Shortest Path Problems

Minimum Spanning Trees (MST)

Problems Covered

1. Minimum Spanning Tree Weight (Kruskal's Algorithm) (Easy)
2. Minimum Cost to Connect All Points (Prim's Algorithm) (Easy - Medium)
3. Minimum Bottleneck Path (Medium)
4. Optimize Water Distribution in a Village (Medium - Hard)
5. Critical and Pseudo-Critical Edges of an MST (Hard)
6. Second-Best Minimum Spanning Tree (Hard)

Minimum Spanning Tree Weight

Problem 1 | Easy

The Problem

- Given a weighted, undirected graph with n vertices (0 to n−1) and a list of edges (u, v, w), find the total weight of its **minimum spanning tree**: n−1 edges that connect all vertices with the smallest possible total weight.
- If the graph is not connected, no spanning tree exists: return −1.

Example:

n = 6

Edges (u, v, w): (0,1,4), (0,2,3), (1,2,1), (1,3,2), (2,3,4), (3,4,2), (4,5,6), (3,5,7)

Output: 14

Any ideas on how this can be solved?

Hints

- A spanning tree on n vertices has exactly n−1 edges and no cycle. In which order should the edges be considered?
- If you pick edges from cheapest to costliest, when must an edge be rejected?
- How can you quickly check whether two vertices are already connected by the edges chosen so far?

The Solution

Key Idea (Kruskal)

- Greedy: the cheapest edge that does not form a cycle is always safe to add.
- A disjoint-set union (DSU) tells us whether an edge joins two different components.

Steps

- Sort all edges by weight. Put every vertex in its own set.
- For each edge (u, v, w): if u and v are in different sets, take the edge, merge the sets and add w.
- Stop after n−1 edges are taken.
- If fewer than n−1 edges were taken, the graph is disconnected: return −1.

```
Kruskal(n, edges):
  sort edges by weight
  make-set(0..n-1)
  total = 0
  used = 0
  for (u, v, w) in edges:
    if union(u, v):
      total += w
      used += 1
  if used == n - 1:
    return total
  return -1
```

The Solution (contd.)

Disjoint Set Union

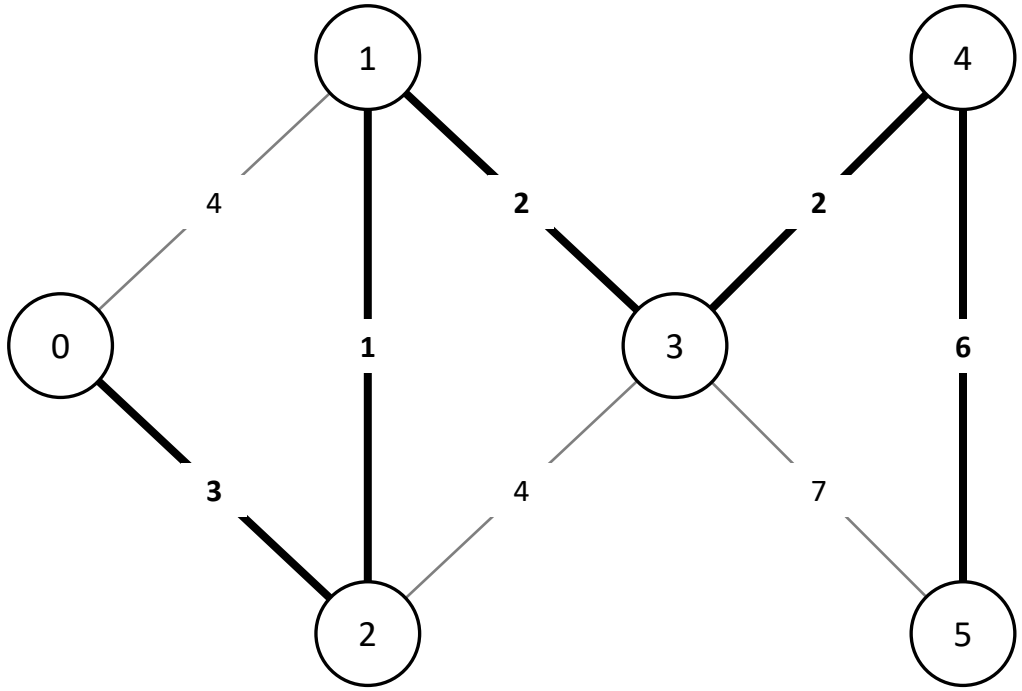
- parent[x] points towards the representative of x's set. find(x) follows these pointers.
- Path compression: after find, every visited node points directly to the root.
- Union by size: attach the smaller tree below the larger one.
- union returns false when both vertices are already in the same set. Adding that edge would create a cycle, so Kruskal skips it.
- With both tricks, each operation costs O(α(V)), which is almost constant.

```
find(x):
  if parent[x] != x:
    parent[x] = find(parent[x])
  return parent[x]

union(a, b):
  ra = find(a)
  rb = find(b)
  if ra == rb: return false
  if size[ra] < size[rb]:
    swap(ra, rb)
  parent[rb] = ra
  size[ra] += size[rb]
  return true
```

