analyze of Kruskal's algorithm

1. Data Structures:

- **edge[n][3]:** This 2D array represents the edges of the graph. Each row in the array represents an edge, with:
    - edge[i][0]: The source vertex of the edge.
    - edge[i][1]: The destination vertex of the edge.
    - edge[i][2]: The weight of the edge.
- **parent[n]:** An array to store the parent of each vertex in the Minimum Spanning Tree. Initially, each vertex is its own parent.
- **rank[n]:** An array to store the rank of each vertex in the union-find data structure, used to optimize the union operation.

2. Algorithm Steps:

- **Sorting:** The edges are first sorted in ascending order of their weights using the qsort function and a custom comparator function. This ensures that edges are considered in increasing order of weight.
- **Initialization:** The makeSet function initializes the parent and rank arrays for each vertex, setting each vertex as its own parent and rank to 0.
- **MST Construction:**
    - For each edge in the sorted list:
        - Find the parent of the source and destination vertices using the findParent function.
        - If the parents are different, it means the vertices belong to different components.
        - Union the two components using the unionSet function.
        - Add the weight of the edge to the minCost.
- **Output:**
    - Print the edges included in the MST.
    - Print the total minimum cost of the MST.

3. Time Complexity:

- **Sorting:** Sorting the edges using qsort takes O(E log E) time, where E is the number of edges.
- **Initialization:** The makeSet function takes O(V) time, where V is the number of vertices.
- **Union-Find:** The findParent and unionSet operations, when implemented with path compression and union by rank, have amortized time complexity of O(α(V)), where α is the inverse Ackermann function, which grows extremely slowly. Therefore, the total time complexity of the union-find operations is O(E α(V)).

Overall Time Complexity:

Since E (number of edges) is usually proportional to V^2 in a graph, the overall time complexity of Kruskal's algorithm is **O(E log E)** or **O(V^2 log V)**.

4. Space Complexity:

The space complexity of Kruskal's algorithm is O(V + E) to store the graph, parent array, and rank array.

analyze for heap sort.

1. Algorithm:

The code implements the heap sort algorithm, which sorts an array in ascending order by using a binary heap data structure.

2. Key Steps:

- **Heapify:**
  - The heapify function is the core of the algorithm.
  - It ensures that the subtree rooted at a given node satisfies the heap property (parent node is greater than or equal to its children).
  - It recursively sifts down the node until the heap property is maintained.
- **Building the Heap:**
  - The heapSort function starts by building a max-heap from the given array.

    - It iterates through the array from the last non-leaf node (n/2 - 1) to the root (0) and calls heapify for each node.

- **Sorting:**
    - The heapSort function then repeatedly extracts the maximum element (root) from the heap and swaps it with the last element of the unsorted portion.
    - The size of the heap is reduced by one after each extraction, and the heap is re-heapified to maintain the heap property.

3. Time Complexity:

- **Building the Heap:** Building the initial heap takes O(n) time.
- **Extracting and Re-heapifying:** In each iteration, extracting the maximum element takes O(1) time, and re-heapifying takes O(log n) time. This is repeated n-1 times, resulting in a total time complexity of O(n log n).

4. Space Complexity:

The space complexity of heap sort is O(1) as it sorts the array in-place, without requiring any additional data structures.

5. Correctness:

The code correctly implements the heap sort algorithm and produces the expected sorted output.