ALGORITHM

(i) Heap sort

What's the heap sorting

Write all required algorithms for Heap-Sort

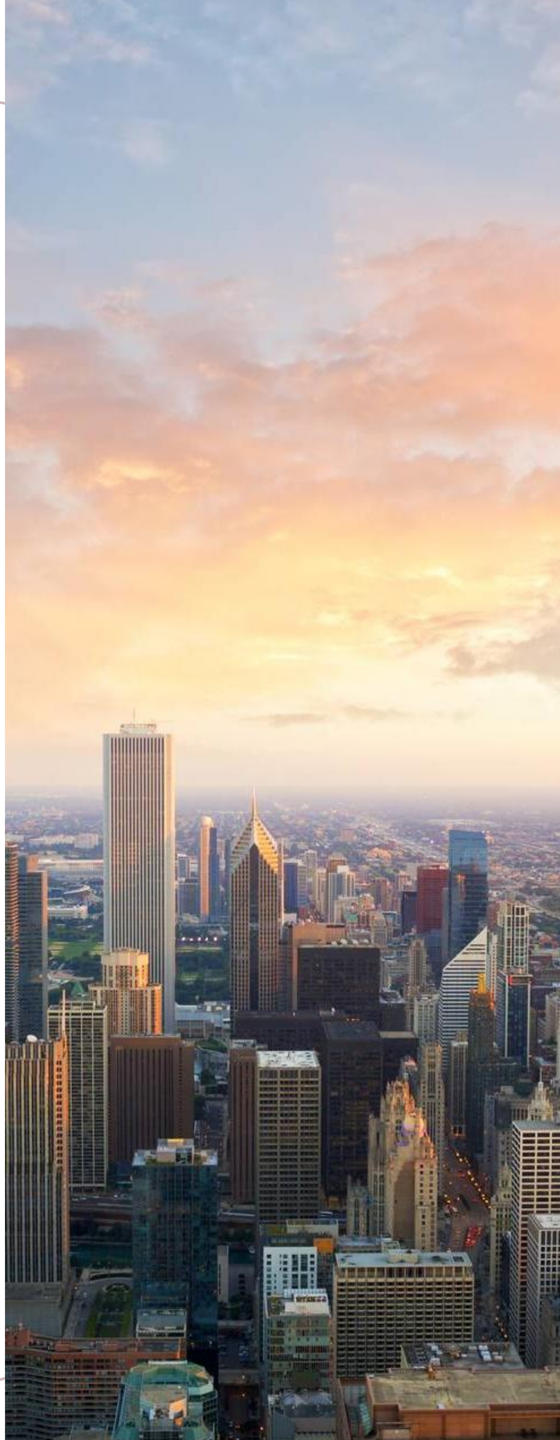
Analyze the written algorithms

(ii) Kruskal's algorithm to find MST of a network.

Kruskal's Algorithm Overview

Write all Required Algorithms:

Analyze the written algorithms



HEAP SORT

OVERVIEW

REQUIRED ALGORITHMS

Heap-Sort Overview

Heap-Sort is a comparison-based sorting algorithm that leverages a binary heap data structure. The key operations are building a max heap and repeatedly extracting the maximum element to sort the array.

