NAME: MD. NAYEEM MOLLA

Subject: Algorithm Design and Analysis

Problem formulation

Sorting algorithm is an algorithm that puts elements of a list in a certain order. The most frequently used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the efficiency of other algorithms which require input data to be in sorted lists. According to question I have to do direct insertion sort, bubble sort, quick sort, select sort, heap sort and two-way merge sort. I will do all of this:

The Design of Algorithm

Insertion Sort:

Insertion sort is a comparison-based algorithm that builds a final sorted array one element at a time. It iterates through an input array and removes one element per iteration, finds the place the element belongs in the array, and then places it there

Pseudocode:

```
for i from 1 to N
key = a[i]
j = i - 1
while j >= 0 and a[j] > key
a[j+1] = a[j]
j = j - 1
a[j+1] = key
```

Algorithm

To sort an array of size n in ascending order:
1: Iterate from arr[1] to arr[n] over the array.
2: Compare the current element (key) to its predecessor.
3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Example:
12, 11, 13, 5, 6

Let us loop for i = 1 (second element of the array) to 4 (last element of the array)

i = 1. Since 11 is smaller than 12, move 12 and insert 11 before 12
11, 12, 13, 5, 6

i = 2. 13 will remain at its position as all elements in A[0..I-1] are smaller than 13
11, 12, 13, 5, 6

i = 3. 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.
5, 11, 12, 13, 6

i = 4. 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.
5, 6, 11, 12, 13

Bubble Sort:

Bubble sort is a comparison-based algorithm that compares each pair of elements in an array and swaps them if they are out of order until the entire array is sorted. For each element in the list, the algorithm compares every pair of elements.

Pseudocode:

```
swapped = true
  while swapped
      swapped = false
      for j from 0 to N - 1
         if a[j] > a[j + 1]
            swap( a[j], a[j + 1] )
            swapped = true
```

Example:
First Pass:
(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:
(**1 4** 2 5 8) –> (**1 4** 2 5 8)
(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:
(**1 2** 4 5 8) –> (**1 2** 4 5 8)
(1 **2 4** 5 8) –> (1 **2 4** 5 8)

(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Quick Sort:

Quick sort is a comparison-based algorithm that uses divide-and-conquer to sort an array. The algorithm picks a pivot element, A[q] and then rearranges the array into two subarrays A[p . . . . q-1] , such that all elements are less than A[q] and A[q+1 . . . r] , such that all elements are greater than or equal to A[q].

Pseudocode:

```
Quicksort(A as array, low as int, high as int){
   if (low < high){
      pivot_location = Partition(A,low,high)
      Quicksort(A,low, pivot_location)
      Quicksort(A, pivot_location + 1, high)
   }
}
Partition(A as array, low as int, high as int){
    pivot = A[low]
    leftwall = low

    for i = low + 1 to high{
       if (A[i] < pivot) then{
          swap(A[i], A[leftwall + 1])
          leftwall = leftwall + 1
       }
    }
    swap(pivot,A[leftwall])

   return (leftwall)}
```

Algorithm
There can be many ways to do partition, following pseudo code adopts the method given in CLRS book. The logic is simple, we start from the leftmost

element and keep track of index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current element with arr[i]. Otherwise, we ignore current element.

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
   if (low < high)
   {
      /* pi is partitioning index, arr[pi] is now
         at right place */
      pi = partition(arr, low, high);

      quickSort(arr, low, pi - 1);  // Before pi
      quickSort(arr, pi + 1, high); // After pi
   }
}
   swap arr[i + 1] and arr[high])
   return (i + 1)
}
```

Selection Sort:

Selection sort is an in-place comparison-based algorithm that divided the list into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

Pseudocode:

```
func selsrtI(lst)
    max = length(lst) - 1

     for i from 0 to max
    key = lst[i]
    keyj = i

    for j from i+1 to max
    if lst[j] < key
            key = lst[j]
            keyj = j

     lst[keyj] = lst[i]
     lst[i] = key

     return lst
 end func
```

Following example explains the above steps:

arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4]

// and place it at beginning

11 25 12 22 64

// Find the minimum element in arr[1...4]

// and place it at beginning of arr[1...4]

11 12 25 22 64

// Find the minimum element in arr[2...4]

// and place it at beginning of arr[2...4]

11 12 22 25 64

// Find the minimum element in arr[3...4]

// and place it at beginning of arr[3...4]

11 12 22 25 64

Heap Sort:

Heap sort is a comparison-based algorithm that uses a binary heap data structure to sort elements. It divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element and moving that to the sorted region.

Pseudocode:

```
Heapsort(A as array)
   BuildHeap(A)
   for i = n to 1
```

```
        swap(A[1], A[i])
        n = n - 1
        Heapify(A, 1)

BuildHeap(A as array)
    n = elements_in(A)
    for i = floor(n/2) to 1
        Heapify(A,i,n)

Heapify(A as array, i as int, n as int)
    left = 2i
    right = 2i+1

    if (left <= n) and (A[left] > A[i])
        max = left
    else
        max = i

    if (right<=n) and (A[right] > A[max])
        max = right

    if (max != i)
        swap(A[i], A[max])
        Heapify(A, max)
```

How to build the heap:
Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom-up order.
Lets understand with the help of an example:

Input data: 4, 10, 3, 5, 1

```
      4(0)

     /  \

  10(1)  3(2)
```

```
   /   \
 5(3)    1(4)
```

The numbers in bracket represent the indices in the array representation of data.

Applying heapify procedure to index 1:

```
        4(0)
       /   \
   10(1)    3(2)
   /   \
5(3)    1(4)
```

Applying heapify procedure to index 0:

```
       10(0)
       /  \
    5(1)  3(2)
   /   \
 4(3)    1(4)
```

The heapify procedure calls itself recursively to build heap in top-down manner.

Two-away Merge Sort:

Merge sort is a comparison-based algorithm that focuses on how to merge together two pre-sorted arrays such that the resulting array is also sorted.

Pseudocode:

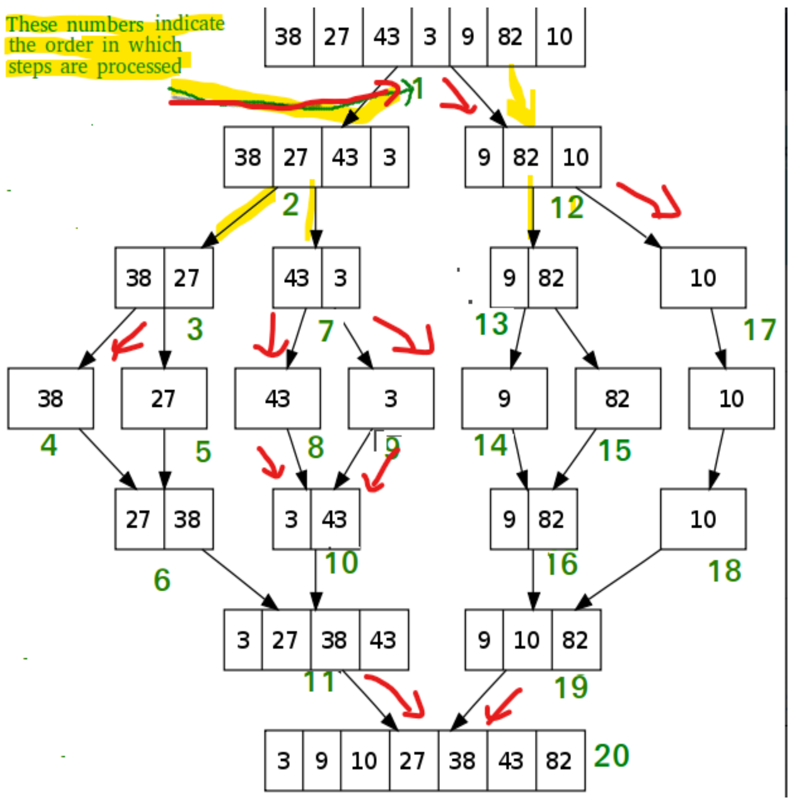
```
func mergesort( var a as array )
  if ( n == 1 ) return a

  var l1 as array = a[0] ... a[n/2]
  var l2 as array = a[n/2+1] ... a[n]

  l1 = mergesort( l1 )
  l2 = mergesort( l2 )

  return merge( l1, l2 )
end func

func merge( var a as array, var b as array )
  var c as array

  while ( a and b have elements )
      if ( a[0] > b[0] )
          add b[0] to the end of c
          remove b[0] from b
      else
          add a[0] to the end of c
          remove a[0] from a
  while ( a has elements )
      add a[0] to the end of c
      remove a[0] from a
  while ( b has elements )
      add b[0] to the end of c
      remove b[0] from b
  return c
 end func
```

Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. **The merge () function** is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See the following C implementation for details.

ergeSort(arr[], l, r)

