**Part (a): Write all required algorithms**

As mentioned before, we need these three core functions for a standard Heapsort implementation:

1. **heapify(array, n, i) (also known as siftDown or maxHeapify)**
   * **Purpose:** This function takes an array (representing a binary tree), its size (n), and an index i. It assumes the subtrees rooted at the children of i are already valid max-heaps. It then ensures that the subtree rooted at i is a valid max-heap by "sifting down" the element at index i to its correct position.
   * **Algorithm (Pseudocode):**
   * function heapify(array, n, i):
   * largest = i // Initialize largest as the root (i)
   * left = 2\*i + 1 // Calculate the left child
   * right = 2\*i + 2 // Calculate the right child
   * // Check if the left child is larger than root
   * if left < n and array[left] > array[largest]:
   * largest = left
   * // Check if the right child is larger than largest so far
   * if right < n and array[right] > array[largest]:
   * largest = right
   * // If the largest is not the root, swap and heapify the affected subtree
   * if largest != i:
   * swap array[i] and array[largest]
   * heapify(array, n, largest) //Recursively heapify the affected sub tree

content\_copy download

Use code [with caution](https://support.google.com/legal/answer/13505487).Pseudocode

* + **Key Points:**
    - We assume a 0-based array indexing.
    - It uses a recursive approach to ensure the heap property is maintained. If you don't prefer recursion, you can also write this algorithm using an iterative approach using a while loop.
    - The "max-heap" property is that the parent is always larger than its children. If a min-heap is required, simply change the greater than symbol > to less than symbol <.

1. **buildHeap(array, n)**
   * **Purpose:** This function converts a given array into a max-heap.
   * **Algorithm (Pseudocode):**
   * function buildHeap(array, n):
   * // Start from the last non-leaf node (parent of the last node)
   * for i from floor(n/2) - 1 downto 0:
   * heapify(array, n, i)

content\_copy download

Use code [with caution](https://support.google.com/legal/answer/13505487).Pseudocode

* + **Key Points:**
    - The last non-leaf node in a 0-based array is at index floor(n/2) - 1.
    - We iterate from the last non-leaf node up to the root (index 0), calling heapify at each node. This builds the max-heap from the bottom up.

1. **heapsort(array, n)**
   * **Purpose:** Sorts the given array in ascending order using the heap.
   * **Algorithm (Pseudocode):**
   * function heapsort(array, n):
   * buildHeap(array, n) //Build max-heap from the array
   * for i from n-1 downto 1:
   * swap array[0] and array[i] // Swap the root (largest) with the last element in the unsorted portion of the array
   * heapify(array, i, 0) // Heapify the root of the reduced heap size (excluding the sorted portion)

content\_copy download

Use code [with caution](https://support.google.com/legal/answer/13505487).Pseudocode

* + **Key Points:**
    - First, buildHeap() transforms the array into a max-heap.
    - Then, we extract the largest element (always at the root of the max-heap), place it at the end of the array, and reduce the heap size. Then heapify the new root.
    - Repeat until the whole array is sorted.

**Part (b): Analyze in detail**

Here's a detailed complexity analysis:

* **heapify(array, n, i):**
  + **Time Complexity:**
    - **Best Case:** O(1). If the node at index i is already greater than its children.
    - **Average Case:** O(log n). The height of the subtree is reduced with each step down.
    - **Worst Case:** O(log n). The node may need to travel all the way down to the leaf in the tree.
  + **Space Complexity:**
    - **O(log n)** due to the recursive call stack. If using an iterative implementation, then O(1) or constant space complexity.
* **buildHeap(array, n):**
  + **Time Complexity:**
    - **Best Case:** O(n).
    - **Average Case:** O(n).
    - **Worst Case:** O(n). This can be a bit unintuitive, because heapify takes O(log n), and the loop iterates approximately n/2 times. However, if we analyze the steps of each sift down operation, we will see that most heapify operations are done closer to the bottom of the heap which will take less time, that is why the overall time complexity of buildHeap is O(n) rather than O(n log n)
  + **Space Complexity:**
    - O(log n) due to call stack of heapify in a recursive implementation. If using an iterative implementation, then O(1) or constant space complexity.
* **heapsort(array, n):**
  + **Time Complexity:**
    - **Best Case:** O(n log n).
    - **Average Case:** O(n log n).
    - **Worst Case:** O(n log n).
  + **Space Complexity:**
    - O(log n) due to the call stack of heapify in the recursive implementation, In case an iterative version of heapify is used instead, then the space complexity is O(1).
  + **Overall complexity**
    - **Time Complexity:** O(n log n). The sorting part in heapsort is the major contributor here.
    - **Space Complexity:**
      * O(log n) if heapify is done recursively, and O(1) if done iteratively. This algorithm is considered in-place with no auxiliary data structures, therefore it doesn't require extra space other than the implicit recursion call stack.

