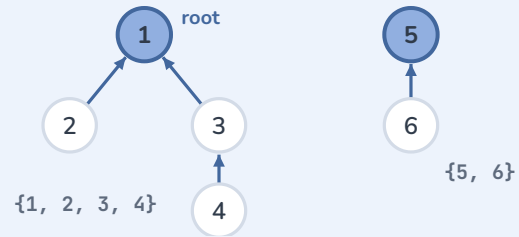


Disjoint sets

Chapter 3.2 · Overall level: Intermediate



Cables are added to a network one at a time, and between additions someone keeps asking: *can computer 3 already talk to computer 8?* Searching the whole network for every question is far too slow. A few hundred thousand questions need a smarter memory.

A disjoint-set structure, also called **union-find**, answers each question in almost constant time. It forgets the routes and remembers only which group every vertex belongs to, which is exactly what the question needs.

3.2.1 Connectivity without storing routes

Suppose n vertices start separate, and two kinds of operations arrive: **join** u and v (add an edge), and **ask** whether u and v are connected. With BFS from Chapter 3.1, each question costs $O(n + m)$, so q questions cost up to $O(q(n + m))$: about 10^{10} steps when everything is 200,000.

Notice what a question really asks. The routes do not matter; only the **connected components** do. The vertices are split into groups that never share a member, a **partition** into disjoint sets, and a join merges two groups into one. If each group has one chosen member, its **representative**, then “are u and v connected?” becomes “do they have the same representative?”

Two operations are needed: **find**(x) returns the representative of the group of x , and **union**(u , v) merges the groups of u and v . The rest of the chapter makes both fast.

3.2.2 A forest of parent pointers

Each group is stored as a tree in an array `parent`. Every vertex points to a parent in its own group; the root points to itself and is the representative. `find(x)` follows parent pointers from `x` until it reaches a vertex that is its own parent. `union(u, v)` finds both roots and, if they differ, makes one root the parent of the other: a single assignment merges two whole groups.

The running example has $n = 5$ vertices and the operations `union(1, 2)`, `union(3, 4)`, `union(2, 3)`, then `find(4)`. The linking rule used here is **union by size**, explained in Section 3.2.3: the root of the smaller tree is hung under the root of the larger one, and on a tie the root of the *first* argument stays the root. In this example that is also the smaller number, but any fixed tie rule works.

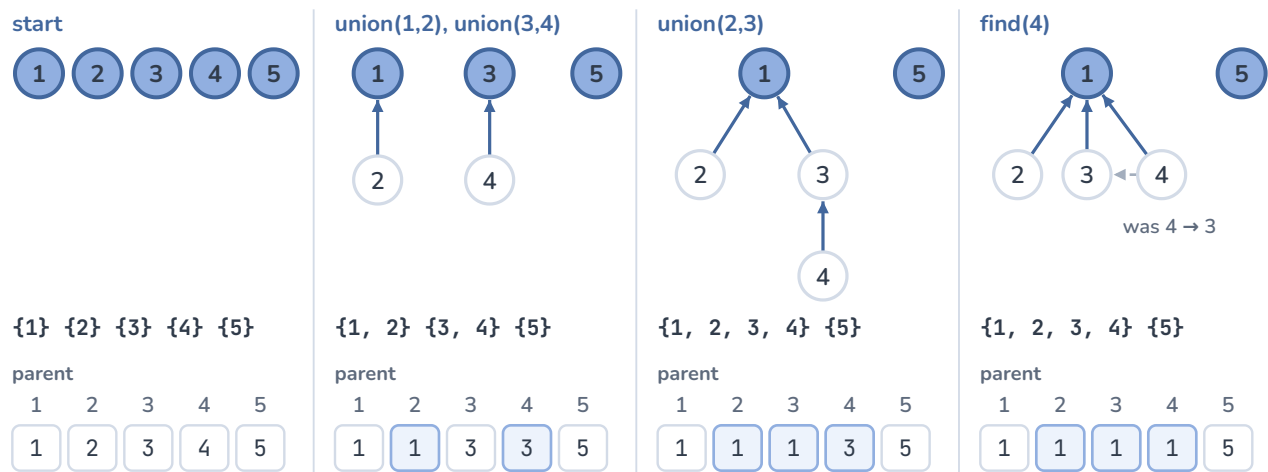


Figure 3.2.1 — Representatives encode membership; their numeric identity has no inherent meaning. Arrows are actual parent pointers; the sets written under each forest are the conceptual groups. Roots are blue; shaded parent entries point elsewhere.