Example

n = 6 (bold edges form the MST)



Edge (u,v)	Weight	Action	Total
(1,2)	1	take	1
(1,3)	2	take	3
(3,4)	2	take	5
(0,2)	3	take	8
(0,1)	4	skip (cycle)	8
(2,3)	4	skip (cycle)	8
(4,5)	6	take	14
(3,5)	7	skip (cycle)	14

MST weight = 14 (5 edges = n−1)

Complexity Analysis

1. Time Complexity

- Sorting E edges takes O(E log E). Since $E \leq V^2$, this is the same as O(E log V).
- The loop makes at most E union/find calls, each $O(\alpha(V))$, which is almost constant.
- Time Complexity = **O(E log E)**

2. Space Complexity

- parent[] and size[] arrays: O(V). Edge list: O(E).
- Space Complexity = **O(V + E)**

Minimum Cost to Connect All Points

Problem 2 | Easy - Medium

The Problem

- Given n points on a 2D plane, points[i] = (xi, yi). Connecting two points costs their **Manhattan distance**: $|xi - xj| + |yi - yj|$.
- Find the minimum total cost to connect all the points so that there is exactly one simple path between any two of them.

Example:

n = 5

points = (0,0), (2,2), (3,10), (5,2), (7,0)

Output: 20

Any pair can be connected, so there are $n(n-1)/2$ possible edges. Do we need to list all of them?

Hints

- Every pair of points can be joined, so this is an MST on a complete graph with $n(n-1)/2$ edges. Is sorting all of them a good idea?
- Start from any one point and grow a tree one point at a time. Which outside point should be attached next?
- For each point outside the tree, you only need its cheapest link to the tree. What changes when a new point joins the tree?

The Solution

Key Idea (Prim)

- Grow one tree from point 0. Each step adds the outside point with the cheapest link to the tree.
- Keep d[v] = cheapest cost to connect v to the current tree. Distances are computed on the fly, so no edge list is needed.

