Programming Assignment

(a): Algorithms for Heap-Sort

1. **Build Max-Heap**
   Convert an unsorted array into a max-heap, where the parent node is greater than or equal to its child nodes.
2. **Heapify (Max-Heapify)**
   Ensure that the heap property holds for a subtree rooted at a given index.
3. **Heap-Sort Algorithm**
   Repeatedly extract the maximum element from the heap and rebuild the heap for the remaining elements.

(b): Analysis of the Algorithms

1. **Time Complexity**:
   - **Heapify**: O(logn) (each call fixes the heap property for a subtree).
   - **Build Max-Heap**: O(n) (because `Heapify` is applied to n/2 nodes with decreasing subtree sizes).
   - **Heap-Sort**: O(nlogn) (looping n times and calling `Heapify` on decreasing heap sizes).
2. **Space Complexity**:
   - In-place sorting, so the space complexity is O(1).
3. **Key Insights**:
   - The heap property ensures that the maximum element is always at the root, facilitating efficient extraction.
   - Unlike QuickSort, Heap-Sort guarantees O(nlogn) in all cases, making it more predictable.

(c): Implementation

Code in next file

(a): Kruskal’s Algorithm

1. **Union-Find (Disjoint Set Union, DSU)**:
   - This data structure helps efficiently manage the connected components of the graph.
   - It supports two main operations:
     - **Find**: Determines the root or representative of the component containing a node.
     - **Union**: Merges two components into one.
2. **Kruskal’s Algorithm**:
   - **Sort all edges by weight**: Sort the edges of the graph in non-decreasing order of their weights.
   - **Iterate through the edges**: For each edge, check if the vertices it connects belong to the same component. If not, add the edge to the MST and union the two components.
   - **Termination**: Stop when the MST contains exactly `n-1` edges (where `n` is the number of vertices in the graph).

b. Analysis of the Written Algorithms in Part (a)

1. **Union-Find Algorithm (Disjoint Set Union)**:
   - **Initialization**:
     - We initialize a `parent` array where each node is its own parent initially.
     - We also initialize a `rank` array to keep track of the tree depth, used for **union by rank**.
   - **Find Operation**:
     - **Path Compression**: This technique flattens the tree structure by making each node in the path point directly to the root, improving the time complexity of subsequent `find` operations.
   - **Union Operation**:
     - **Union by Rank**: The tree with the smaller rank is attached under the root of the tree with the larger rank. This helps keep the trees flat and improves the efficiency of the `find` operation.

   Time Complexity:

   - The **find** operation has an almost constant time complexity, specifically **O(α(n))**, where α is the inverse Ackermann function, which grows extremely slowly.
   - The **union** operation also runs in **O(α(n))**.
2. **Kruskal’s Algorithm**:
   - **Edge Sorting**: We need to sort the edges by their weight, which requires **O(E log E)** time complexity, where `E` is the number of edges.
   - **Iterating through edges**: For each edge, we perform a `find` and `union` operation, each of which has **O(α(n))** complexity. Thus, processing all edges takes

     O(E α(n)) time.

Overall Time Complexity:

- **O(E log E + E α(n)) = O(E log E)**, as **α(n)** grows very slowly and can be treated as a constant for practical purposes.

3. **Edge Case Considerations**:
    - If the graph is disconnected, Kruskal's algorithm will only return the MST for the connected components and will not connect the entire graph.
    - The algorithm assumes the graph is undirected.
    - Kruskal's algorithm does not work if the graph contains negative weight cycles or if there are duplicate edges with the same weight.

(c): Implementation

Code in next file