```
If r > l
  1. Find the middle point to divide the array into two halves:
        middle m = (l+r)/2
  2. Call mergeSort for first half:
        Call mergeSort(arr, l, m)
  3. Call mergeSort for second half:
        Call mergeSort(arr, m+1, r)
  4. Merge the two halves sorted in step 2 and 3:
        Call merge(arr, l, m, r)
```

These numbers indicate the order in which steps are processed



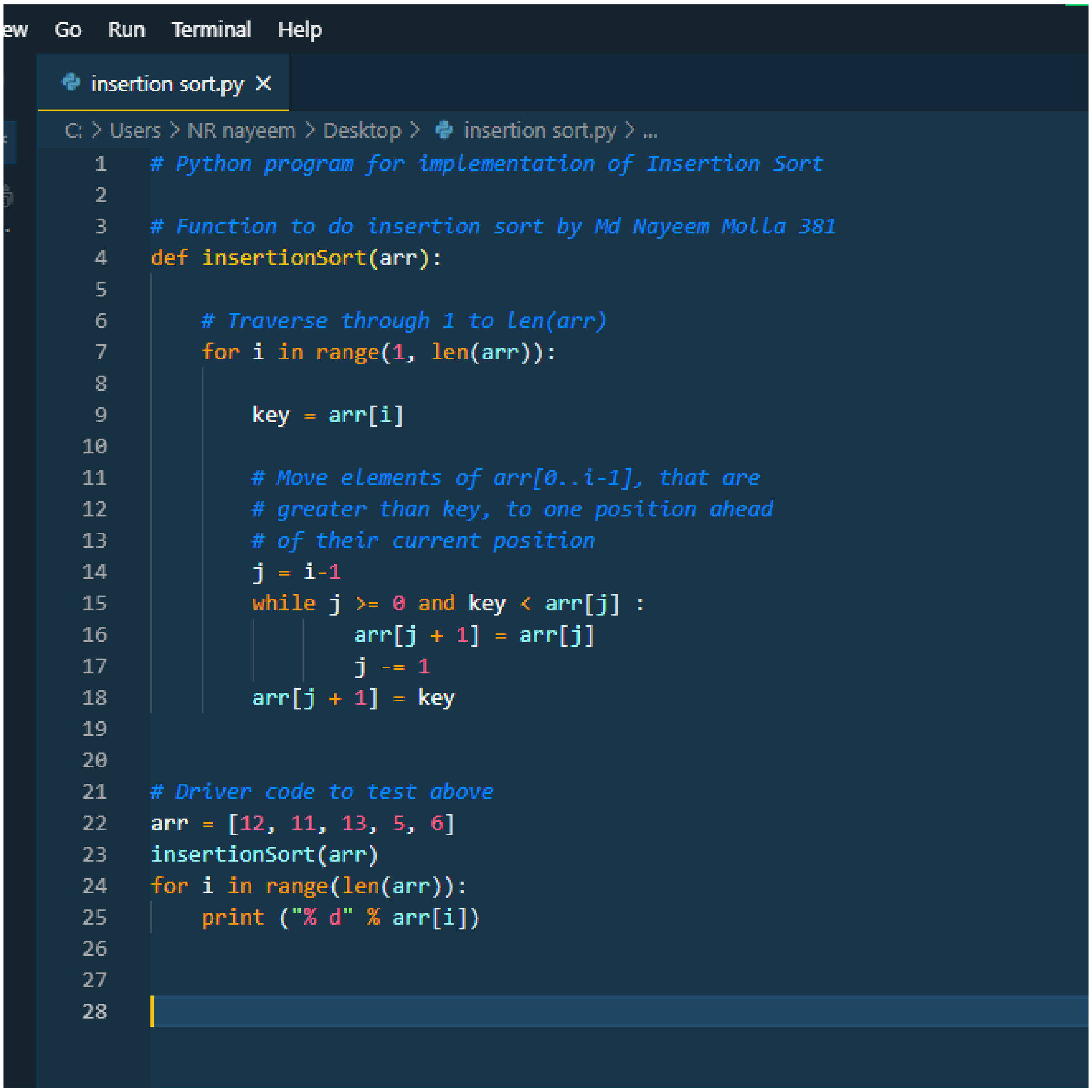
Implementation

Insertion sort

Source code:

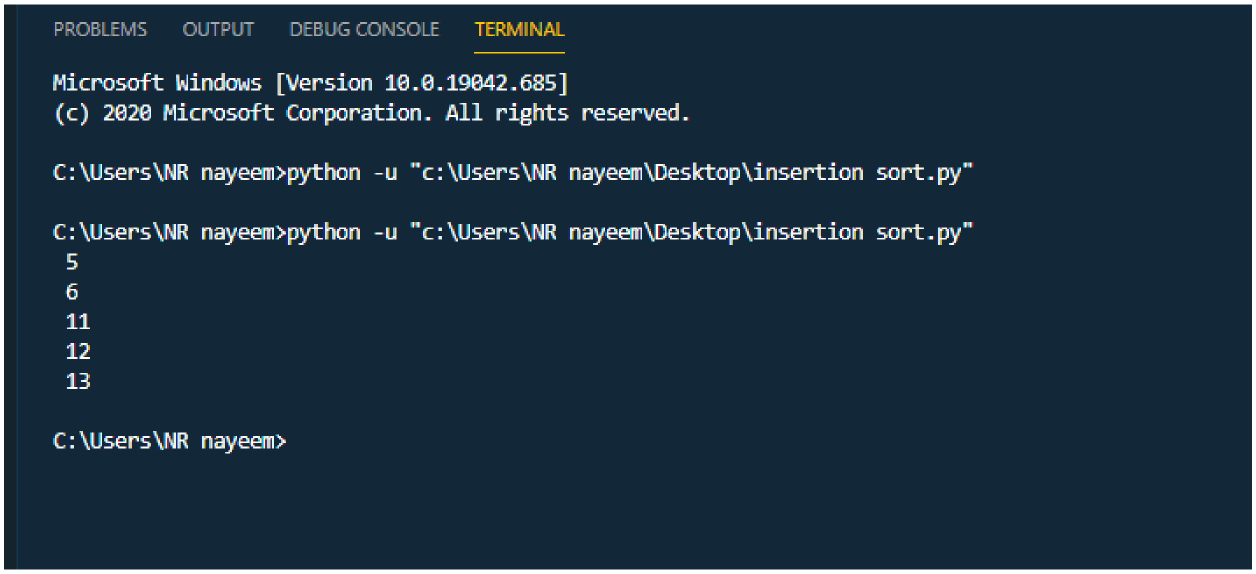
```
# Python program for implementation of Insertion Sort

# Function to do insertion sort by Md Nayeem Molla 381
def insertionSort(arr):

    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):

        key = arr[i]

        # Move elements of arr[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
        j = i-1
        while j >= 0 and key < arr[j] :
                arr[j + 1] = arr[j]
                j -= 1
        arr[j + 1] = key


# Driver code to test above
arr = [12, 11, 13, 5, 6]
insertionSort(arr)
for i in range(len(arr)):
    print ("% d" % arr[i])
```

Input:



```python
# Python program for implementation of Insertion Sort

# Function to do insertion sort by Md Nayeem Molla 381
def insertionSort(arr):

    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):

        key = arr[i]

        # Move elements of arr[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
        j = i-1
        while j >= 0 and key < arr[j] :
                arr[j + 1] = arr[j]
                j -= 1
        arr[j + 1] = key