**Part (c): Implement your written algorithms**

You will need to choose a programming language and write code that directly follows the pseudocode you created in Part (a). This involves:

1. **Creating functions** for heapify, buildHeap, and heapsort.
2. **Taking an array as input** to the heapsort function.
3. **Sorting the input array.**
4. **Possibly printing the sorted array** or returning it as a result, according to the needs of your instructor.
5. **Testing**: be sure to test the code with various sample data to make sure it is correct and performs the way it should.

**Example Implementation (Python):**

import math

def 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]

heapify(arr, n, largest)

def buildHeap(arr, n):

for i in range(math.floor(n/2) - 1, -1, -1):

heapify(arr, n, i)

def heapsort(arr):

n = len(arr)

buildHeap(arr, n)

for i in range(n - 1, 0, -1):

arr[0], arr[i] = arr[i], arr[0]

heapify(arr, i, 0)

return arr

# Example usage:

array\_to\_sort = [12, 11, 13, 5, 6, 7]

sorted\_array = heapsort(array\_to\_sort)

print("Sorted Array:", sorted\_array)

content\_copy download

Use code [with caution](https://support.google.com/legal/answer/13505487).Python

**Key things to include in your report:**

* The pseudocode for each algorithm.
* A detailed analysis of the time and space complexity.
* The actual implementation in the chosen programming language.
* Testing results.
* Any observations you have about the implementation, or any issues you faced.

Remember that you might want to clarify which heap implementation to use (max-heap or min-heap, although max-heap is standard for ascending order) and about any allowed or disallowed implementations with your instructor before starting your implementation.

**Part (a): Write all required algorithms**

Kruskal's algorithm primarily relies on the following steps: sorting edges by weight and using a Disjoint-Set data structure (also known as Union-Find) to avoid cycles. Thus, we need:

1. **find(parent[], i):** This function finds the root (or representative) of the set to which element i belongs. It's a part of the Disjoint-Set data structure. It can optionally include path compression optimization.
   * **Purpose:** To determine which set a node belongs to.
   * **Algorithm (Pseudocode):**
   * function find(parent[], i):
   * if parent[i] == i: // i is root of this set
   * return i
   * else:
   * // Path compression (optional but improves efficiency)
   * parent[i] = find(parent, parent[i])
   * return parent[i]

content\_copy download

Use code [with caution](https://support.google.com/legal/answer/13505487).Pseudocode

* + **Key Points:**
    - If parent[i] is i itself, then i is the root of that set.
    - The path compression (recursive step) makes the find operation more efficient in the long run by collapsing the paths and making them shorter.
    - Path Compression is not needed to make the algorithm work, but if you don't add it, then your find method would be slower as its time complexity would be O(n) instead of an almost constant time complexity.

1. **union(parent[], rank[], x, y):** This function merges the sets containing elements x and y. It uses "union by rank" for efficiency which is optional, but it helps making the sets' trees balanced (less deep) thus making the find operation more efficient.
   * **Purpose:** To merge two sets.
   * **Algorithm (Pseudocode):**
   * function union(parent[], rank[], x, y):
   * rootX = find(parent, x)
   * rootY = find(parent, y)
   * if rootX != rootY:
   * if rank[rootX] < rank[rootY]:
   * parent[rootX] = rootY
   * elif rank[rootX] > rank[rootY]:
   * parent[rootY] = rootX
   * else:
   * parent[rootY] = rootX
   * rank[rootX] = rank[rootX] + 1

content\_copy download

Use code [with caution](https://support.google.com/legal/answer/13505487).Pseudocode

* + **Key Points:**
    - Finds the roots of the sets containing x and y first.
    - Merges the sets, connecting the root of smaller rank (shorter tree) with the root of larger rank (deeper tree).
    - If ranks are equal, increment the rank of the new root by one.
    - Union by Rank is not needed for the algorithm to work, but without it you can run into the scenario where the find method can have O(n) time complexity instead of an almost constant time complexity.