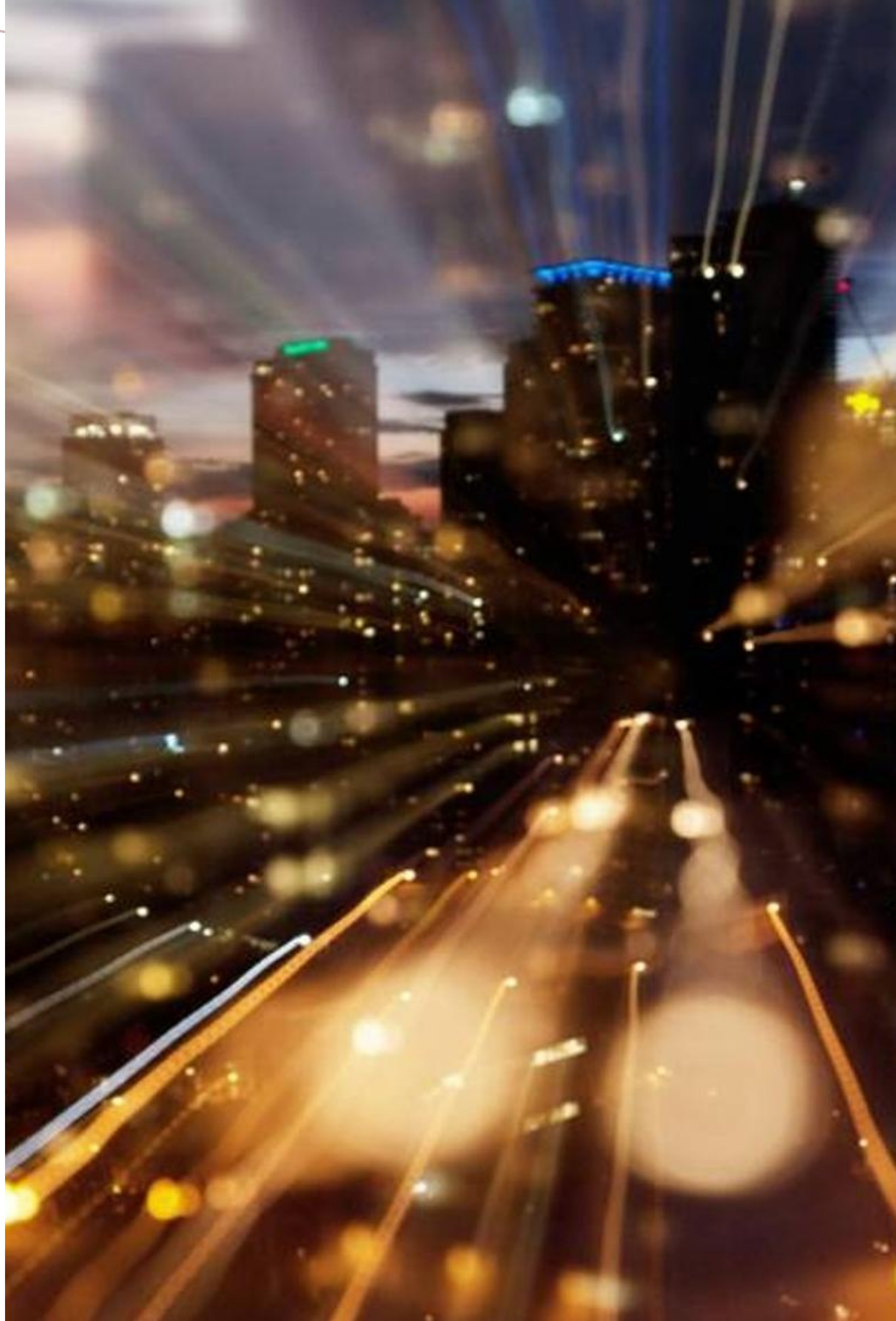
Here are the required algorithms:

1. **Max-Heapify**: Maintains the max-heap property.
2. **Build-Max-Heap**: Converts an array into a max heap.
3. **Heap-Sort**: Sorts an array using the above functions.

Max-heapify algorithm

```
Max-Heapify(A, i, heap_size):
1. left ← 2 * i + 1
2. right ← 2 * i + 2
3. largest ← i
4. If left < heap_size and A[left] > A[largest], then largest ← left
5. If right < heap_size and A[right] > A[largest], then largest ← right
6. If largest ≠ i, then
    a. Swap A[i] and A[largest]
    b. Max-Heapify(A, largest, heap_size)
```