# Driver code to test above
arr = [12, 11, 13, 5, 6]
insertionSort(arr)
for i in range(len(arr)):
    print ("% d" % arr[i])


```

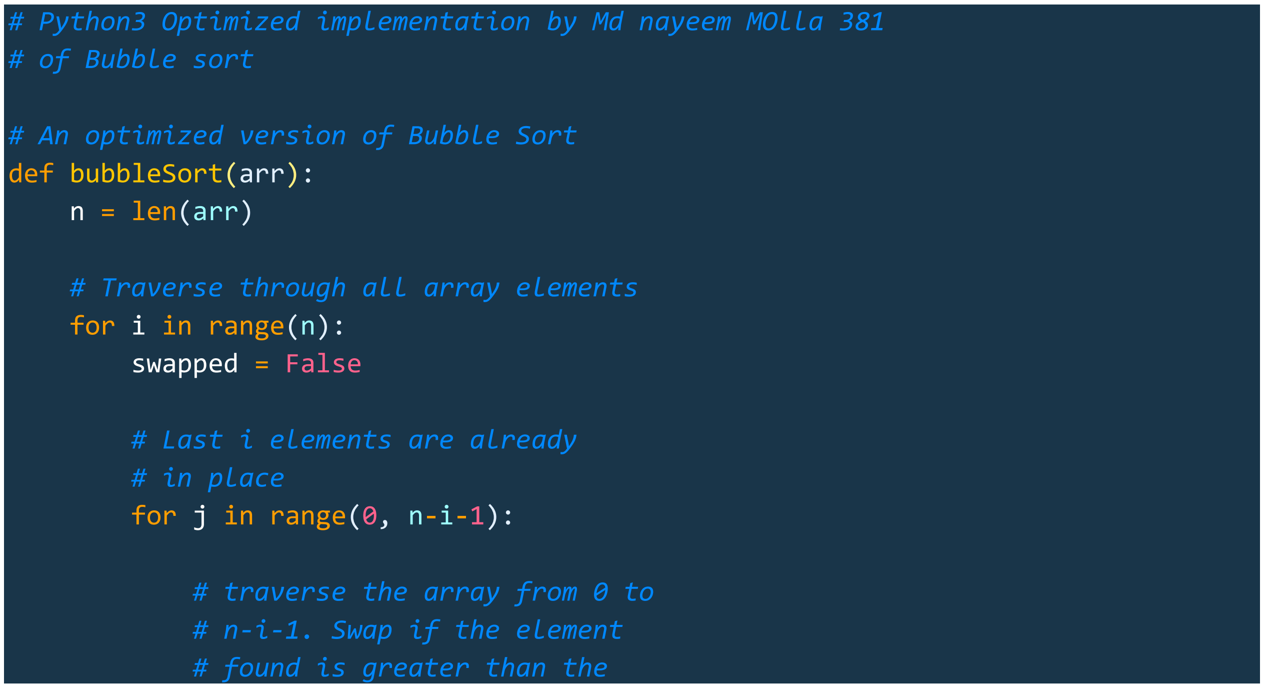
Output:



```
Microsoft Windows [Version 10.0.19042.685]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\NR nayeem>python -u "c:\Users\NR nayeem\Desktop\insertion sort.py"

C:\Users\NR nayeem>python -u "c:\Users\NR nayeem\Desktop\insertion sort.py"
 5
 6
 11
 12
 13

C:\Users\NR nayeem>
```

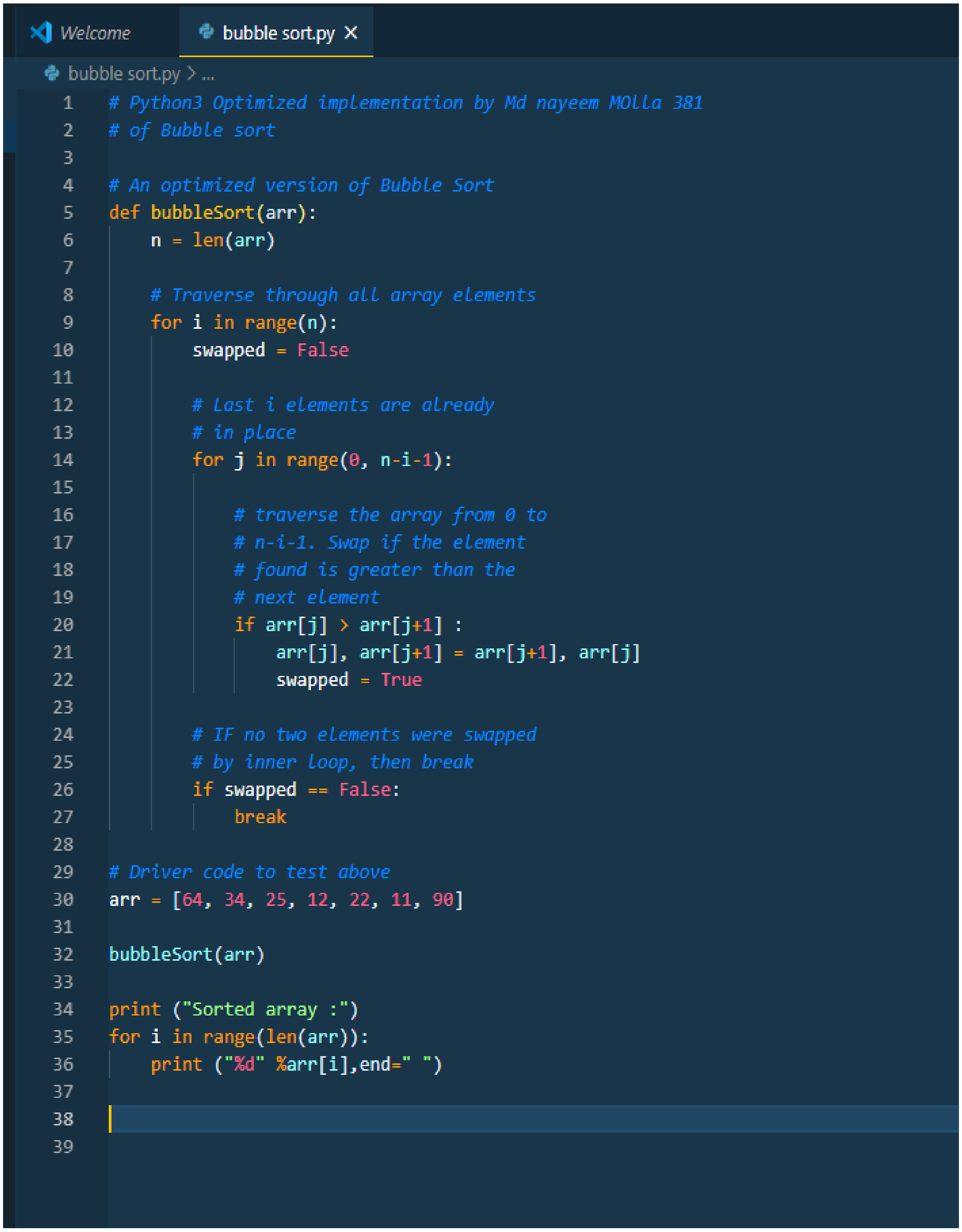
Bubble sort

Source code:



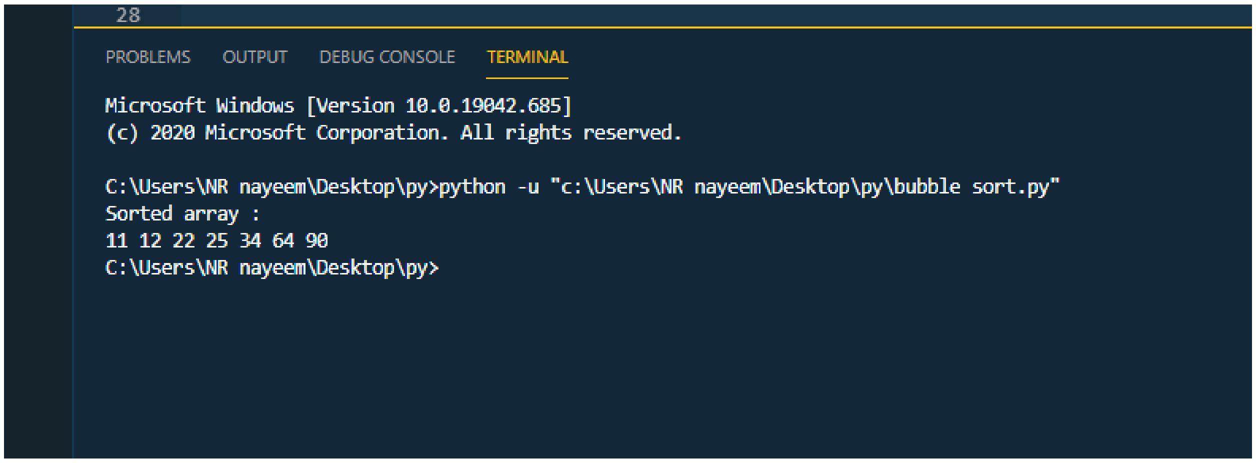
```python
# Python3 Optimized implementation by Md nayeem MOlla 381
# of Bubble sort

# An optimized version of Bubble Sort
def bubbleSort(arr):
    n = len(arr)

    # Traverse through all array elements
    for i in range(n):
        swapped = False

        # Last i elements are already
        # in place
        for j in range(0, n-i-1):

            # traverse the array from 0 to
            # n-i-1. Swap if the element
            # found is greater than the
            # next element
            if arr[j] > arr[j+1] :
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True

        # IF no two elements were swapped
        # by inner loop, then break
        if swapped == False:
            break

# Driver code to test above
arr = [64, 34, 25, 12, 22, 11, 90]

bubbleSort(arr)

print ("Sorted array :")
for i in range(len(arr)):
    print ("%d" %arr[i],end=" ")
```

input:



```python
# Python3 Optimized implementation by Md nayeem MOlla 381
# of Bubble sort

