Kruskals Algorithm

Minimum Spanning Trees (MST), Kruskal, Prim & Disjoint Sets — Exam Notes

What is Kruskal’s algorithm used for?

- **Task:** Build a **Minimum Spanning Tree (MST)** of a connected, weighted, undirected graph — i.e., connect **all** vertices with **V−1** edges, **no cycles**, **minimum total weight**. lecture13_kruskals

---

Kruskal’s process (high level)

1. **Sort** all edges by **increasing weight**.
2. Start with a **forest** of single-vertex trees (each vertex alone).
3. Scan edges in order; **add** an edge **iff** it connects **two different trees** (doesn’t form a cycle).
4. Stop when you’ve added **V−1** edges → that set is an MST. lecture13_kruskals

Correctness (simple English):

At the moment Kruskal considers an edge that connects two components, that edge is the **lightest** across the “cut” separating those components; by the **cut property**, it’s safe (belongs to some MST). Replacing any heavier cross-cut edge in a supposed MST with this lighter one can only improve or match the weight, so Kruskal never makes a bad choice.
lecture13_kruskals

Prim’s algorithm — what, how, why, performance

What: Another MST algorithm that grows a **single tree** instead of a forest.

How (binary heap version):

- Start from any vertex with an empty tree.
- Maintain a priority queue (min-key) of **cut edges** that connect the current tree to outside vertices.
- Repeatedly extract the **minimum-weight** eligible edge and **add** that new vertex and its edges to the queue.
- When all vertices are added → MST. lecture13_kruskals

Why correct (plain English):

At each step, Prim picks the **lightest edge leaving the current tree**. By the **cut property** (the tree vs. the rest is a cut), that lightest leaving edge is always safe to add; thus every step preserves optimality until the whole graph is spanned.

Performance (binary heap / PQ):

- Using a binary heap: **$O(|V| \log |V| + |E| \log |V|)$**. (Insert/update $|E|$ edges; $|V|$ extractions.) lecture13_kruskals
- (For context you can remember: an adjacency-matrix + no-heap variant is **$O(V^2)$**, but the heap version above is what you'll usually cite.)

Kruskal — performance & space

- **With efficient DSU:** sort edges **$O(E \log E)$**; each **find/union** is ~amortized **$O(\alpha(V))$** → dominated by sorting ⇒ **$O(E \log E)$** overall. Space **$O(V + E)$**. lecture13_kruskals
- **Naïve implementation (why it's bad):**
  - Store a `leader[vertex]` array and, on union, **scan all vertices** to relabel ($O(V)$ per successful union).
  - With ~$V-1$ unions this costs **$O(V^2)$**; plus sorting **$O(E \log E)$** → **$O(V^2 + E \log E)$** overall. Space still **$O(V + E)$**. lecture13_kruskals

---

Fixing the naïve approach: Disjoint Sets (Union–Find)

The operations & what we want

- **find(x):** returns the **representative (leader)** of x's set → lets us test "same component?".
- **union(x, y):** merges the **two sets** containing x and y.

  Kruskal needs these operations to be very fast.
  lecture13_kruskals

Forest representation (parent pointers)

- Maintain a **forest** where each set is a tree; every node stores a **parent**; roots are **leaders** (parent = self).
- **union(A,B):** make the root of one tree point to the root of the other → **O(1)** pointer change.
- **find(x):** walk parent pointers to the root → **O(height)** of the tree (can be **O(N)** in the worst case if trees get tall). lecture13_kruskals

Union by height (a.k.a. union by rank) — why this gives log V

Idea: Keep trees **shallow** by always attaching the **shorter** tree under the **taller** tree.

- A tree's **height increases by 1** only when you **merge two trees of equal height**.
- Induction/doubling argument: the **minimum** number of nodes in a tree of height **h** is **2^h** (because to get height h you had to combine two height h−1 trees, each with at least 2^(h−1) nodes).
- Therefore with **n** total nodes, the height can be at most $\lfloor \log_2 n \rfloor$.
- That makes **find** worst-case **O(log V)** (and union is one or two finds plus a constant). lecture13_kruskals

Path compression — how it speeds things up

Mechanism: After you do `find(x)` and discover the root **r**, **rewrite every node on the path** from x to r so its parent points **directly to r**.