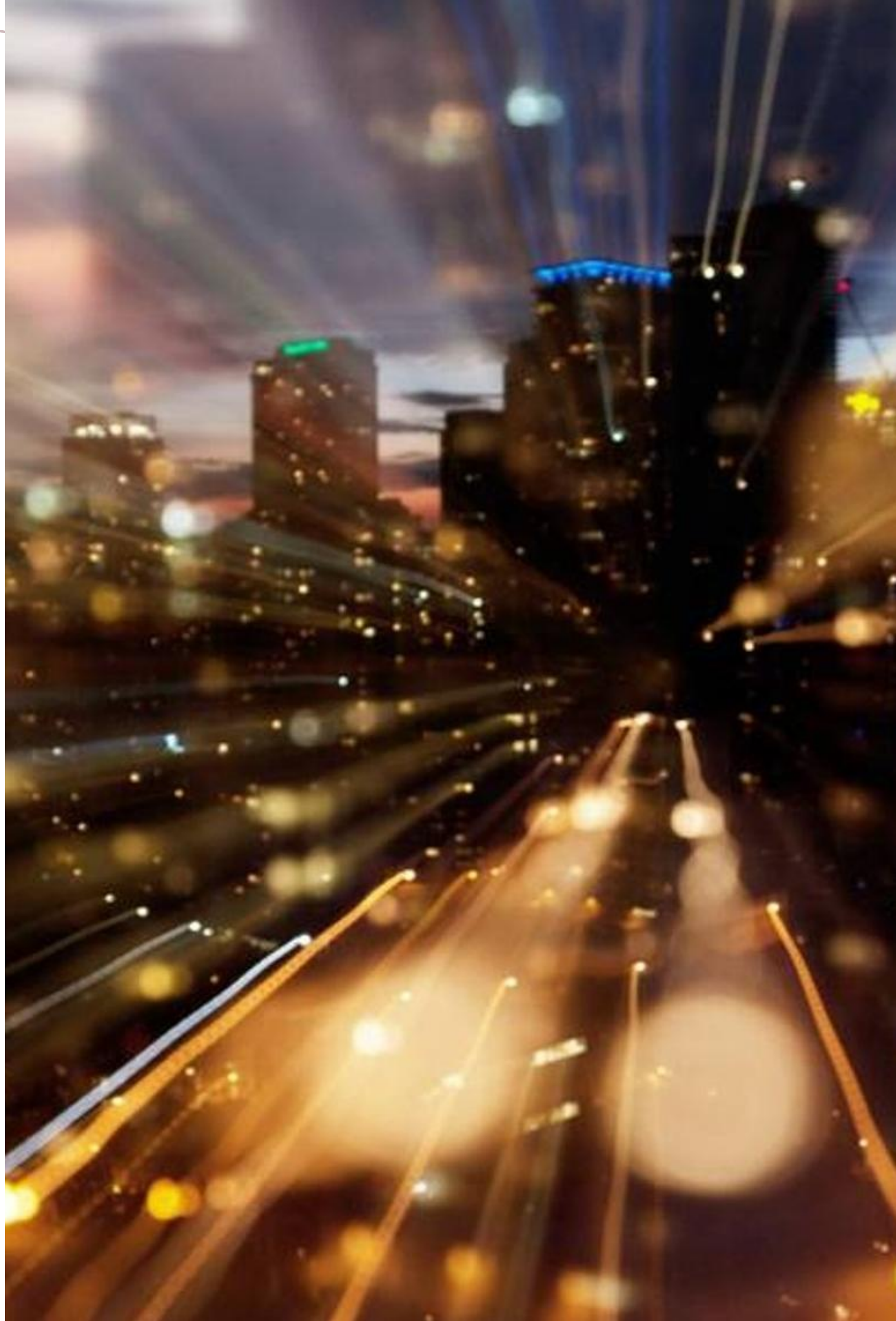
- Time complexity: O(h), where h is proportional to log(n).
- Operates on a single subtree of height h.



Build-max-heap algorithm

```
Build-Max-Heap(A):
1. heap_size ← length(A)
2. For i ← floor(length(A)/2) - 1 down to 0:
    a. Max-Heapify(A, i, heap_size)
```