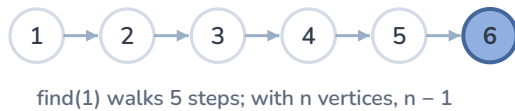
Follow the third panel. `union(2, 3)` does not link 2 and 3 directly. It computes `find(2) = 1` and `find(3) = 3`, both trees have size 2, so by the tie rule root 3 becomes a child of root 1. Vertex 4 is now two steps below its representative. The groups are $\{1, 2, 3, 4\}$ and $\{5\}$; a later `union(1, 4)` finds the same root 1 for both and changes nothing.

Representatives are only labels. Had the tie rule chosen 3, every answer to “same group?” would be identical. Never use the number of a representative as data; compare representatives, nothing more.

3.2.3 Union by size keeps trees shallow

The cost of `find` is the depth of the starting vertex. A careless rule, “always put the root of u under the root of v ”, can build a chain: `union(1, 2), union(2, 3), ...` produce $1 \rightarrow 2 \rightarrow 3 \rightarrow \dots \rightarrow n$, and every `find(1)` then walks $n - 1$ steps.

naive: root of u under root of v



by size: smaller tree under larger

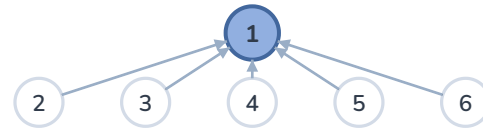


Figure 3.2.2 — The same joins with two linking rules. Keeping the larger tree’s root on top stops chains from forming.

Union by size stores, for every root, the number of vertices in its tree, and always hangs the smaller tree under the larger root. Why this works: a vertex gets one step deeper only when its tree is hung under another tree that is at least as large, so the size of the tree containing it at least *doubles*. A size cannot exceed n , so a vertex sinks at most $\log_2 n$ times. With $n = 200,000$ that is at most 17 steps for any `find`.

3.2.4 Path compression

A second idea makes later calls even cheaper. After `find(x)` has located the root, walk the same path again and point every vertex on it directly at the root. The fourth panel of Figure 3.2.1 shows `find(4)`: the path $4 \rightarrow 3 \rightarrow 1$ is replaced by $4 \rightarrow 1$, so the next `find(4)` takes one step.

Compression never changes which root a vertex reaches, so no answer changes; only the shape of the trees does.

With both improvements, any sequence of k operations on n vertices takes $O(k \cdot \alpha(n))$ time in total, where α is the inverse Ackermann function: it grows so slowly that $\alpha(n) \leq 4$ for every n that fits in a computer. This is an **amortized** bound: it is a guarantee about the total, or the average per operation. A single `find` can still take up to about $\log_2 n$ steps the first time it walks a long path; it simply makes later calls cheaper. It is therefore wrong to say that each call is $O(1)$ in the worst case.

The code writes `find` as two loops instead of a recursive call: the first finds the root, the second redirects the path. A recursive version is shorter, but before compression a path can be long enough to overflow the call stack in some languages.

3.2.5 Counting components and sizes

Two pieces of information come almost for free. The number of components starts at n and drops by exactly one for each union that joins two *different* roots; a union whose endpoints already share a root changes nothing, so it must not change the count or add any sizes. The size of a component is `size[find(x)]`, and the largest size can only grow when a union succeeds, so one comparison after each successful union keeps it up to date.

Operation	Roots found	Joined?	Components	Largest size
start	—	—	5	1
union(1, 2)	1, 2	yes: size[1] = 2	4	2
union(3, 4)	3, 4	yes: size[3] = 2	3	2
union(2, 3)	1, 3	yes: size[1] = 4	2	4
union(1, 4)	1, 1	no: already together	2	4

Why it is correct. Invariant: two vertices have the same root exactly when the joins so far connect them. It holds at the start (every vertex alone). A successful union merges exactly the two groups containing its endpoints, which is exactly what the new edge does to connectivity; a union inside one group changes neither. `find` and path compression never move a vertex to another tree.