# An optimized version of Bubble Sort
def bubbleSort(arr):
    n = len(arr)

    # Traverse through all array elements
    for i in range(n):
        swapped = False

        # Last i elements are already
        # in place
        for j in range(0, n-i-1):

            # traverse the array from 0 to
            # n-i-1. Swap if the element
            # found is greater than the
            # next element
            if arr[j] > arr[j+1] :
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True

        # IF no two elements were swapped
        # by inner loop, then break
        if swapped == False:
            break

# Driver code to test above
arr = [64, 34, 25, 12, 22, 11, 90]

bubbleSort(arr)

print ("Sorted array :")
for i in range(len(arr)):
    print ("%d" %arr[i],end=" ")

```

Output:



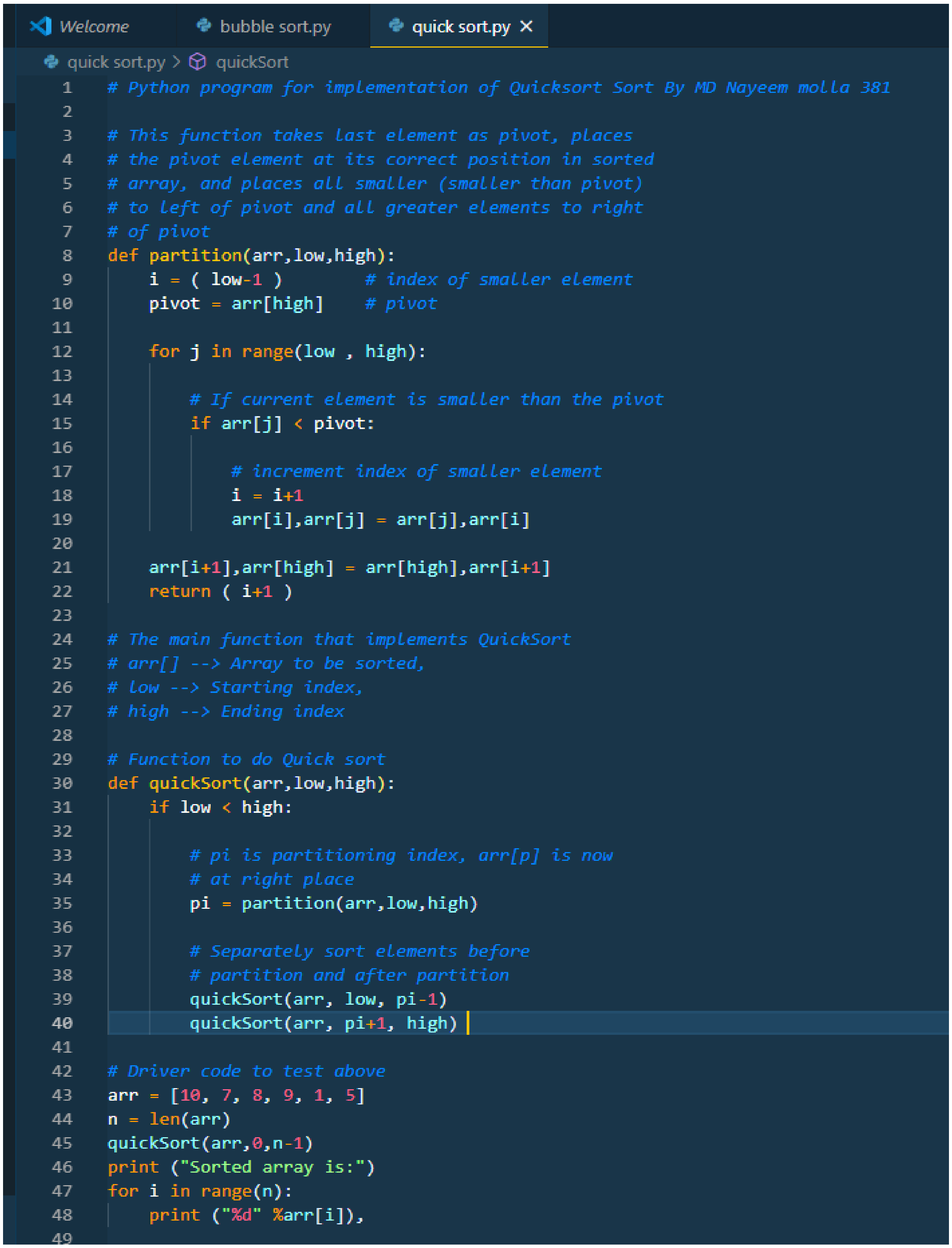
```
Microsoft Windows [Version 10.0.19042.685]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\NR nayeem\Desktop\py>python -u "c:\Users\NR nayeem\Desktop\py\bubble sort.py"
Sorted array :
11 12 22 25 34 64 90
C:\Users\NR nayeem\Desktop\py>
```

Quick sort

Source code:

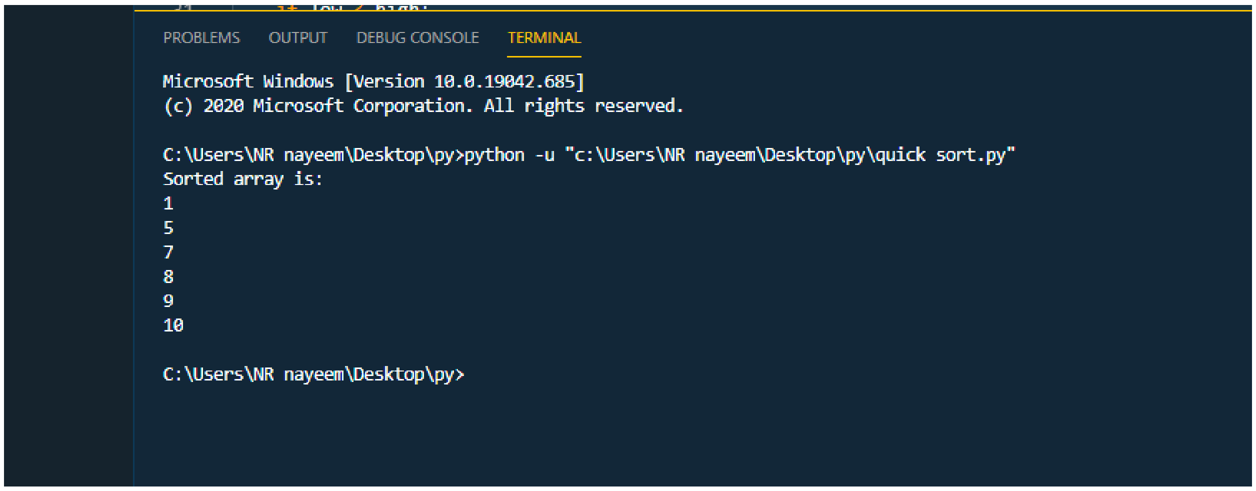
```python
# Python program for implementation of Quicksort Sort By MD Nayeem molla 381

# This function takes last element as pivot, places
# the pivot element at its correct position in sorted
# array, and places all smaller (smaller than pivot)
# to left of pivot and all greater elements to right
# of pivot
def partition(arr,low,high):
    i = ( low-1 )         # index of smaller element
    pivot = arr[high]     # pivot

    for j in range(low , high):

        # If current element is smaller than the pivot
        if arr[j] < pivot:

            # increment index of smaller element
            i = i+1
            arr[i],arr[j] = arr[j],arr[i]

    arr[i+1],arr[high] = arr[high],arr[i+1]
    return ( i+1 )

# The main function that implements QuickSort
# arr[] --> Array to be sorted,
# low --> Starting index,
# high --> Ending index

# Function to do Quick sort
def quickSort(arr,low,high):
    if low < high:

        # pi is partitioning index, arr[p] is now
        # at right place
        pi = partition(arr,low,high)

        # Separately sort elements before
        # partition and after partition
        quickSort(arr, low, pi-1)
        quickSort(arr, pi+1, high)

# Driver code to test above
arr = [10, 7, 8, 9, 1, 5]
n = len(arr)
quickSort(arr,0,n-1)
print ("Sorted array is:")
for i in range(n):
    print ("%d" %arr[i]),
```

Input:



```python
# Python program for implementation of Quicksort Sort By MD Nayeem molla 381

# This function takes last element as pivot, places
# the pivot element at its correct position in sorted
# array, and places all smaller (smaller than pivot)
# to left of pivot and all greater elements to right
# of pivot
def partition(arr,low,high):
    i = ( low-1 )         # index of smaller element
    pivot = arr[high]     # pivot

    for j in range(low , high):

        # If current element is smaller than the pivot
        if arr[j] < pivot:

            # increment index of smaller element
            i = i+1
            arr[i],arr[j] = arr[j],arr[i]

    arr[i+1],arr[high] = arr[high],arr[i+1]
    return ( i+1 )