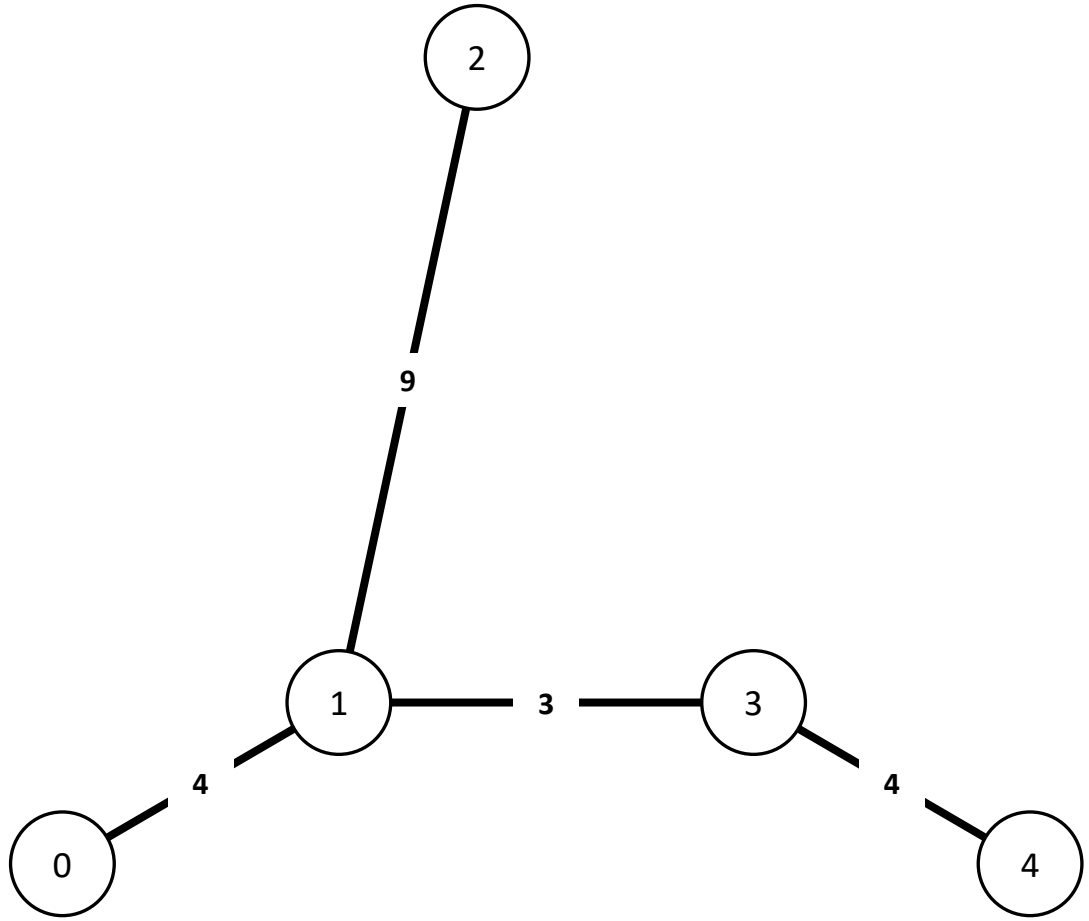
Steps

- d[0] = 0 and d[v] = ∞ for all other points.
- Pick the unvisited u with the smallest d[u]. Mark it visited and add d[u] to the total.
- For every unvisited v: d[v] = min(d[v], dist(u, v)).
- Repeat n times. The total is the answer.

```
Prim(points, n):
  d[all] = INF
  d[0] = 0
  visited[all] = false
  total = 0
  repeat n times:
    u = unvisited i with min d[i]
    visited[u] = true
    total += d[u]
    for v in 0..n-1:
      if not visited[v]:
        c = dist(points[u], points[v])
        d[v] = min(d[v], c)
  return total
```

Example

points = (0,0), (2,2), (3,10), (5,2), (7,0) (tree edges shown)



Add	Cost	Total	d[] after update
P0	0	0	[–, 4, 13, 7, 7]
P1	4	4	[–, –, 9, 3, 7]
P3	3	7	[–, –, 9, –, 4]
P4	4	11	[–, –, 9, –, –]
P2	9	20	[–, –, –, –, –]

Minimum cost = 20

– means the point is already in the tree.

Complexity Analysis

1. Time Complexity

- n rounds. Each round scans n points to find the minimum and n points to update d[].
- No sorting and no stored edge list: each distance is computed in O(1) when needed.
- Time Complexity = **$O(n^2)$**
- (Kruskal on all $n(n-1)/2$ edges would need $O(n^2 \log n)$ because of sorting.)

2. Space Complexity

- d[] and visited[] arrays: O(n).
- Space Complexity = **O(n)**

Minimum Bottleneck Path

Problem 3 | Medium

The Problem

- Given a weighted, undirected graph with n vertices and two vertices s and t, the cost of a path is the **largest edge weight** on it (not the sum).
- Find the smallest possible cost over all paths from s to t. If t cannot be reached from s, return −1.

Example:

n = 6, s = 0, t = 5

Edges (u, v, w): (0,1,2), (0,2,7), (1,2,3), (1,3,9), (2,4,5), (3,5,4), (4,5,6), (3,4,8)

Output: 6 (path 0→1→2→4→5 has largest edge 6)

Any ideas on how this can be solved?