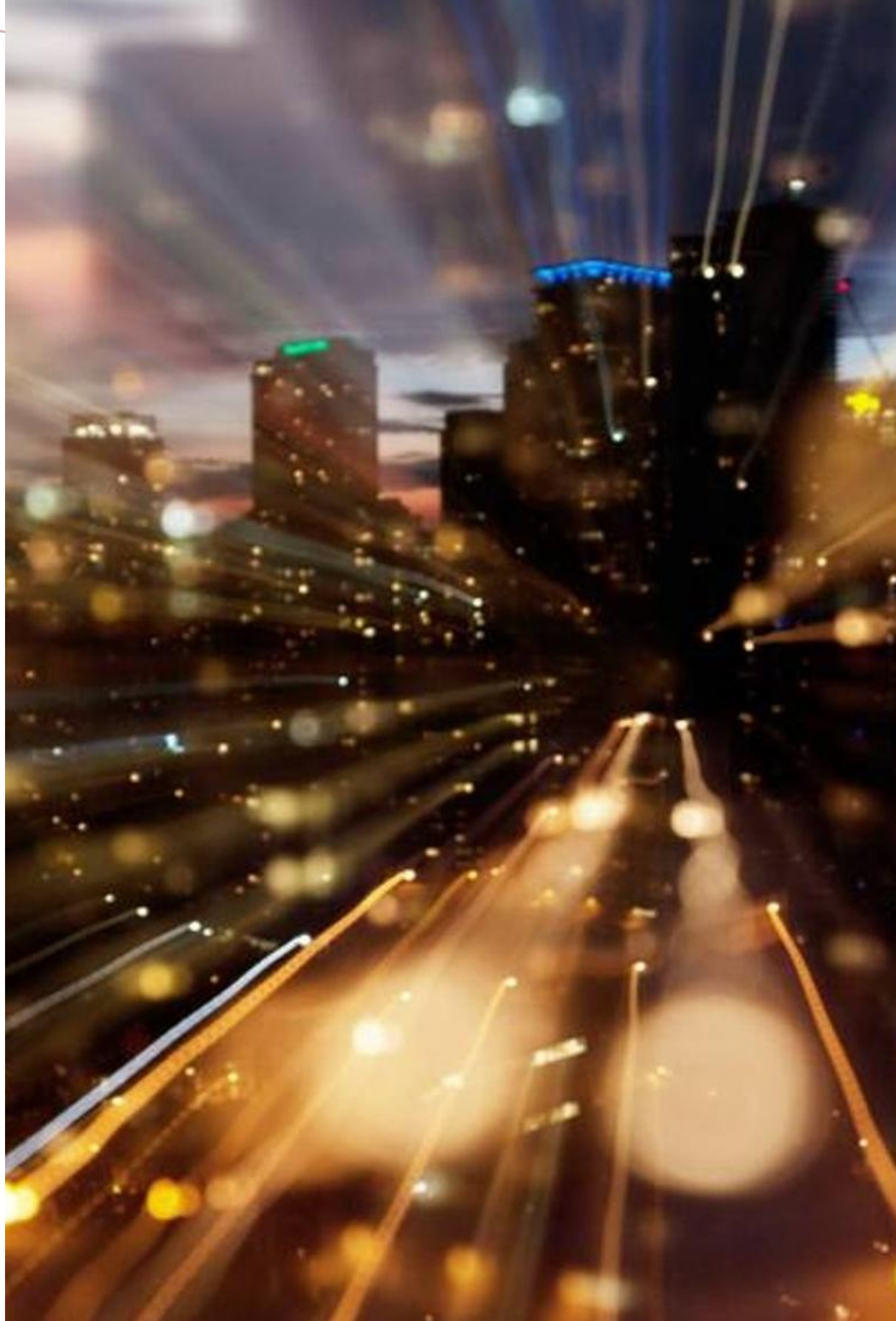
•Runs Max−Heapify on all non-leaf nodes.
•Total time complexity: O(n).



Heap-Sort Algorithm

```
Heap-Sort(A):
1. Build-Max-Heap(A)
2. heap_size ← length(A)
3. For i ← length(A) - 1 down to 1:
    a. Swap A[0] and A[i]
    b. heap_size ← heap_size - 1
    c. Max-Heapify(A, 0, heap_size)
```

- Builds the max heap (O(n)).
- Extracts the maximum n−1 times (O(nlogn)).
- Total time complexity: O(nlogn).



KRUSKAL'S ALGORITHM

OVERVIEW

REQUIRED ALGORITHMS

Overview

Kruskal’s algorithm is a greedy approach to finding the MST of a weighted, connected, and undirected graph. The algorithm works by:

