Algorithms project

Q.1

Part (b): Algorithm Analysis

1. **Time Complexity**:
    - Building the max heap takes (O(n)) time.
    - The heapify operation takes (O(\log n)) time, and since we perform this operation (n) times (once for each element), the total time complexity for the sorting phase is (O(n \log n)).
    - Therefore, the overall time complexity of Heap-Sort is (O(n \log n)).
2. **Space Complexity**:
    - Heap-Sort is an in-place sorting algorithm, meaning it requires a constant amount of additional space (O(1)) for the sorting process. However, the recursive calls to **heapify** may use (O(\log n)) space on the call stack in the worst case.
3. **Stability**:
    - Heap-Sort is not a stable sorting algorithm. Stability means that two equal elements retain their relative order after sorting. In Heap-Sort, this is not guaranteed.
4. **Use Cases**:
    - Heap-Sort is useful when you need a guaranteed (O(n \log n)) time complexity and when memory usage is a concern, as it does not require additional space for another array.

Q.2

Part (b): Algorithm Analysis

1. **Time Complexity**:
    - Sorting the edges takes (O(E \log E)), where (E) is the number of edges.
    - Each union and find operation takes nearly constant time, (O(\alpha(V))), where (\alpha) is the inverse Ackermann function, which grows very slowly. Thus, for (E) edges, the total time for union-find operations is (O(E \alpha(V))).
    - Therefore, the overall time complexity of Kruskal's algorithm is (O(E \log E)).

2. **Space Complexity**:
    - The space complexity is (O(V + E)) for storing the edges and the union-find structure, where (V) is the number of vertices.
3. **Correctness**:
    - Kruskal's algorithm is correct because it always adds the smallest edge that does not form a cycle, ensuring that the resulting tree is a minimum spanning tree.
4. **Use Cases**:
    - Kruskal's algorithm is particularly useful for sparse graphs where the number of edges is much less than the number of vertices squared.