Hints

- The cost is the largest edge, not the sum. Does the usual idea of Dijkstra still apply?
- Suppose only edges with weight ≤ X are allowed. When is t reachable from s? What happens as X grows?
- Add edges from the lightest to the heaviest. At which moment do s and t first become connected?

The Solution

Key Idea

- Add edges in increasing order of weight, as in Kruskal. The moment s and t fall into the same component, the edge just added is the answer.
- All edges added so far weigh ≤ w, so a path with largest edge w exists. Before this edge, no such path existed with smaller weights.
- The s–t path inside the MST is a minimum bottleneck path.

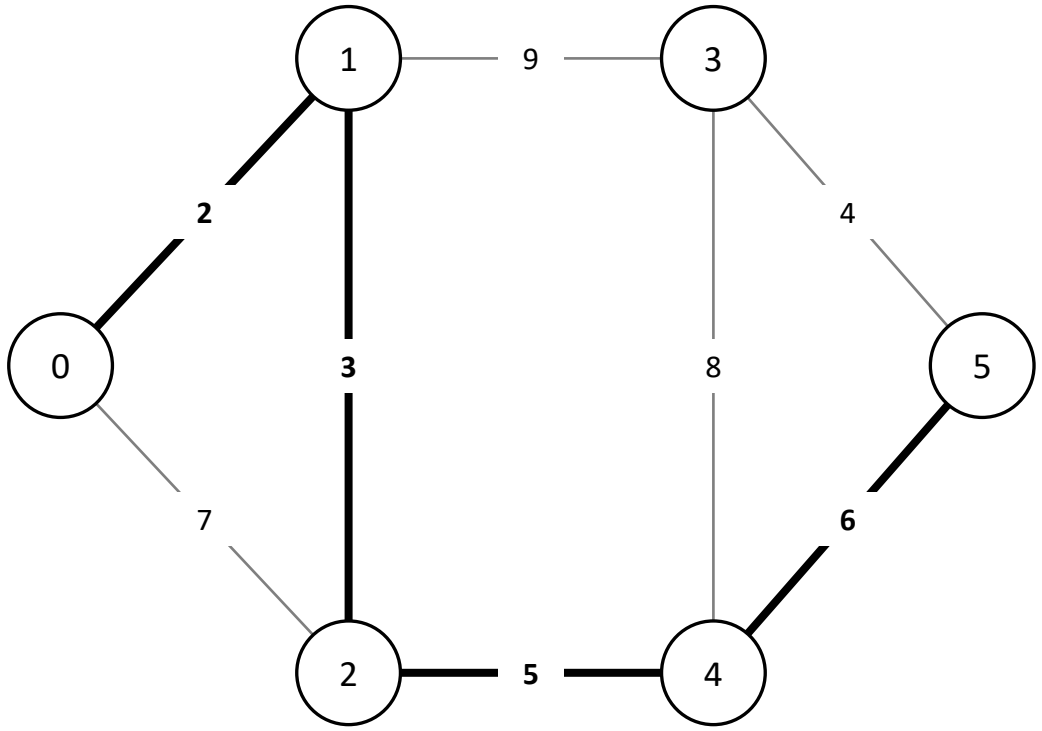
Steps

- If s == t, return 0.
- Sort edges; start with every vertex in its own set.
- For each edge: union(u, v). If find(s) == find(t), return w.
- If the edges run out first, return −1.

```
MinBottleneck(n, edges, s, t):
  if s == t:
    return 0
  sort edges by weight
  make-set(0..n-1)
  for (u, v, w) in edges:
    union(u, v)
    if find(s) == find(t):
      return w
  return -1
```

Example

n = 6, s = 0, t = 5 (bold = answer path)



Edge	w	Components after	0 ~ 5?
(0,1)	2	{0,1}	no
(1,2)	3	{0,1,2}	no
(3,5)	4	{0,1,2} {3,5}	no
(2,4)	5	{0,1,2,4} {3,5}	no
(4,5)	6	{0,1,2,3,4,5}	yes

Answer = 6

The shortest path by sum (0→1→3→5, total 15) has largest edge 9, so it is not the answer.