- This **flattens** the tree along that path.

- A single `find` might still take up to O(log V), but it makes **future finds** on those nodes **O(1)** until another union changes the structure.
- Combining **union by rank (height)** + **path compression** yields a structure where any **sequence of m operations** on **n** elements costs **$O(m \cdot \alpha(n))$**, where **α** is the inverse Ackermann function (grows **slower than log***; ≤ 4 for all practical n). In practice this is **near-constant** time per op. lecture13_kruskals

> Why "amortized" and not worst-case per op?
>
> A costly find happens rarely and **pays for itself** by flattening paths, so the average over many ops becomes tiny. The formal proof (CLRS §19.4) is beyond scope, but the slides give the result and intuition.
> lecture13_kruskals

Bottom line for Kruskal with DSU

- Sort edges **O(E log E)**; each of the **E** `find/union` ops is ~**$O(\alpha(V))$** amortized ⇒ total **O(E log E)**. **This is the standard bound you quote.** lecture13_kruskals

---

Proof ideas you can write fast in an exam

Kruskal (cut property, plain English)

- When Kruskal picks an edge connecting two components, it's the **cheapest edge** across that **cut**; any MST must use some crossing edge, and if it used a heavier one, **swapping** in Kruskal's lighter edge would not hurt and can improve. Thus every chosen edge is **safe**, and after **V−1** such choices we have an MST. lecture13_kruskals

Prim (same cut property)

- The current tree vs. the rest defines a cut. The **lightest outgoing edge** is safe by the cut property; adding it preserves optimality until all vertices are included. (Prim recap and complexity on slides.) lecture13_kruskals

---

Pseudocode sketches (what data structures are doing)

Kruskal (with DSU):

```
sort edges by weight
make-set(v) for all vertices v
mst_weight = 0
for (u,v,w) in edges:
    if find(u) != find(v):   # different components?
        mst_weight += w
        union(u, v)
return mst_weight
```

- **Data structures:** array/list of edges; **DSU** with `parent[]`, `rank[]`, path compression in `find`.
- **Flow:** sort → scan → conditionally union. lecture13_kruskals

Prim (binary heap):

```
pick any start vertex s
dist[v] = +∞ for all v; dist[s] = 0
push (0, s) into min-heap
while heap not empty:
    (key,u) = extract-min()
    if u already in tree: continue
    add u to tree; add key to total
    for each edge (u,v,w):
        if v not in tree and w < dist[v]:
            dist[v] = w
            push (w, v) into heap
```

- **Data structures:** adjacency list; **min-heap** keyed by best edge weight to connect each outside vertex.
- **Flow:** grow one tree, always add the **cheapest connecting edge**. lecture13_kruskals

---

Naïve vs. optimized — time & space summary

Algorithm / DS	Time	Space	Notes
Kruskal (naïve "leader relabel" union)	**$O(V^2 + E \log E)$**	O(V+E)	Union relabels whole set each time (O(V)). lecture13_kruskals
Kruskal + DSU (rank + path compression)	**O(E log E)** overall (sorting dominates)	O(V+E)	Each `find/union` amortized **O(α(V))**. lecture13_kruskals
Prim (binary heap)	**O(	V	log

Quick "why" bullets for DSU complexities (write these if asked "explain why"):

- **Union by height/rank ⇒ O(log V) find (worst-case):** Height increases only when merging **equal-height** trees; minimal size doubles each increase ⇒ height ≤ $\log_2 n$. lecture13_kruskals
- **Path compression ⇒ amortized near-O(1):** Each expensive `find` flattens its path; future `find`s are cheap. Over **m** operations the total time is **O(m α(n))**, with **α** ≤ 4 in practice. lecture13_kruskals