Heap Sort

```
def heapify (arr, n, i):
        largest = i
        left = 2*i+1
        right = 2*i+2

    if left < n and arr[left] > arr [largest]:
        largest = left

    if right < n and arr[right] > arr [largest]:
            largest = right

    if largest != i:
        arr [i], arr[largest] = arr [largest], arr[i

        heapify (arr, n, largest)

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

    for i in range (n //2-1, -1, -1):
            heapify (arr, n, i)

    for i in range (n-1, 0, -1):
            arr[i], arr[0] = arr[0], arr[i]
```

```
heapify (arr, i, 0)

arr = [12, 11, 13, 5, 6, 7]
heap_sort(arr)
print(" Sorted array :", arr)
```

Time Complexity :

- Heapify : $O(\log n)$
- Building the Heap : $O(n)$
- Sorting Process : $O(n \log n)$

Overall, time Complexity is $O(n \log n)$

Space Complexity :

Heap Sort is an in-place sorting algorithm, so the space complexity is $O(1)$

Q2 Depth First Search (DFS)

```
def dfs (graph, start, visited = None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start, end = '')
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)

graph = {
    'A' : ['B', 'C'],
    'B' : ['D', 'E'],
    'C' : ['F'],
    'D' : [],
    'E' : ['F']
    'F' : []
}
dfs(graph, 'A')
```

Time Complexity (DFS):

- DFS visits every vertex & every edge once
- TC = O(V+E) where V is the number of vertices and E is the number of edges

Space Complexity (DFS)

- The space complexity is O(V) due to the recursion stack in the worst case (in case of deep recursion).

Breadth First Search (BFS)

```
from collections import deque
  def bfs (graph, start):
        visited = set()
      queue = deque ([start])
      visted.add (start)
  ~~while~~
  while queue:
      vertex = queue.popleft()
      print (vertex, end = ' ')

     for neighbor in graph [vertex]:
         if neighbor not in visited:
             visited.add (neighbor)
             queue. append (neighbor)
graph = {
    'A' : ['B', 'C'],
    'B' : ['D', 'E'],
    'C' : ['F'],
    'D' : [],
    'E' : ['F'],
    'F' : []
  }
  bfs (graph, 'A')
```

Time Complexity (BFS) :
- BFS also visits every vertex & every edge once.

Time Complexity : $O(V+E)$

Space Complexity (BFS) :

BFS uses a queue, so the space complexity is $O(V)$ in the worst case.

Q3 Merge Sort

```
def merge_sort (arr):
    if len(arr) > 1:
    mid = len (arr) //2
    left_half = arr[:mid]
    right_half = arr[mid :]

    merge_sort (left_half)
    merge_sort (right_half)
     i = j = k = 0
    while i < len (left_half) and j < len(right_half):
        if left_half[i] < right_half[j]:
            arr[k] = left_half[i]
            i += 1.
        else:
            arr[k] = right_half [j]
              j += 1
            k += 1
    while i < len (left_half):
        arr[k] = left_half [i]
            i = i+1
            k += 1
    while j < len (right_half):
        arr [k] = right_half[j]
            j += 1
            k += 1

arr = [12, 11, 13, 5, 6, 7]
merge_sort (arr)
print("Sorted array:" arr)
```

PAGE NO.:
DATE:

Time Complexity

- Splitting the array : $O(\log n)$
- Merging process : $O(n)$

Thus, the overall time complexity of Merge Sort $O(n\log n)$

Space Complexity

Merge Sort require $O(n)$ auxilary space for the temporary arrays used during the merge process.