Complexity Analysis

1. Time Complexity

- Sorting the edges: O(E log E). The loop runs at most E times with O(α(V)) per step.
- Time Complexity = **O(E log E)**
- (Another way: binary search on X and run BFS using edges ≤ X, giving O(E log W).)

2. Space Complexity

- DSU arrays: O(V). Edge list: O(E).
- Space Complexity = **O(V + E)**

Optimize Water Distribution in a Village

Problem 4 | Medium - Hard

The Problem

- A village has n houses numbered 1 to n. Each house can either **build its own well** at cost wells[i−1], or get water through a **pipe** from another house that has water.
- pipes[j] = (house1, house2, cost) is the cost of laying a pipe between two houses. Pipes work in both directions.
- Find the minimum total cost to supply water to every house.

Example:

n = 4, wells = [3, 5, 2, 5]

pipes = (1,2,1), (3,4,3), (1,3,7), (2,4,4)

Output: 9 (wells at houses 1 and 3, pipes 1–2 and 3–4)

Hints

- Building a well and laying a pipe look like different kinds of choices. Can both be treated as the same kind of edge?
- Imagine one extra vertex 0, the water source. What would an edge (0, i) with cost wells[i−1] mean?
- With that vertex added, what does "every house has water" look like in graph terms?

The Solution

Key Idea

- Add a virtual vertex 0 and an edge (0, i, wells[i−1]) for every house. Building a well at house i means connecting i to the source.
- Every house has water exactly when all n+1 vertices are connected, so the cheapest way is an MST of the new graph.

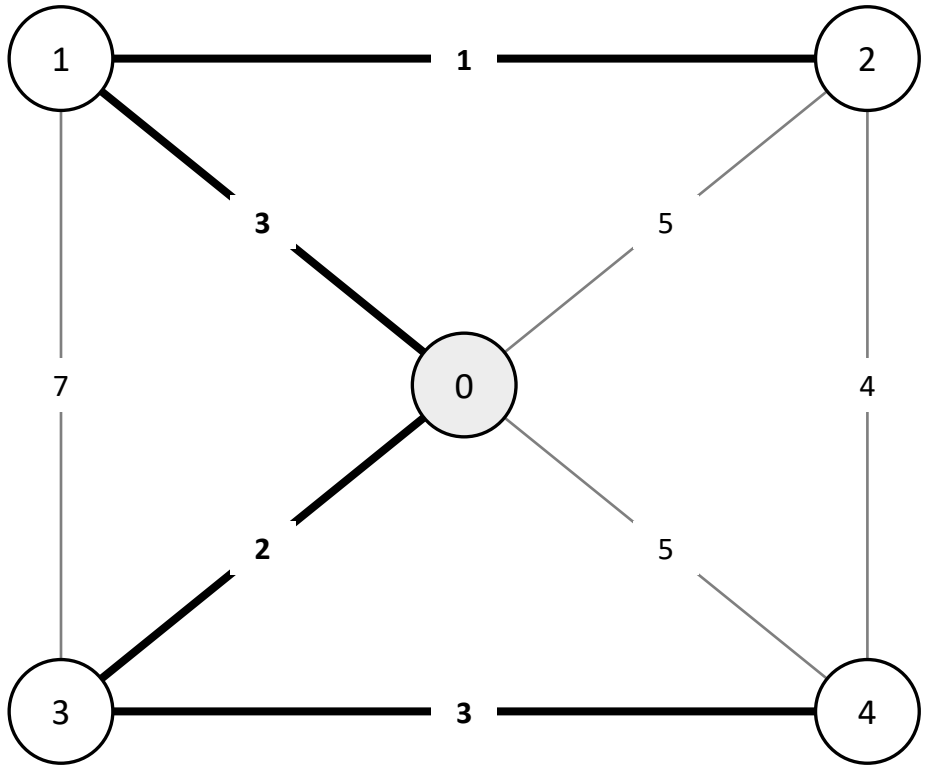
Steps

- edges = pipes + the n well edges (0, i, wells[i−1]).
- Run Kruskal on vertices 0 to n.
- The MST weight is the answer. The graph is always connected because every house has a well edge.
- Edges that touch 0 are wells; the rest are pipes.

```
MinCostWater(n, wells, pipes):
  edges = pipes
  for i in 1..n:
    edges.add((0, i, wells[i-1]))
  sort edges by weight
  make-set(0..n)
  total = 0
  for (u, v, w) in edges:
    if union(u, v):
      total += w
  return total
```

