1. Proof: Every Minimum Spanning Tree of G Contains All Safe Edges

1. **Assumption for Contradiction:**
   Let $G = (V, E)$ be a connected, weighted graph, and let T be a minimum spanning tree (MST) of G.
   Suppose there exists a safe edge $e = (u, v)$ that is not included in T.

2. **Safe Edge Definition:**
   By definition, e is the minimum-weight edge with exactly one endpoint in some component S of G.
   Without loss of generality, assume $u \in S$ and $v \notin S$.

3. **Cycle Property:**
   If we add e to T, it will create a unique cycle C in $T \cup \{e\}$.
   Since $u \in S$ and $v \notin S$, there must exist another edge e' in C that also crosses S.

4. **Edge Comparison:**
   Because e is a safe edge, it is the minimum-weight edge crossing S.
   Thus, $w(e) \leq w(e')$.

5. **Constructing a New MST:**
   Remove e' from $T \cup \{e\}$ to form $T' = T \cup \{e\} \backslash \{e'\}$.
   Now:

   - T' is still a spanning tree (connected and acyclic).

   - The total weight of T' is

$$w(T') = w(T) + w(e) - w(e')$$

6. **Weight Comparison:**
   Since $w(e) \leq w(e')$, we have $w(T') \leq w(T)$.
   But T is an MST, so $w(T') = w(T)$.
   This implies $w(e) = w(e')$.

7. **Conclusion:**
   T' is also an MST, and it includes the safe edge e.
   This contradicts the assumption that T does not include e.
   Therefore, **every MST must contain all safe edges**.

Q.E.D.

2. Please modified above algorithm to let find minimum spanning forest on an unconnected graph G, and count the number of trees.

```
#include <list>
#include <queue>
```

```
#include <vector>

typedef int WeightType;
struct Edge {
  int u, v;
  WeightType w;
};

struct Graph {
  Graph(int n) : E(n) {}
  std::vector<std::list<Edge>> E;
  void add_edge(int u, int v, WeightType w) {
    E[u].push_back({u, v, w});
    E[v].push_back({v, u, w});
  }
  int n() const { return E.size(); }
};

std::pair<Graph, int> MSF(const Graph &G) {
  auto WeightComp = [](const Edge &e1, const Edge &e2) { return e1.w >
e2.w; };
  std::priority_queue<Edge, std::vector<Edge>, decltype(WeightComp)> ca
ndidates(
      WeightComp);

  Graph forest(G.n());
  std::vector<bool> visited(G.n(), false);
  int tree_count = 0;

  for (int start = 0; start < G.n(); ++start) {
    if (!visited[start]) {
      tree_count++;
      visited[start] = true;

      for (const Edge &e : G.E[start]) {
        if (!visited[e.v])
          candidates.push(e);
      }

      while (!candidates.empty()) {
        Edge safe = candidates.top();
        candidates.pop();
        int u = visited[safe.v] ? safe.u : safe.v;
        if (!visited[u]) {
          visited[u] = true;
          forest.add_edge(safe.u, safe.v, safe.w);

          for (const Edge &e : G.E[u]) {
            if (!visited[e.v])
```

```
                candidates.push(e);
            }
          }
        }
      }
    }
  return {forest, tree_count};
}
```

3. Please design a function to detect an edge is useless or not and analyz e the time complexity of Kurskal’s algorithm which running with your u seless checker.

```
class DSU {
private:
  std::vector<int> parent;
  std::vector<int> rank;

public:
  DSU(int n) : parent(n), rank(n, 0) {
    for (int i = 0; i < n; ++i) {
      parent[i] = i;
    }
  }

  int Find(int u) {
    if (parent[u] != u) {
      parent[u] = Find(parent[u]);
    }
    return parent[u];
  }

  bool Union(int u, int v) {
    int rootU = Find(u);
    int rootV = Find(v);
    if (rootU == rootV) {
      return false;
    }

    if (rank[rootU] > rank[rootV]) {
      parent[rootV] = rootU;
    } else if (rank[rootU] < rank[rootV]) {
      parent[rootU] = rootV;
    } else {
      parent[rootV] = rootU;
      rank[rootU]++;
    }
    return true;
  }
```

```
};

bool isUselessEdge(DSU &dsu, const Edge &e) {
  return dsu.Find(e.u) == dsu.Find(e.v);
}
// sorting edge O(ElogE)
// Find, union O(α(V))
// O(ElogE) or O(ElogV) ，because logE <= 2logV

Graph KruskalMST(const Graph &G) {
  std::vector<Edge> edges;

  for (int u = 0; u < G.V; ++u) {
    for (const Edge &e : G.adj[u]) {
      if (e.u < e.v) {
        edges.push_back(e);
      }
    }
  }

  std::sort(edges.begin(), edges.end(),
            [](const Edge &a, const Edge &b) { return a.w < b.w; });

  DSU dsu(G.V);
  Graph MST(G.V);
  for (const Edge &e : edges) {
    if (!isUselessEdge(dsu, e)) {
      dsu.Union(e.u, e.v);
      MST.addEdge(e.u, e.v, e.w);
    }
  }

  return MST;
}
```