# The main function that implements QuickSort
# arr[] --> Array to be sorted,
# low --> Starting index,
# high --> Ending index

# Function to do Quick sort
def quickSort(arr,low,high):
    if low < high:

        # pi is partitioning index, arr[p] is now
        # at right place
        pi = partition(arr,low,high)

        # Separately sort elements before
        # partition and after partition
        quickSort(arr, low, pi-1)
        quickSort(arr, pi+1, high)

# Driver code to test above
arr = [10, 7, 8, 9, 1, 5]
n = len(arr)
quickSort(arr,0,n-1)
print ("Sorted array is:")
for i in range(n):
    print ("%d" %arr[i]),

```

Output:



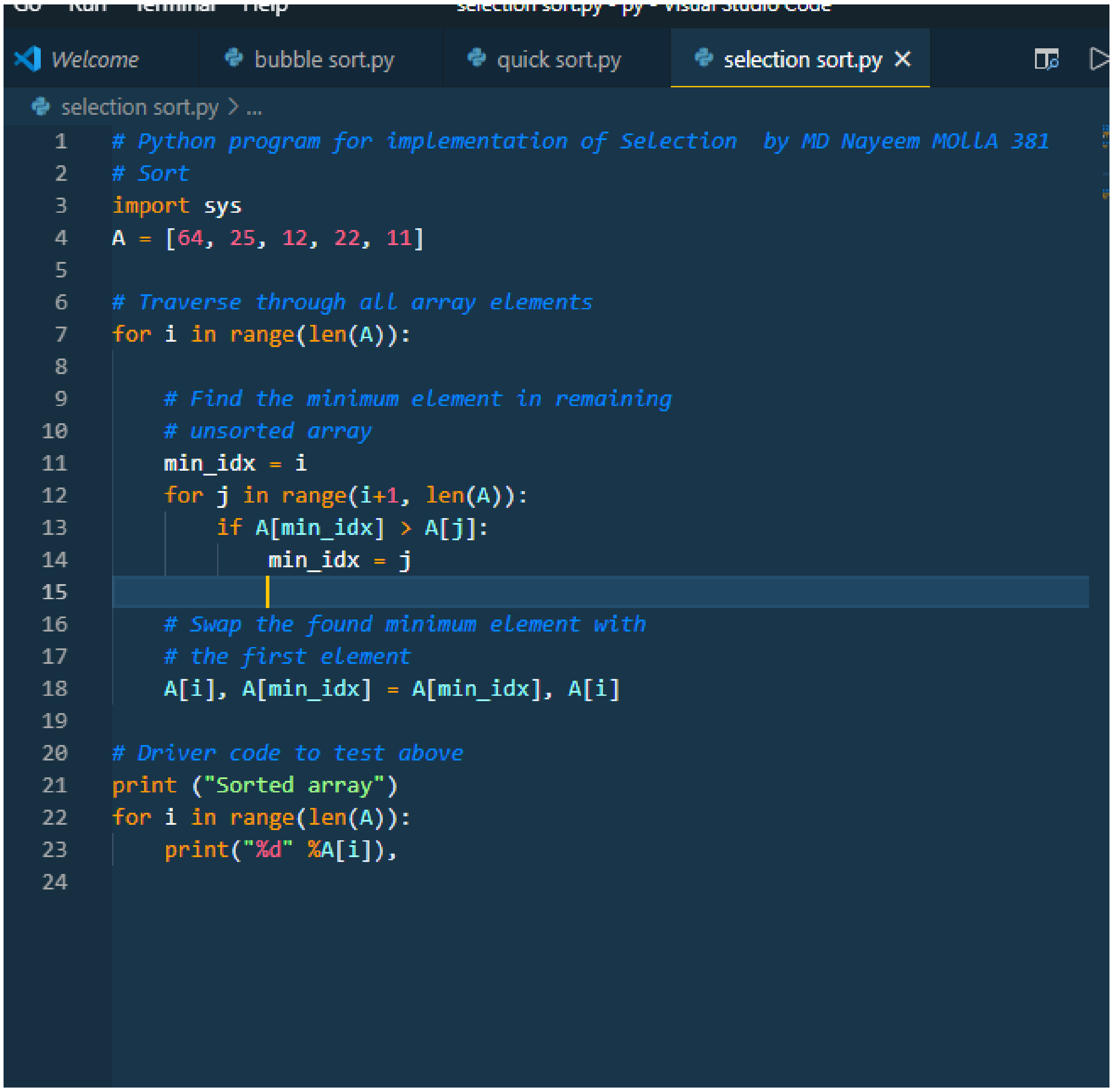
```
Microsoft Windows [Version 10.0.19042.685]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\NR nayeem\Desktop\py>python -u "c:\Users\NR nayeem\Desktop\py\quick sort.py"
Sorted array is:
1
5
7
8
9
10

C:\Users\NR nayeem\Desktop\py>
```

Selection Sort:

Source code:

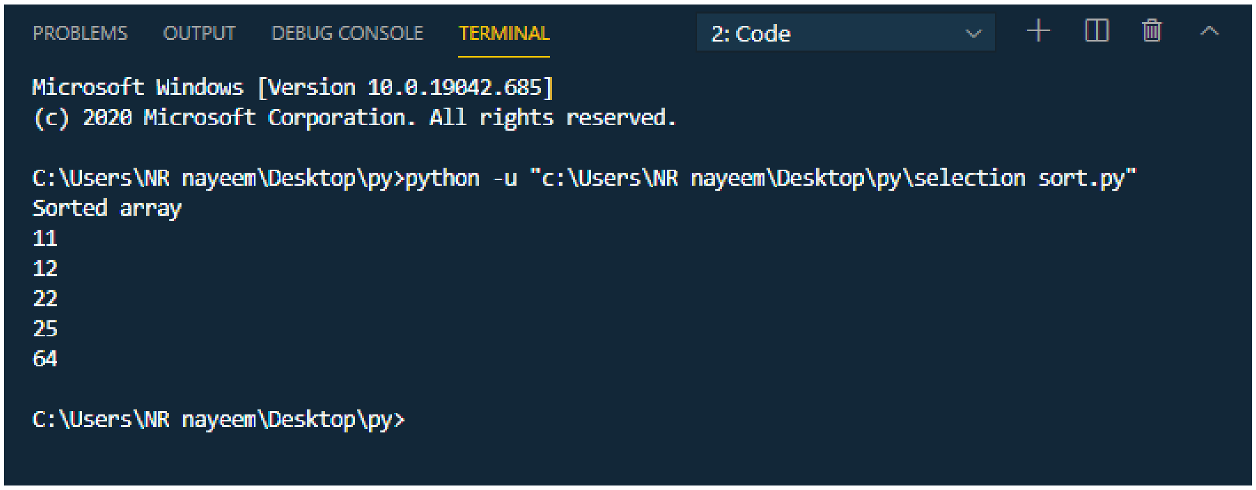
```python
# Python program for implementation of Selection  by MD Nayeem MOLLA 381
# Sort
import sys
A = [64, 25, 12, 22, 11]

# Traverse through all array elements
for i in range(len(A)):

    # Find the minimum element in remaining
    # unsorted array
    min_idx = i
    for j in range(i+1, len(A)):
        if A[min_idx] > A[j]:
            min_idx = j

    # Swap the found minimum element with
    # the first element
    A[i], A[min_idx] = A[min_idx], A[i]

# Driver code to test above
print ("Sorted array")
for i in range(len(A)):
    print("%d" %A[i]),
```

<u>Input:</u>



```
# Python program for implementation of Selection  by MD Nayeem MOlLA 381
# Sort
import sys
A = [64, 25, 12, 22, 11]

# Traverse through all array elements
for i in range(len(A)):

    # Find the minimum element in remaining
    # unsorted array
    min_idx = i
    for j in range(i+1, len(A)):
        if A[min_idx] > A[j]:
            min_idx = j