Example

Vertex 0 = water source (bold edges form the MST)



Edge	Weight	Action	Total
(1,2) pipe	1	take	1
(0,3) well	2	take	3
(0,1) well	3	take	6
(3,4) pipe	3	take	9
(2,4) pipe	4	skip (cycle)	9
(0,2) well	5	skip (cycle)	9
(0,4) well	5	skip (cycle)	9
(1,3) pipe	7	skip (cycle)	9

Minimum cost = 9 (4 edges = (n+1) − 1)

Complexity Analysis

1. Time Complexity

- Let P be the number of pipes. The new graph has n+1 vertices and P + n edges.
- Sorting dominates; the DSU operations are almost O(1) each.
- Time Complexity = **O((n + P) log(n + P))**

2. Space Complexity

- Edge list: O(n + P). DSU arrays: O(n).
- Space Complexity = **O(n + P)**

Critical and Pseudo-Critical Edges

Problem 5 | Hard

The Problem

- Given a connected, weighted, undirected graph with n vertices and edges[i] = (u, v, w), find all critical and pseudo-critical edges of its MSTs. An MST may not be unique.
- **Critical**: the edge is in every MST (deleting it makes the MST heavier or disconnects the graph).
- **Pseudo-critical**: the edge is in some MSTs but not in all of them.
- Return both lists of edge indices.

Example:

n = 5, edges as index: (u,v,w)

0: (0,1,1) 1: (1,2,1) 2: (2,3,2) 3: (0,3,2) 4: (0,4,3) 5: (3,4,3) 6: (1,4,6)

Output: critical = [0, 1], pseudo-critical = [2, 3, 4, 5]

Hints

- First compute W, the weight of an MST. Every MST of the graph has this same weight.
- If you ban edge i and the MST becomes heavier than W (or the graph breaks), what does that say about edge i?
- If you force edge i into the tree and the MST still weighs W, what does that say about edge i? What if the weight goes up?

The Solution

Key Idea

- Let W = MST weight. Test every edge i with two Kruskal runs:
- Skip i: weight > W (or disconnected) → i is in every MST → **critical**.
- Force i in: weight == W → i is in some MST. If it is not critical → **pseudo-critical**.
- An edge that is neither is in no MST at all.