1. **kruskal(edges[], vertices, n\_edges):** This is the main function that implements Kruskal's algorithm.
   * **Purpose:** To find the Minimum Spanning Tree of the given graph.
   * **Algorithm (Pseudocode):**
   * function kruskal(edges[], vertices, n\_edges):
   * // Sort edges by weight (ascending order)
   * sort edges by weight
   * //Initialize disjoint-set data structure
   * parent = new array of size equal to vertices size, initialize parent[i] = i
   * rank = new array of size equal to vertices size, initialize rank[i] = 0
   * mst\_edges = empty list
   * for edge in edges:
   * u, v, weight = edge // Get the vertices and weight of the current edge
   * if find(parent, u) != find(parent, v): //Check if adding this edge will make a cycle, if so don't add it
   * union(parent, rank, u, v) // Merge the sets if there is no cycle
   * mst\_edges.append(edge) // Add the edge to the MST
   * return mst\_edges // or the total weight of mst edges, according to the requirements.

content\_copy download

Use code [with caution](https://support.google.com/legal/answer/13505487).Pseudocode

* + **Key Points:**
    - We start with an empty MST and no connections between any vertices.
    - We sort the edges based on their weight.
    - We iterate through the sorted edges, and if a certain edge's endpoints are not in the same set (this means adding it won't create a cycle), we add it to MST and merge the sets that the edge's endpoints belong to.
    - We do this process until we go through all of the edges, at the end mst\_edges will have all of the edges that form the Minimum Spanning Tree.

**Part (b): Analyze in detail**

Let's analyze the complexities:

* **find(parent[], i):**
  + **Time Complexity:**
    - **Worst Case:** O(n). occurs when all elements form a single list or skewed tree, where there are no branches, and each find() has to traverse all n nodes until reaching the root node, with no path compression done to enhance the performance of this.
    - **Average/Amortized Case (with path compression):** Almost O(1). Path compression will greatly improve this case time complexity. More precisely, it can be said that the amortized time complexity is O(α(n)) which is the inverse Ackermann function. This function grows so slowly, that it can be considered constant time complexity.
  + **Space Complexity:** O(1), constant space as it uses the existing parent array and stack frame for recursive calls.
* **union(parent[], rank[], x, y):**
  + **Time Complexity:**
    - **Worst Case:** O(n) if the find methods don't use path compression optimization, and there are no optimization applied when merging sets (for example union by size or union by rank), and all elements form a single list where there are no branches, then this will cause the find to traverse through the entire list before merging
    - **Average/Amortized Case (with union by rank and path compression):** O(α(n)). This is because this function uses find function inside it, and uses a constant amount of time to merge the nodes after finding their representative roots.
  + **Space Complexity:** O(1), constant space as it uses the existing parent and rank arrays.
* **kruskal(edges[], vertices, n\_edges):**
  + **Time Complexity:**
    - **Sorting Edges:** O(E log E), where E is the number of edges. This dominates the overall time complexity.
    - **Disjoint Set Operations:** O(E \* α(V)), where V is the number of vertices and E is number of edges, however because α(V) grows so slowly, it is considered almost constant time complexity.
    - **Overall Worst-Case Time Complexity:** O(E log E) because sorting edges takes the majority of the time.
  + **Space Complexity:** O(V) space to store parent, rank arrays and O(E) to store edges if it's needed. The algorithm is considered to be in-place and no new big size array is being created.

**Part (c): Implement your written algorithms**

Implement the pseudocodes from part (a) in your programming language. Make sure to:

1. **Create functions** for find, union, and kruskal.
2. **Choose a way to represent the graph (edges list)**. Edges can be stored in an array or list of tuples or objects representing connections between two nodes with a weight.
3. **Make sure your kruskal function takes the graph representation, vertices size and number of edges as input.**
4. **Sort the edges by weight correctly** in the kruskal function.
5. **Use the find and union operations correctly** in the kruskal function.
6. **Make sure to create the disjoint set data structures parent and rank in the beginning of kruskal method.**
7. **Return the MST** as a list of edges or as the total weight of the MST, according to the needs of your instructor.
8. **Test**: make sure to test your code with different test cases to check if the output is correct.

**Example Implementation (Python):**

def find(parent, i):

if parent[i] == i:

return i

parent

content\_copy download

Use code [with caution](https://support.google.com/legal/answer/13505487).Python