    # Swap the found minimum element with
    # the first element
    A[i], A[min_idx] = A[min_idx], A[i]

# Driver code to test above
print ("Sorted array")
for i in range(len(A)):
    print("%d" %A[i]),
```

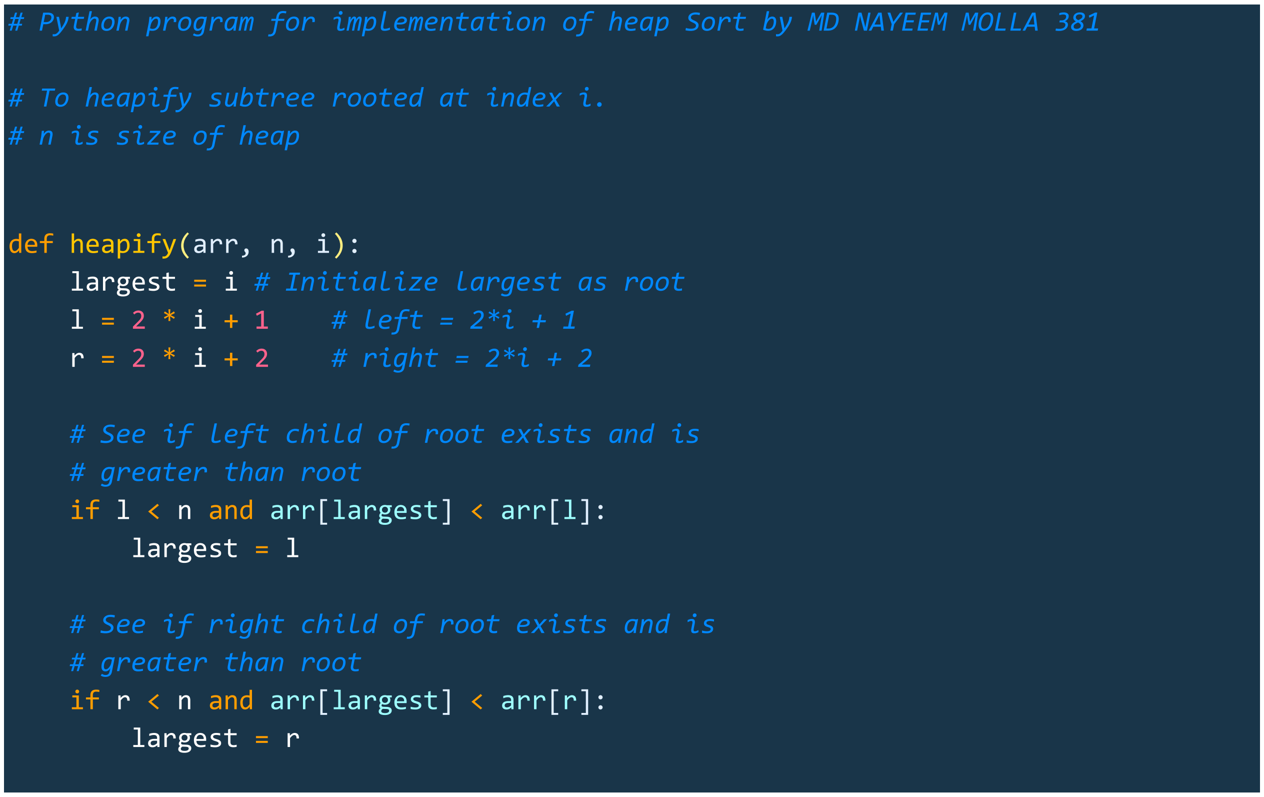
Output:



```
Microsoft Windows [Version 10.0.19042.685]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\NR nayeem\Desktop\py>python -u "c:\Users\NR nayeem\Desktop\py\selection sort.py"
Sorted array
11
12
22
25
64

C:\Users\NR nayeem\Desktop\py>
```

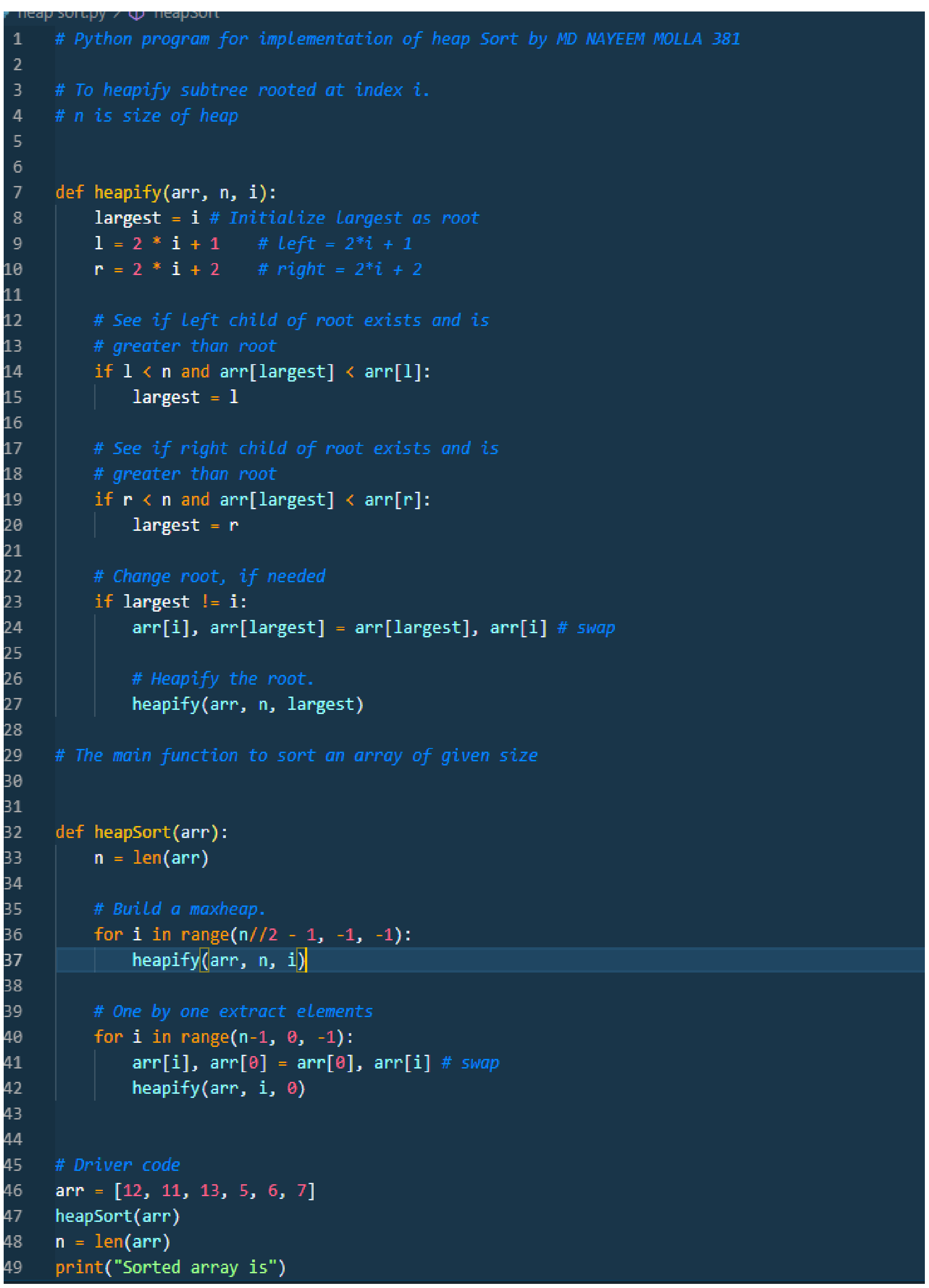
Heap sort

Source code:

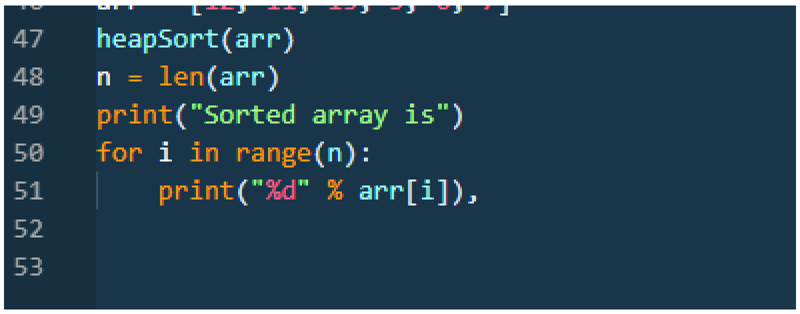


```python
# Python program for implementation of heap Sort by MD NAYEEM MOLLA 381

# To heapify subtree rooted at index i.
# n is size of heap


def heapify(arr, n, i):
    largest = i # Initialize largest as root
    l = 2 * i + 1     # left = 2*i + 1
    r = 2 * i + 2     # right = 2*i + 2

    # See if left child of root exists and is
    # greater than root
    if l < n and arr[largest] < arr[l]:
        largest = l

    # See if right child of root exists and is
    # greater than root
    if r < n and arr[largest] < arr[r]:
        largest = r
    # Change root, if needed
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i] # swap

        # Heapify the root.
        heapify(arr, n, largest)

# The main function to sort an array of given size


def heapSort(arr):
    n = len(arr)