Steps

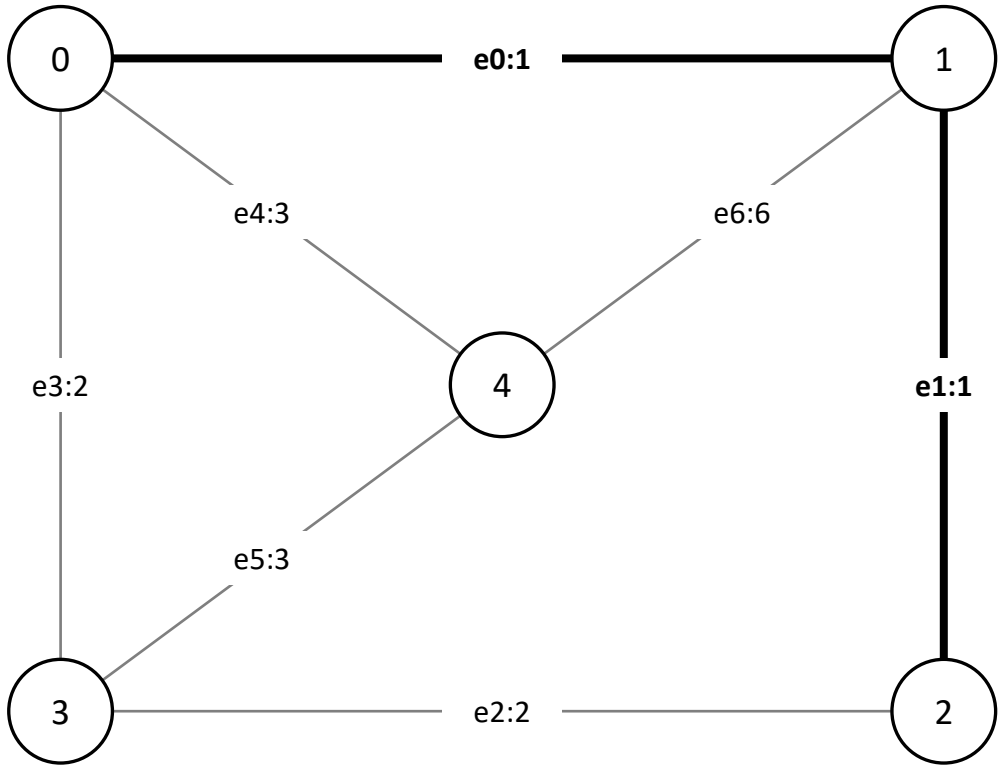
- Sort edges by weight, remembering original indices.
- Compute W with plain Kruskal.
- For each edge, run mst(skip = i), then mst(force = i) if needed.

```
mst(skip, force):
  make-set(0..n-1)
  total = 0, cnt = 0
  if force != -1:
    union(force.u, force.v)
    total += force.w, cnt++
  for e in sorted edges:
    if e == skip: continue
    if union(e.u, e.v):
      total += e.w, cnt++
  return total if cnt == n-1 else INF

W = mst(-1, -1)
for each edge i:
  if mst(i, -1) > W: critical.add(i)
  else if mst(-1, i) == W:
    pseudo.add(i)
```

Example

W = 7 (bold = critical edges, label = index:weight)



Edge	w	Skip	Force	Type
e0 (0,1)	1	8	–	Critical
e1 (1,2)	1	8	–	Critical
e2 (2,3)	2	7	7	Pseudo-critical
e3 (0,3)	2	7	7	Pseudo-critical
e4 (0,4)	3	7	7	Pseudo-critical
e5 (3,4)	3	7	7	Pseudo-critical
e6 (1,4)	6	7	10	In no MST

Skip / Force = MST weight with that edge skipped / forced in.

critical = [0, 1], pseudo-critical = [2, 3, 4, 5]

Complexity Analysis

1. Time Complexity

- Sorting once: O(E log E).
- Each of the E edges triggers at most two Kruskal runs, and each run scans E edges with O(α(V)) per DSU step.
- Time Complexity = **$O(E^2 \alpha(V))$**, which is fine for the usual limit of a few hundred edges.

2. Space Complexity

- Sorted edge list: O(E). DSU arrays (reset on every run): O(V).
- Space Complexity = **O(V + E)**

Second-Best Minimum Spanning Tree

Problem 6 | Hard

The Problem

- Given a connected, undirected graph with n vertices and m edges with **distinct** weights, the MST T is unique.
- The second-best spanning tree is the spanning tree other than T with the smallest total weight. Find that weight.
- If the graph itself is a tree (m = n−1), return −1.

Example:

n = 5

Edges (u, v, w): (0,1,1), (1,2,3), (0,2,4), (2,3,2), (3,4,5), (1,3,6), (2,4,7)

Output: 12 (the MST weighs 11)

Any ideas on how this can be solved?

Hints

- Build the MST T first. Does the second-best tree differ from T in one edge, or can it differ in many?
- Add one non-tree edge (u, v) to T. It creates exactly one cycle. Which edge of that cycle should be removed to keep the weight as small as possible?
- The removed edge is the heaviest tree edge on the path u → v. How can you get this value quickly for any pair?

The Solution

Key Idea

- The second-best tree differs from T by one swap: add a non-tree edge (u, v, w) and remove the heaviest tree edge on the path u → v.
- New weight = W + w − maxEdge(u, v). Try every non-tree edge and keep the minimum.

