Part (i): Heapsort Algorithm

a. Write all required algorithms needed to sort a sequence of numbers using Heapsort

Here is a JavaScript implementation of the Heapsort algorithm:

```
function heapsort(arr) {
  const n = arr.length;

  // Build a max heap
  for (let i = Math.floor(n / 2) - 1; i >= 0; i--) {
    heapify(arr, n, i);
  }

  // One by one extract elements from heap
  for (let i = n - 1; i > 0; i--) {
    // Move current root to end
    [arr[0], arr[i]] = [arr[i], arr[0]];
    // Call max heapify on the reduced heap
    heapify(arr, i, 0);
  }

  return arr;
}

function heapify(arr, n, i) {
  let largest = i; // Initialize largest as root
```

```
  const left = 2 * i + 1; // left = 2*i + 1
  const right = 2 * i + 2; // right = 2*i + 2

  // If left child is larger than root
  if (left < n && arr[left] > arr[largest]) {
    largest = left;
  }

  // If right child is larger than largest so far
  if (right < n && arr[right] > arr[largest]) {
    largest = right;
  }

  // If largest is not root
  if (largest !== i) {
    [arr[i], arr[largest]] = [arr[largest], arr[i]]; // Swap

    // Recursively heapify the affected sub-tree
    heapify(arr, n, largest);
  }
}

// Example usage
const arr = [12, 11, 13, 5, 6, 7];
console.log("Sorted array is:", heapsort(arr));
```

b. Analyze in detail your written algorithms in Part (a)

Explanation

1. Heap Construction:

- The function begins by building a max heap from the input array. A max heap is a binary tree where the parent node is always greater than or equal to its child nodes. This is done by calling the heapify function for each non-leaf node in reverse level order, starting from the last non-leaf node down to the root.

2. Heapify Function:

- The heapify function ensures that the subtree rooted at index i satisfies the heap property. It checks whether the left or right child is larger than the current node (root). If either child is larger, it swaps the nodes and recursively calls heapify on the affected subtree.

3. Sorting Process:

- After building the max heap, the sorting process begins. The largest element (root of the heap) is swapped with the last element of the array, reducing the size of the heap by one. The heapify function is then called on the reduced heap to maintain the heap property.

4. Repeat:

- This process is repeated until the size of the heap is reduced to one. At this point, the array is sorted in ascending order.

5. Return Value:

- The sorted array is returned after all elements have been processed. The algorithm operates in-place, meaning it requires a constant amount of additional space. The overall time complexity is O(n log n), making it efficient for large datasets.

c. Implement your written algorithms in Part (a)

The implementation is already provided in the code

Part (ii): Kruskal's Algorithm for Minimum Spanning Tree (MST)

a. Write all required algorithms needed to find MST using Kruskal's Algorithm

Here’s a JavaScript implementation of Kruskal's algorithm:

```
class DisjointSet {
  constructor(n) {
    this.parent = new Array(n);
    this.rank = new Array(n).fill(0);
    for (let i = 0; i < n; i++) {
      this.parent[i] = i;
    }
  }

  find(u) {
    if (this.parent[u] !== u) {
      this.parent[u] = this.find(this.parent[u]); // Path compression
    }
    return this.parent[u];
  }

  union(u, v) {
    const rootU = this.find(u);
    const rootV = this.find(v);

    if (rootU !== rootV) {
      // Union by rank
      if (this.rank[rootU] > this.rank[rootV]) {
        this.parent[rootV] = rootU;
      } else if (this.rank[rootU] < this.rank[rootV]) {
        this.parent[rootU] = rootV;
```

```
        } else {
          this.parent[rootV] = rootU;
          this.rank[rootU]++;
        }
      }
    }
}

function kruskal(vertices, edges) {
    // Sort edges by weight
    edges.sort((a, b) => a.weight - b.weight);

    const ds = new DisjointSet(vertices);
    const mst = [];

    for (const edge of edges) {
      const { u, v, weight } = edge;
      // If u and v are in different sets, include this edge in the MST
      if (ds.find(u) !== ds.find(v)) {
        ds.union(u, v);
        mst.push(edge);
      }
    }

    return mst;
}

// Example usage
const vertices = 4;
```

```
const edges = [
  { u: 0, v: 1, weight: 10 },
  { u: 0, v: 2, weight: 6 },
  { u: 0, v: 3, weight: 5 },
  { u: 1, v: 3, weight: 15 },
  { u: 2, v: 3, weight: 4 }
];

console.log("Minimum Spanning Tree:", kruskal(vertices, edges));
```

b. Analyze in detail your written algorithms in Part (a)

. Disjoint Set (Union-Find):

- A data structure to manage a partition of a set into disjoint subsets. It supports two main operations:
  - Find: Determines which subset a particular element is in. Uses path compression for efficiency.
  - Union: Merges two subsets into a single subset. Uses union by rank to keep the tree flat.

2. Kruskal's Algorithm:

- Input: A list of edges, each with two vertices and a weight, and the number of vertices.
- Sorting Edges: The edges are sorted in non-decreasing order based on their weights.
- Constructing MST: The algorithm iterates through the sorted edges, and for each edge, it checks if the vertices belong to different sets using the find operation. If they do, it adds the edge to the MST and merges the sets using the union operation.
- The process continues until enough edges have been added to the MST (specifically, vertices - 1 edges).

3. Output:

- The function returns the edges that comprise the Minimum Spanning Tree.