Pseudocode

Disjoint sets with union by size and path compression

```

parent[v] ← v and size[v] ← 1 for every v;  components ← n

find(x):
    root ← x
    while parent[root] ≠ root: root ← parent[root]
    while x ≠ root:             (path compression)
        next ← parent[x]; parent[x] ← root; x ← next
    return root

union(u, v):
    a ← find(u); b ← find(v)
    if a = b: return             (already together: nothing changes)
    if size[a] < size[b]: swap a and b    (a is the larger root)
    parent[b] ← a; size[a] ← size[a] + size[b]; components ← components - 1

```

3.2.6 Code in four languages

Each program solves the first practice problem, **Connection checks**. The first line holds n and q . Each of the next q lines is “type u v ”: type 1 joins u and v , type 2 asks whether they are connected, and the program prints YES or NO for every question.

Sample input	Sample output
4 5	NO
2 1 3	YES
1 1 2	NO
1 2 3	
2 1 3	
2 1 4	

In the sample the first question comes before any join, so 1 and 3 are separate. After joining 1–2 and 2–3 they share a root, while 4 is still alone. The whole run takes $O((n + q) \alpha(n))$ time and $O(n)$ memory.

Python 3

```
import sys

def main():
    data = sys.stdin.buffer.read().split()
    n, q = int(data[0]), int(data[1])
    parent = list(range(n + 1)) # every vertex starts as its own root
    size = [1] * (n + 1)       # size[r] is meaningful for roots only

    def find(x):
        root = x
        while parent[root] != root:
            root = parent[root]
        while x != root:          # path compression: point at the root
            nxt = parent[x]
            parent[x] = root
            x = nxt
        return root

    out = []
    for k in range(q):
        t = data[2 + 3 * k]
        ru, rv = find(int(data[3 + 3 * k])), find(int(data[4 + 3 * k]))
        if t == b"1":
            if ru != rv:
                if size[ru] < size[rv]:
                    ru, rv = rv, ru # ru is now the root of the larger tree
                parent[rv] = ru
                size[ru] += size[rv]
            else:
                out.append("YES" if ru == rv else "NO")
        else:
            out.append("NO")
    print("\n".join(out))

main()
```

C++17

```
#include <bits/stdc++.h>
using namespace std;

vector<int> parent, sz;

int find(int x) {
    int root = x;
    while (parent[root] != root) root = parent[root];
    while (x != root) {
        int next = parent[x];
        parent[x] = root;
        x = next;
    }
    return root;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n, q;
    cin >> n >> q;
    parent.resize(n + 1);
    iota(parent.begin(), parent.end(), 0); // every vertex is its own root
    sz.assign(n + 1, 1);
    while (q--) {
        int t, u, v;
        cin >> t >> u >> v;
        int ru = find(u), rv = find(v);
        if (t == 1) {
            if (ru == rv) continue;
            if (sz[ru] < sz[rv]) swap(ru, rv); // ru: root of the larger tree
            parent[rv] = ru;
            sz[ru] += sz[rv];
        } else {
            cout << (ru == rv ? "YES" : "NO") << '\n';
        }
    }
}
```

Java 17

```
import java.io.*;

public class Main {
    static int[] parent, size;

    static int find(int x) {
        int root = x;
        while (parent[root] != root) root = parent[root];
        while (x != root) {
            int next = parent[x];
            parent[x] = root;
            x = next;
        }
        return root;
    }

    public static void main(String[] args) throws IOException {
        StreamTokenizer in = new StreamTokenizer(
            new BufferedReader(new InputStreamReader(System.in)));
        in.nextToken(); int n = (int) in.nval;
        in.nextToken(); int q = (int) in.nval;
        parent = new int[n + 1];
        size = new int[n + 1];
        for (int v = 0; v <= n; v++) { parent[v] = v; size[v] = 1; }
        StringBuilder out = new StringBuilder();
        for (int k = 0; k < q; k++) {
            in.nextToken(); int t = (int) in.nval;
            in.nextToken(); int ru = find((int) in.nval);
            in.nextToken(); int rv = find((int) in.nval);
            if (t == 2) {
                out.append(ru == rv ? "YES\n" : "NO\n");
            } else if (ru != rv) {
                if (size[ru] < size[rv]) { int s = ru; ru = rv; rv = s; }
                parent[rv] = ru;
                size[ru] += size[rv];
            }
        }
        System.out.print(out);
    }
}
```