Steps

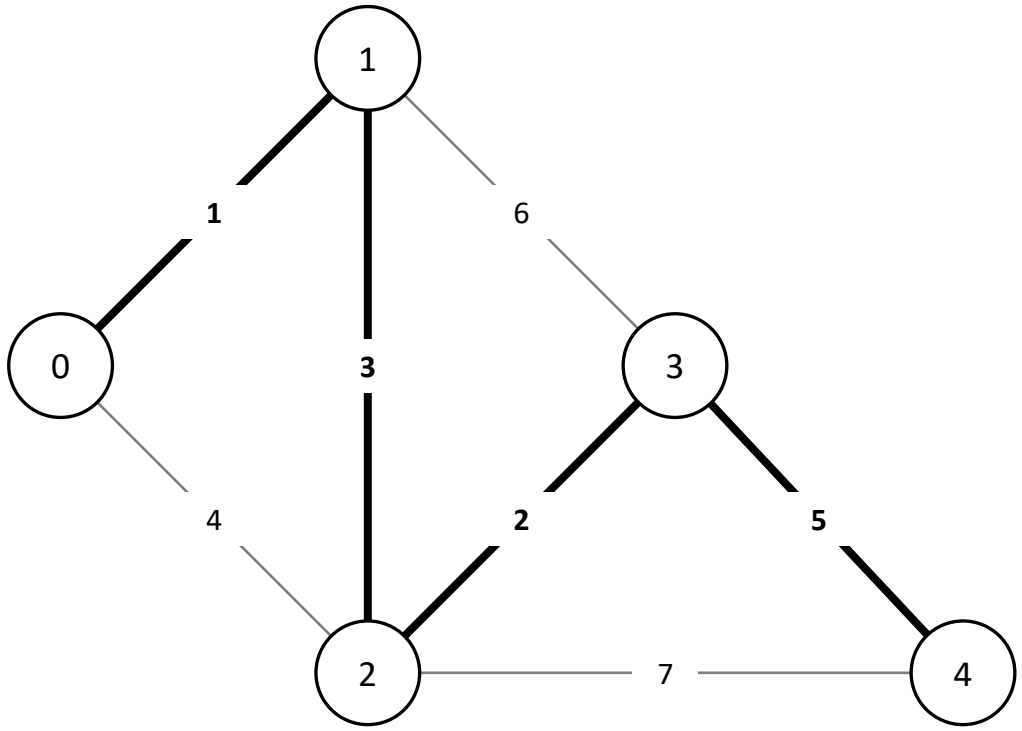
- Build T with Kruskal; note its weight W.
- From every vertex s, run a DFS on T and store mx[s][v] = heaviest edge on the path s → v.
- For each non-tree edge compute W + w − mx[u][v].
- The smallest value is the answer (−1 if no non-tree edge).

```
SecondBest(n, edges):
  (T, W) = Kruskal(n, edges)
  for s in 0..n-1:
    dfs(s, s, -1, 0)
  ans = INF
  for (u, v, w) not in T:
    ans = min(ans, W + w - mx[u][v])
  return ans if ans < INF else -1

dfs(s, u, parent, best):
  mx[s][u] = best
  for (v, w) in T[u]:
    if v != parent:
      dfs(s, v, u, max(best, w))
```

Example

n = 5 (bold = MST T, W = 11)



Add edge	Path in T	Max	New weight
(0,2,4)	0–1–2	3	11+4−3 = 12
(1,3,6)	1–2–3	3	11+6−3 = 14
(2,4,7)	2–3–4	5	11+7−5 = 13

Second-best weight = 12

Swap: add (0,2) and remove the heaviest edge on its tree path, (1,2).

Complexity Analysis

1. Time Complexity

- Kruskal: O(E log E). n DFS runs on a tree, O(V) each, fill mx[][]: $O(V^2)$. Scanning non-tree edges: O(E).
- Time Complexity = **$O(E \log E + V^2)$**

2. Space Complexity

- mx[][] table: $O(V^2)$. Tree adjacency lists: O(V).
- Space Complexity = **$O(V^2)$**
- (For large V, binary lifting on T gives O(E log V) time and O(V log V) space.)