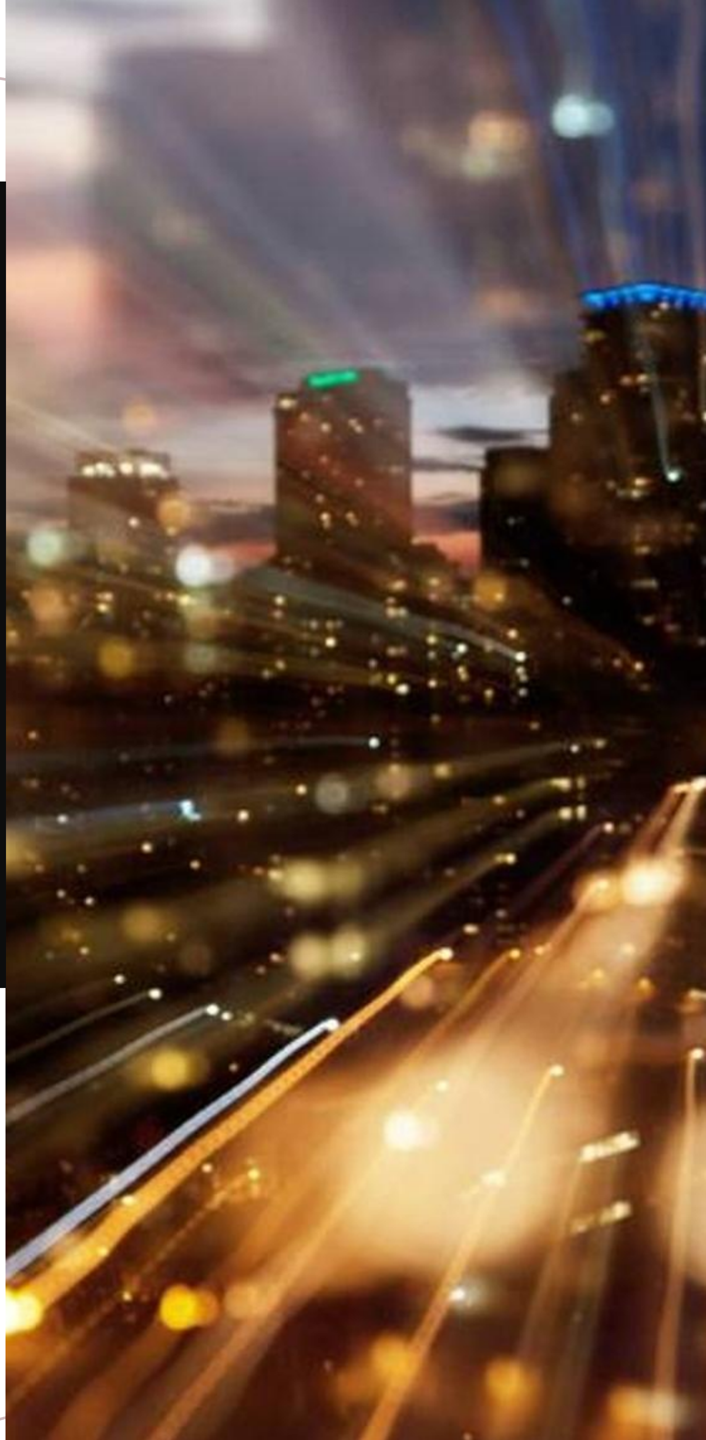
1. Sorting all edges by their weights.
2. Iteratively adding the smallest edge to the MST, ensuring no cycles are formed.

- *Union-Find (Disjoint Set Union):*
- *To check whether adding an edge creates a cycle.*
- *Operations:*
  - Find: Determine the set to which an element belongs.
  - Union: Merge two sets.
- **Kruskal's Algorithm**:
- Uses the Union-Find data structure.

UNION-FIND DATA STRUCTURE

```
Find(parent, x):
1. If parent[x] ≠ x:
     a. parent[x] ← Find(parent, parent[x])
2. Return parent[x]

Union(parent, rank, x, y):
1. rootX ← Find(parent, x)
2. rootY ← Find(parent, y)
3. If rootX ≠ rootY:
     a. If rank[rootX] > rank[rootY]:
            parent[rootY] ← rootX
        Else if rank[rootX] < rank[rootY]:
            parent[rootX] ← rootY
        Else:
            parent[rootY] ← rootX
            rank[rootX] ← rank[rootX] + 1
```



KRUSKAL'S ALGORITHM

```
Kruskal(Graph):
1. Sort all edges in ascending order of their weights.
2. Initialize parent and rank arrays for Union-Find.
3. MST ← empty list
4. For each edge (u, v, weight) in sorted edges:
    a. If Find(parent, u) ≠ Find(parent, v):
        Add (u, v, weight) to MST
        Union(parent, rank, u, v)
5. Return MST
```

ALGORITHM ANALYSIS

Time Complexity Analysis:

1. **Sorting edges**:
    - Time complexity: $O(E \log E)$, where E is the number of edges.

2. **Union-Find Operations**:
    - Using path compression and union by rank:
        - Find: $O(\alpha(V))$, where α\alphaα is the inverse Ackermann function.
        - Union: $O(\alpha(V))$.
    - Total for E edges: $O(E \cdot \alpha(V))$.

3. **Overall Kruskal's Algorithm**:
    - Time complexity: $O(E \log E + E \cdot \alpha(V))$, dominated by $O(E \log E)$.

THANK YOU

Hassan atia