JavaScript (Node)

```
const data = require("fs").readFileSync(0, "utf8").trim().split(/\s+/).map(Number);
const n = data[0], q = data[1];
const parent = new Int32Array(n + 1), size = new Int32Array(n + 1).fill(1);
for (let v = 0; v <= n; v++) parent[v] = v; // every vertex is its own root

function find(x) {
    let root = x;
    while (parent[root] !== root) root = parent[root];
    while (x !== root) {
        const next = parent[x]; // path compression
        parent[x] = root;
        x = next;
    }
    return root;
}

const out = [];
for (let k = 0; k < q; k++) {
    const t = data[2 + 3 * k];
    let ru = find(data[3 + 3 * k]), rv = find(data[4 + 3 * k]);
    if (t === 2) {
        out.push(ru === rv ? "YES" : "NO");
    } else if (ru !== rv) {
        if (size[ru] < size[rv]) [ru, rv] = [rv, ru]; // ru: larger tree's root
        parent[rv] = ru;
        size[ru] += size[rv];
    }
}
console.log(out.join("\n"));
```

3.2.7 Redundant edges

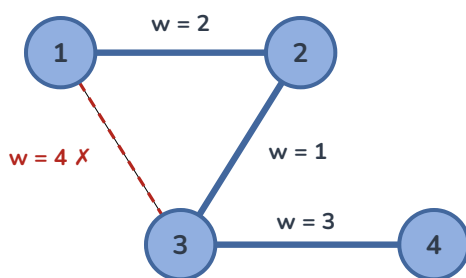
Process the edges of a graph in order. An edge is **redundant** if its endpoints are already connected by earlier edges, so adding it creates a cycle and does not change connectivity; a self-loop $v-v$ is always redundant. With disjoint sets this is one comparison: if $\text{find}(u) = \text{find}(v)$ the edge is redundant, otherwise unite the two groups. The structure holds exactly the connectivity of the earlier edges, which is precisely what the definition asks about.

Redundant edges are the key to the next section. Every non-redundant edge reduces the number of components by one, so in a connected graph with n vertices exactly $n - 1$ edges are not redundant: they form a **spanning tree**, a set of edges that connects everything without any cycle. Parallel edges and self-loops need no special care in either problem: once the endpoints are joined, every further copy is redundant, and a self-loop always is.

3.2.8 Kruskal's algorithm for a minimum spanning tree

Each edge now has a cost w , for instance the price of laying a cable, and the goal is to connect all n vertices as cheaply as possible. The cheapest connecting set never contains a cycle (dropping any edge of a cycle keeps everything connected and does not cost more), so it is a spanning tree of minimum total weight: a **minimum spanning tree**.

Kruskal's algorithm is a greedy method (Chapter 2.5) powered by disjoint sets. Sort the edges by weight, as in Chapter 1.5. Go through them from cheapest to most expensive, and accept an edge exactly when its endpoints are in different components; otherwise it is redundant and rejected. Stop when $n - 1$ edges have been accepted.



edge	w	find(u), find(v)	decision
2-3	1	$2 \neq 3$	accept
1-2	2	$1 \neq 2$	accept
3-4	3	$2 \neq 4$	accept
1-3	4	$2 = 2$	reject: cycle

three accepted edges join 4 vertices: total $1 + 2 + 3 = 6$

Figure 3.2.3 — An edge connecting two different components can extend the minimum spanning forest. Edge 1–3 comes last and is rejected: 1–2 and 2–3 already connect its ends.