    # Build a maxheap.
    for i in range(n//2 - 1, -1, -1):
        heapify(arr, n, i)

    # One by one extract elements
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i] # swap
        heapify(arr, i, 0)


# Driver code
arr = [12, 11, 13, 5, 6, 7]
heapSort(arr)
n = len(arr)
print("Sorted array is")
for i in range(n):
    print("%d" % arr[i]),
```

input:

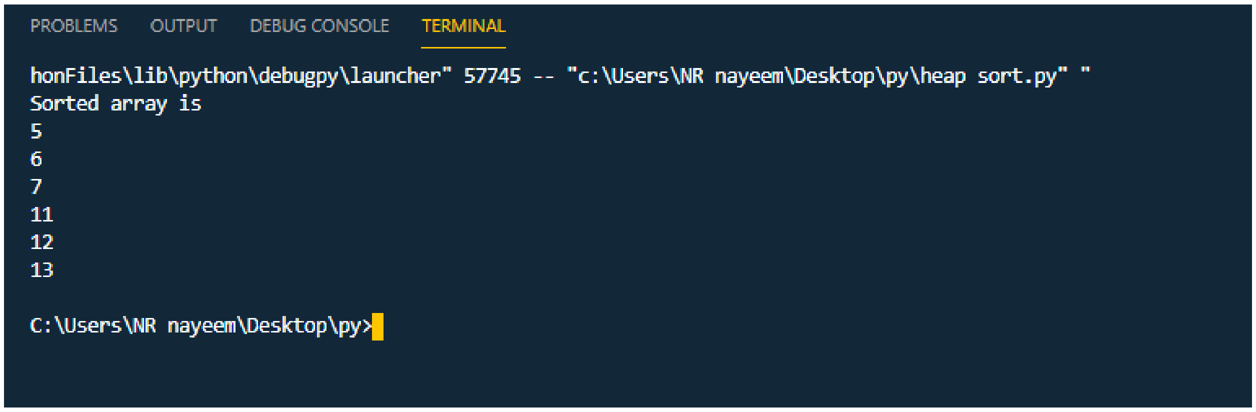


```
# Python program for implementation of heap Sort by MD NAYEEM MOLLA 381

# To heapify subtree rooted at index i.
# n is size of heap


def heapify(arr, n, i):
    largest = i # Initialize largest as root
    l = 2 * i + 1    # left = 2*i + 1
    r = 2 * i + 2    # right = 2*i + 2

    # See if left child of root exists and is
    # greater than root
    if l < n and arr[largest] < arr[l]:
        largest = l

    # See if right child of root exists and is
    # greater than root
    if r < n and arr[largest] < arr[r]:
        largest = r

    # Change root, if needed
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i] # swap

        # Heapify the root.
        heapify(arr, n, largest)

# The main function to sort an array of given size


def heapSort(arr):
    n = len(arr)

    # Build a maxheap.
    for i in range(n//2 - 1, -1, -1):
        heapify(arr, n, i)

    # One by one extract elements
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i] # swap
        heapify(arr, i, 0)


# Driver code
arr = [12, 11, 13, 5, 6, 7]
heapSort(arr)
n = len(arr)
print("Sorted array is")
```



```python
heapSort(arr)
n = len(arr)
print("Sorted array is")
for i in range(n):
    print("%d" % arr[i]),
```

Output:



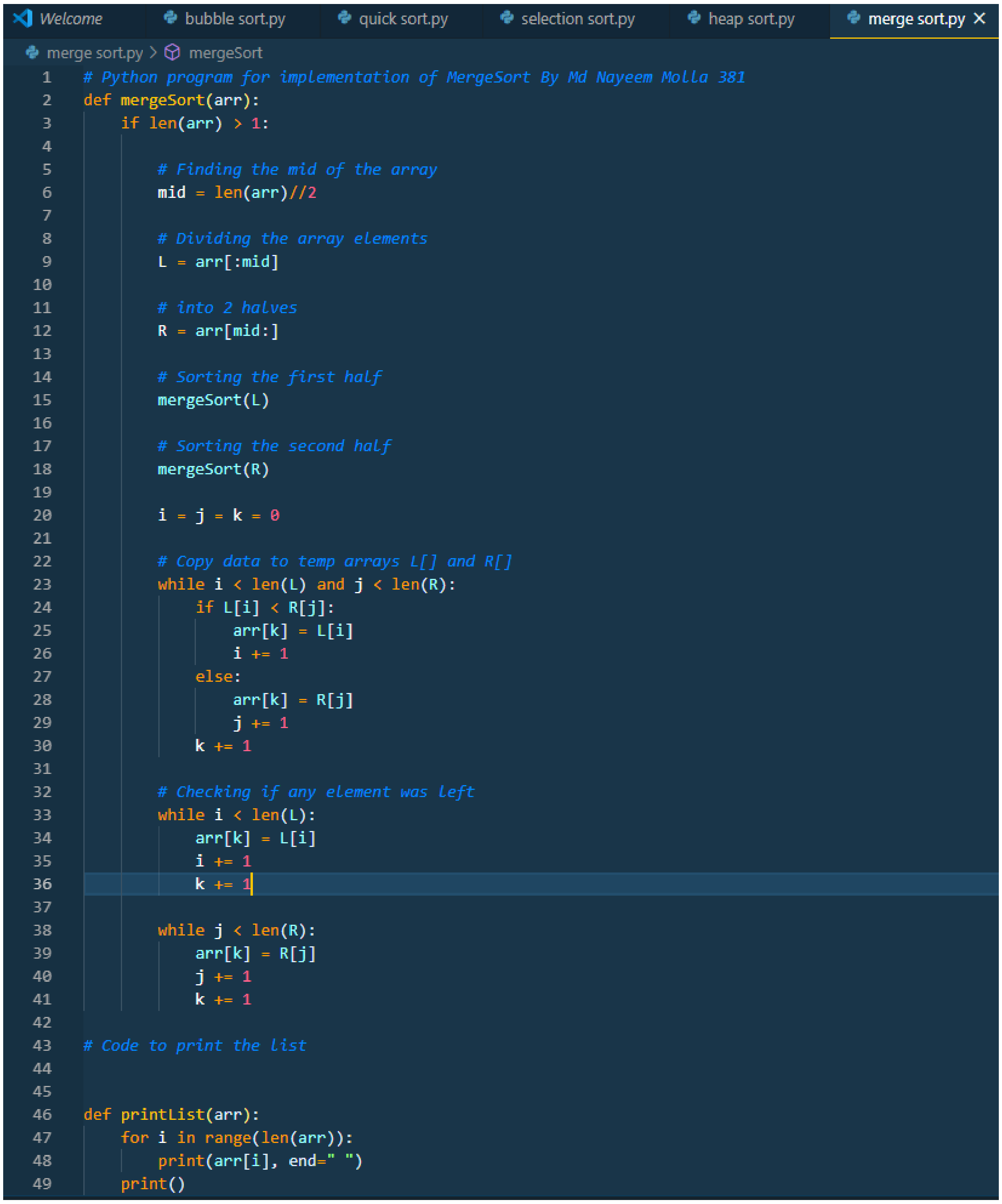
```
honFiles\lib\python\debugpy\launcher" 57745 -- "c:\Users\NR nayeem\Desktop\py\heap sort.py" "
Sorted array is
5
6
7
11
12
13

C:\Users\NR nayeem\Desktop\py>
```

Merge sort

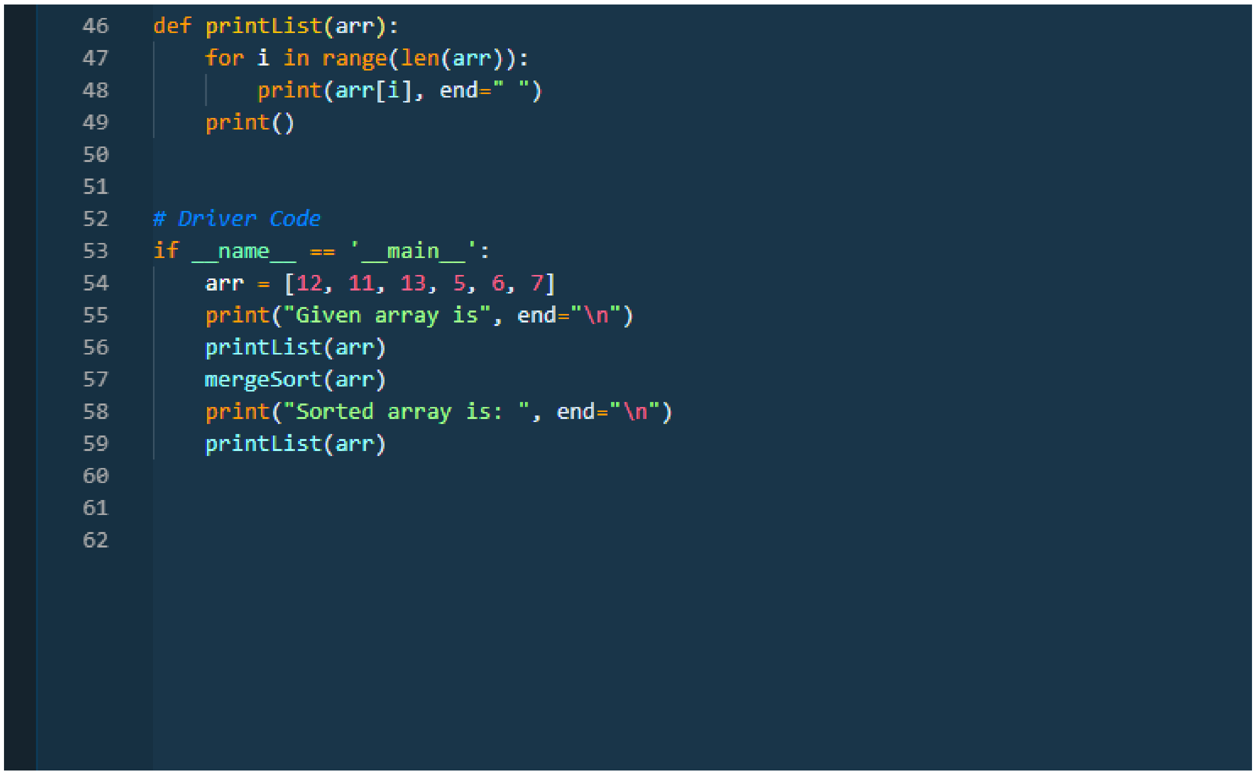
Source code:

