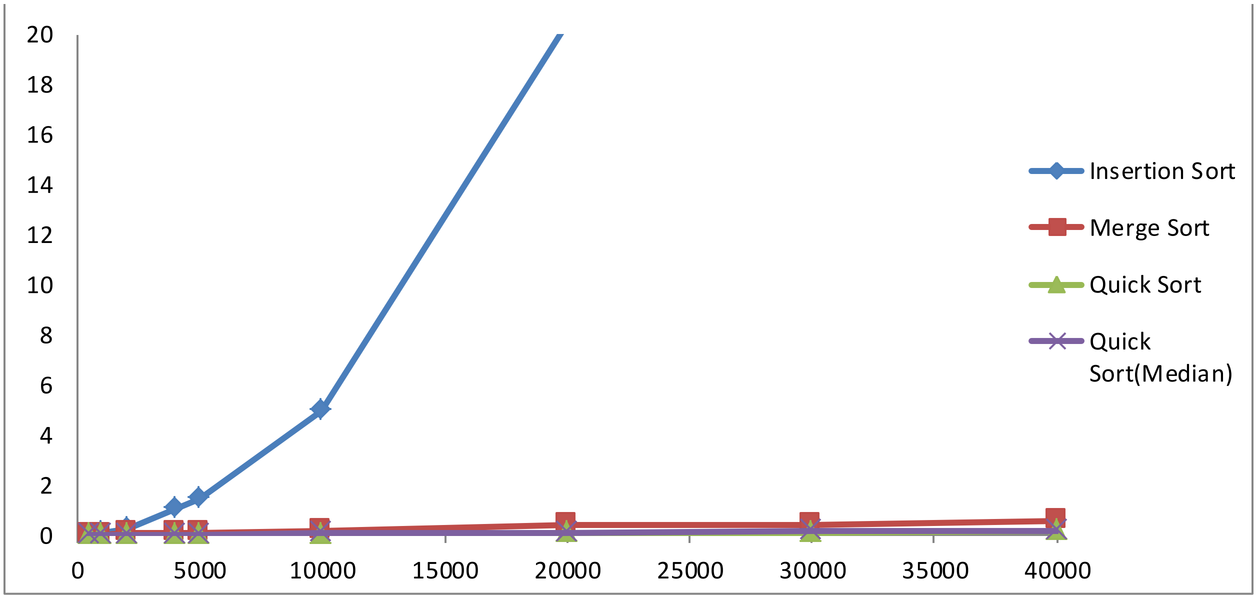
Graph 1:-



Comparison Table 2:-

Length	Insertion Sort	Merge Sort	Quick Sort	Quick Sort(Median)
500	0.018789547	0.008804693	0.00294656	0.002100907
1000	0.07864448	0.015762773	0.005467733	0.00299648
2000	0.2559488	0.048083627	0.01817728	0.009412693
4000	1.025489067	0.050674773	0.013908907	0.022728533
5000	1.42763392	0.107757653	0.029408	0.027313067
10000	4.958820267	0.155646293	0.040307781	0.073736107
20000	20.42371115	0.4408512	0.099501227	0.072462507
30000	44.10685525	0.426875307	0.114717867	0.154274987
40000	67.41122048	0.57406592	0.154274987	0.14127872
50000	111.2691857	0.781492053	0.234605653	0.2257088

Graph 2:-



Conclusion:-

1. From the above graph it can be inferred that insertion sort is very much expensive for the large amount of inputs. Hence it is not advisable to use insertion sort for large amount of data.
2. Insertion sort is almost linear for very small amount of data and hence it is useful in case of modified quick search.
3. Although merge sort and quick sort have same time complexity – O(nlogn), quick sort performs well for large amount of input data.
4. Performance of the quick sort highly depends on the arrangement of an array (sorted/unsorted). Quick sort's performance is of O(n^2) in the following cases:
   a. If input array is sorted and selected pivot is either minimum or maximum
   b. If input array is reversely sorted and selected pivot is either minimum or maximum
   c. If input array has all the elements with same value

5. In order to avoid the limitation, quick sort can be used with pivot equal to median of three approach and with the use of insertion sort for the input data in an array less than 10.
6. One of the important advantage of merge sort over quick sort can be stated as, it takes run time in order of O(nlogn) regardless of the order(sorted/unsorted) of the elements in an array