The time complexity of Kruskal's algorithm

1. Collecting All Edges

```
for (int u = 0; u < G.V; ++u) {
    for (const Edge& e : G.adj[u]) {
        if (e.u < e.v) edges.push_back(e); // Avoid duplicates in undir
ected graphs
    }
}
```

- Time Complexity:
  - Each edge is processed once.
  - Total: $O(E)$

2. Sorting the Edges

```
std::sort(edges.begin(), edges.end(), [](const Edge& a, const Edge& b)
{
    return a.w < b.w;
});
```

- Time Complexity:
  - Sorting E edges using an efficient comparison-based sort (e.g., quicksort, mergesort).
  - Total: $O(E\ log\ E)$

3. Union-Find (DSU) Operations

```
for (const Edge& e : edges) {
    if (!isUselessEdge(dsu, e)) { // Find(u) and Find(v)
        dsu.Union(e.u, e.v);      // Union(u, v)
        MST.addEdge(e.u, e.v, e.w);
    }
}
```

- Per-Operation Complexity:
  - `Find` (with path compression): $O\big(a(V)\big)$ (near-constant time).
  - `Union` (with union-by-rank): $O\big(a(V)\big)$.
- Total Operations:
  - Each edge requires 2 `Find` calls + 1 `Union` call.
  - Total: $O\big(Ea(V)\big) = O(E)$ #### 4. Building the MST

  ```
  MST.addEdge(e.u, e.v, e.w); // O(1) per edge insertion (adjacency list)
  ```

- Time complexity:
  - Inserting $V - 1$ edges into the MST.
  - Total: $O(V)$.

Overall Time Complexity

- Dominant Term: $O(E\ log\ E)$ (from sorting).
- Other Terms:
  - Union-Find operations: $O\big(Ea(V)\big) = O(E)$.
  - Edge collection and MST construction: $O(E + V)$.
- Simplified Expression:
  - Since $log\ E \leq 2\ log\ V$ for simple graphs, we also can write:

$$O(E\ log\ E) = O(E\ log\ V)$$

- Final time complexity:

$$O(E\ log\ V)$$

4. `WeightComp` needs $D_{s,t}$ to measure which edge should choose. Giving a weighted rooted tree $T = (V, E)$ and a vertex $v \in V$, please design an algorithm can measure the path cost from root to v and analyze its time complexity.

1. Using DFS to measure

```
int computePathCost(TreeNode* root, TreeNode* target) {
    function<bool(TreeNode*, int, int&)> dfs = [&](TreeNode* node, int
current_sum, int& cost) {
        if (node == target) {
            cost = current_sum;
            return true;
        }
        for (auto [child, weight] : node->children) {
            if (dfs(child, current_sum + weight, cost)) {
                return true;
            }
        }
        return false;
    };

    int cost = 0;
    dfs(root, 0, cost);
    return cost;
}
```

2. Using BFS to measure

```
int computePathCostBFS(TreeNode* root, TreeNode* target) {
    queue<pair<TreeNode*, int>> q; // (node, current_cost)
    q.push({root, 0});

    while (!q.empty()) {
        auto [node, current_cost] = q.front();
        q.pop();
        if (node == target) {
            return current_cost;
        }
        for (auto [child, weight] : node->children) {
            q.push({child, current_cost + weight});
        }
    }
    return -1; // Target not found (impossible in a tree)
}
```

- Time Complexity:
  - DFS and BFS traverse each node and edge exactly once.
  - In a tree with V nodes, the number of edges $E = V - 1$.
  - Thus, the time complexity is $O(V)$.

Why Not Dijkstra's Algorithm?

- Dijkstra's algorithm has a time complexity of $O(E + VlogV)$ (with a binary heap). For trees, this reduces to $O(V + VlogV) = O(VlogV)$, which is less efficient than DFS/BFS $\big(O(V)\big)$.
- Trees have unique paths, making Dijkstra's priority queue unnecessary.

5. Brief what is A* algorithm and explain why A* can always find the optimal answer.

1. What is A* Algorithm?

- A* (A-star) is an informed search algorithm that finds the shortest path between a start node and a goal node in a weighted graph. It combines:
- Dijkstra's completeness: Guarantees finding the shortest path.
- Greedy Best-First Search's efficiency: Uses heuristics to prioritize promising paths.

2. Key Components

Cost Functions:

$g(n)$: Actual cost from the start node to node n.

$h(n)$: Heuristic estimate of the cost from n to the goal (must be admissible).

$f(n) = g(n) + h(n)$: Total estimated cost of the path through n.

Open/Closed Sets:

Open Set: Nodes to be evaluated (priority queue sorted by $f(n)$). *

Closed Set: Nodes already evaluated.

3. Why A* Finds the Optimal Path?

A* is optimal because it:

1. Uses an admissible heuristic to avoid overestimation

2. Combines actual path cost $\big(g(n)\big)$ and heuristic $\big(h(n)\big)$ to prioritize nodes efficiently.

3. Terminates early when the goal is reached, unlike Dijkstra which explores all nodes.

Key Formula:

$$f(n) = g(n) + h(n)$$

Where $h(n)$ must be admissible for guaranteed optimality.