```python
# Python program for implementation of MergeSort By Md Nayeem Molla 381
def mergeSort(arr):
    if len(arr) > 1:

        # Finding the mid of the array
        mid = len(arr)//2

        # Dividing the array elements
        L = arr[:mid]

        # into 2 halves
        R = arr[mid:]

        # Sorting the first half
        mergeSort(L)

        # Sorting the second half
        mergeSort(R)

        i = j = k = 0

        # Copy data to temp arrays L[] and R[]
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        # Checking if any element was left
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1

# Code to print the list


def printList(arr):
    for i in range(len(arr)):
        print(arr[i], end=" ")
    print()


# Driver Code
if __name__ == '__main__':
    arr = [12, 11, 13, 5, 6, 7]
    print("Given array is", end="\n")
    printList(arr)
    mergeSort(arr)
    print("Sorted array is: ", end="\n")
    printList(arr)
```

Input:

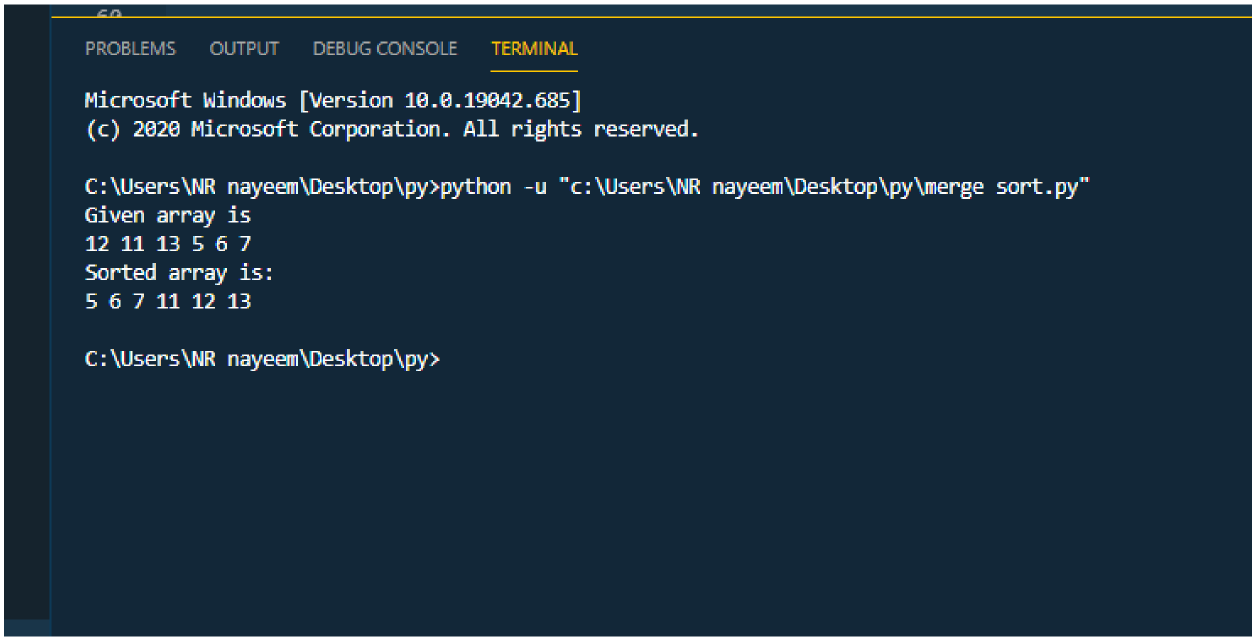


```python
# Python program for implementation of MergeSort By Md Nayeem Molla 381
def mergeSort(arr):
    if len(arr) > 1:

        # Finding the mid of the array
        mid = len(arr)//2

        # Dividing the array elements
        L = arr[:mid]

        # into 2 halves
        R = arr[mid:]

        # Sorting the first half
        mergeSort(L)

        # Sorting the second half
        mergeSort(R)

        i = j = k = 0

        # Copy data to temp arrays L[] and R[]
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        # Checking if any element was left
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1

# Code to print the list


def printList(arr):
    for i in range(len(arr)):
        print(arr[i], end=" ")
    print()
```



```python
def printList(arr):
    for i in range(len(arr)):
        print(arr[i], end=" ")
    print()


# Driver Code
if __name__ == '__main__':
    arr = [12, 11, 13, 5, 6, 7]
    print("Given array is", end="\n")
    printList(arr)
    mergeSort(arr)
    print("Sorted array is: ", end="\n")
    printList(arr)
```

Output:



```
Microsoft Windows [Version 10.0.19042.685]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\NR nayeem\Desktop\py>python -u "c:\Users\NR nayeem\Desktop\py\merge sort.py"
Given array is
12 11 13 5 6 7
Sorted array is:
5 6 7 11 12 13

C:\Users\NR nayeem\Desktop\py>
```

Efficiency analysis and Running time test

Insertion sort

Time Complexity: O(n*2)

Auxiliary Space: O(1)

Boundary Cases: Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

Algorithmic Paradigm: Incremental Approach

Sorting in Place: Yes

Stable: Yes

Online: Yes

Uses: Insertion sort is used when number of elements is small. It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array.

Bubble Sort

Worst and Average Case Time Complexity: O(n*n). Worst case occurs when array is reverse sorted.

Best Case Time Complexity: O(n). Best case occurs when array is already sorted.

Auxiliary Space: O(1)

Boundary Cases: Bubble sort takes minimum time (Order of n) when elements are already sorted.

Quick Sort

Analysis of Quick Sort
Time taken by Quick Sort in general can be written as following.

$T(n) = T(k) + T(n-k-1) + \quad (n)$

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot.
The time taken by Quick Sort depends upon the input array and partition strategy. Following are three cases.

Worst Case: The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case.

$T(n) = T(0) + T(n-1) + \quad (n)$

which is equivalent to

$T(n) = T(n-1) + \quad (n)$

The solution of above recurrence is $\quad (n^2)$.

Best Case: The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case.

$T(n) = 2T(n/2) + \quad (n)$

The solution of above recurrence is $\quad$ (nLogn). It can be solved using case 2 of Master Theorem.

Average Case:
To do average case analysis, we need to consider all possible permutation of array and calculate time taken by every permutation which doesn't look easy.
We can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set. Following is recurrence for this case.

$T(n) = T(n/9) + T(9n/10) + \quad (n)$

Solution of above recurrence is also O(nLogn)

Selection sort:

Time Complexity: O(n^2) as there are two nested loops.

Auxiliary Space: O(1)
The good thing about selection sort is it never makes more than O(n) swaps and can be useful when memory write is a costly operation.

Heap sort:

Notes: I think
Heap sort is an in-place algorithm.
Its typical implementation is not stable, but can be made stable (See this)

Time Complexity: Time complexity of heapify is O(Logn). Time complexity of create And Build Heap () is O(n) and overall time complexity of Heap Sort is O(nLogn).

Two-way Merge sort

Time Complexity: Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.
$T(n) = 2T(n/2) + \theta(n)$

The above recurrence can be solved either using the Recurrence Tree method or the Master method. It falls in case II of Master Method and the solution of the recurrence is θ(nLogn). Time complexity of Merge Sort is θ(nLogn) in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.
Auxiliary Space: O(n)
Algorithmic Paradigm: Divide and Conquer
Sorting in Place: No in a typical implementation
Stable: Yes

Conclusion: I write all code in vs code in my laptop