Heap Sort

a)Algorithm of heap-sort

```
MaxHeap( arr[1..n], i, n)
{
        int larg = i
        int left =2 * i + 1
        int right = 2 * i + 2
        if (left < n && arr[left] > arr[larg]
        {
             larg = left
         }
        if (right < n && arr[right] > arr[larg]
        {
             larg = right
         }
         If (larg != i)
         {
               Swap( arr[i] , arr[larg])
                MaxHeap(arr[], larg, n)
         }
}
BuildMaxHeap(arr[1..n])
{
      n = length(arr[])
```

```
        for i = n/2 - 1 to 0
        {
                MaxHeap(arr[], i, n)
        }
}
HeapSort(arr[1..n])
{
      BuildMaxHeap(arr[])
      n = length(arr[])
      for i = n - 1 to 1
                swap( A[0] , A[i])
                 n--
                MaxHeap( arr[], 0, n)
}
```

b)analysis of the Algorithm

Max-Heapify :

Time Complexity: O(log(n))

Space Complexity: O(1)

Build-Max-Heap:

Time Complexity: O(n)

Space Complexity: O(1)

Heap-Sort:

Time Complexity: O(nlog(n))

Space Complexity: O(1)

Kruskal's Algorithm for Minimum Spanning Tree (MST)

a)Algorithm of Kruskal's

```
Find(u):

{
   if parent[u] != u:
      parent[u] = Find(parent[u])  // Path compression
   return parent[u]
}


Union(u, v):

{
   root_u = Find(u)
   root_v = Find(v)

   if root_u != root_v:  // Only union if they are in different sets
     { if rank[root_u] > rank[root_v]:
          parent[root_v] = root_u
      else if rank[root_u] < rank[root_v]:
         parent[root_u] = root_v
      else:
        {  parent[root_v] = root_u
         rank[root_u] += 1}

     }
}


Kruskal(n, edges):

{
   // n: Number of vertices
   // edges: List of edges represented as (weight, u, v)
```

```
Sort edges by weight

// Initialize Disjoint Set for n nodes
parent = [0, 1, 2, ..., n-1]
rank = [0, 0, 0, ..., 0]

mst_edges = []
mst_weight = 0

for each (weight, u, v) in edges:
   // If u and v are in different sets, add edge to MST
   if Find(u) != Find(v):
      Union(u, v)  // Merge the sets of u and v
      mst_edges.append((u, v, weight))  // Add edge to MST
      mst_weight += weight  // Add weight to total MST weight

   // If the MST has n-1 edges, we're done
   if length of mst_edges == n - 1:
      break

return mst_edges, mst_weight
```

b)Analysis

find() , union() take O(logn)

Sorting the edges take O(E logE) time, where E is the number of edges

$$T(n) = O(E \log E)$$