Implementation of Heap-Sort

```
def heap_sort(arr):
    # Function to maintain the max-heap property
    def max_heapify(arr, n, i):
        largest = i
        left = 2 * i + 1
        right = 2 * i + 2

        if left < n and arr[left] > arr[largest]:
            largest = left
        if right < n and arr[right] > arr[largest]:
            largest = right
        if largest != i:
            arr[i], arr[largest] = arr[largest], arr[i]
            max_heapify(arr, n, largest)

# Function to build the max-heap
def build_max_heap(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        max_heapify(arr, n, i)

# Main heap sort logic
n = len(arr)
    build_max_heap(arr)
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        max_heapify(arr, i, 0)

# Example usage
if name == "__main__":
    array = [4, 10, 3, 5, 1]
    print("Original array:", array)
    heap_sort(array)
    print("Sorted array:", array)
```

Implementation of Kruskal`s Algorithm

```
class DisjointSet:
    def __init__(self, vertices):
        # Initialize parent and rank dictionaries
        self.parent = {v: v for v in vertices}
        self.rank = {v: 0 for v in vertices}

    def find(self, node):
        # Find the root of the set with path compression
        if self.parent[node] != node:
            self.parent[node] = self.find(self.parent[node])
        return self.parent[node]

    def union(self, u, v):
        # Union by rank
        root_u = self.find(u)
        root_v = self.find(v)

        if root_u != root_v:
            if self.rank[root_u] > self.rank[root_v]:
                self.parent[root_v] = root_u
            elif self.rank[root_u] < self.rank[root_v]:
                self.parent[root_u] = root_v
            else:
                self.parent[root_v] = root_u
                self.rank[root_u] += 1


def kruskal(vertices, edges):
    # Initialize the Minimum Spanning Tree (MST)
    mst = []
    dsu = DisjointSet(vertices)

    # Sort edges by weight
    edges.sort(key=lambda x: x[2])  # Each edge is (u, v, weight)

    for u, v, weight in edges:
        # Check if the current edge forms a cycle
        if dsu.find(u) != dsu.find(v):
            dsu.union(u, v)
```

```
            mst.append((u, v, weight))

    return mst


# Example usage
vertices = ['A', 'B', 'C', 'D', 'E']
edges = [
    ('A', 'B', 1),
    ('A', 'C', 5),
    ('B', 'C', 2),
    ('B', 'D', 4),
    ('C', 'D', 6),
    ('D', 'E', 3)
]

mst = kruskal(vertices, edges)
print("Minimum Spanning Tree:", mst)
```