Why the cheapest crossing edge is safe

Suppose the edges accepted so far are part of some minimum spanning tree T , and Kruskal now accepts $e = u-v$. Let S be the current component of u ; e leaves S . If e is in T , fine. Otherwise adding e to T closes a cycle, and that cycle must leave S a second time, through some edge f of T . Edge f was not examined yet: every earlier edge was either accepted (and lies inside a component) or rejected (and joined two vertices of one component), so no earlier edge leaves S . Hence $w(f) \geq w(e)$.

Swapping f for e gives a spanning tree that is no heavier and contains everything accepted so far.

Repeating this for each accepted edge shows that Kruskal's tree is a minimum spanning tree. This is the exchange argument of Chapter 2.5 again.

Disconnected graphs and sizes

If the graph is not connected, fewer than $n - 1$ edges are accepted, and no spanning tree exists: the accepted edges form a **minimum spanning forest**, one tree per component. A program must count the accepted edges and report “impossible” (for example -1) rather than printing the forest's weight. A graph with one vertex needs no edges and costs 0.

Sorting costs $O(m \log m)$, and the m finds add $O(m \alpha(n))$. The total weight can reach $(n - 1) \times 10^9 \approx 2 \times 10^{14}$ with 200,000 vertices: use 64-bit integers (`long long`, `long`). JavaScript Numbers are exact below $2^{53} \approx 9 \times 10^{15}$, so plain addition is safe.

3.2.9 Common mistakes

- **Adding sizes for an already joined pair.** Check $\text{find}(u) \neq \text{find}(v)$ first. Otherwise sizes double-count and the component count goes wrong.
- **Linking vertices instead of roots.** Writing $\text{parent}[u] \leftarrow v$ detaches u from its old tree and splits a group. Always link $\text{find}(u)$ and $\text{find}(v)$.
- **Reporting a disconnected graph as a spanning tree.** Count accepted edges; fewer than $n - 1$ means there is no spanning tree.
- **Claiming $O(1)$ worst case for find.** The near-constant bound is amortized over many operations; a single call can be slower.
- **Using a representative's number as information.** Which vertex becomes the root depends on the linking rule; compare roots, do not interpret them.

3.2.10 What disjoint sets cannot do

A disjoint-set structure only ever merges. A plain one cannot delete an edge, because a tree of parent pointers does not remember which edges caused a merge; splitting a group would need information it has thrown away. It also answers only “same group or not”: it knows nothing about distances (use BFS, Chapter 3.1, or Dijkstra, Chapter 3.3) or about one-way reachability in directed graphs (Chapter 4.4). Variants that undo operations or support deletions exist, but they are beyond this library.

Exercises

Difficulty: ● quick check, ●● needs an extra idea. Try each one before reading the answers.

1. ● Should $\text{union}(u, v)$ reduce the component count when $\text{find}(u) = \text{find}(v)$?
2. ● Can a plain disjoint-set structure delete an edge and still answer connectivity questions correctly?
3. ● With $n = 6$, apply $\text{union}(1, 2)$, $\text{union}(3, 4)$, $\text{union}(5, 6)$, $\text{union}(4, 2)$, using union by size with the first argument's root winning ties. Write the parent array and the number of components.
4. ●● Continue Exercise 3: what does $\text{find}(2)$ return, and what changes in the parent array because of path compression?
5. ●● Run Kruskal on the edges 1–2 (4), 2–3 (4), 3–4 (1), 1–4 (2), 2–4 (3), 1–3 (5). Which edges are accepted, and what is the total weight?
6. ●● Even with union by size and path compression, why is it wrong to say that every single find takes $O(1)$ time?

Answers and explanations

Read these after attempting the exercises.

1. No. The endpoints are already in one group, so nothing merges: the count and the sizes stay the same.
2. No. Parent pointers do not record which edge caused a merge, so the structure cannot tell whether the group would fall apart. Plain disjoint sets only support merging.
3. `union(1, 2): parent[2] = 1`. `union(3, 4): parent[4] = 3`. `union(5, 6): parent[6] = 5`. `union(4, 2):` roots 3 and 1, both of size 2, so the first argument's root 3 stays: `parent[1] = 3`. Parent array for 1..6: **[3, 1, 3, 3, 5, 5]**; **2 components**, of sizes 4 and 2.
4. The path is $2 \rightarrow 1 \rightarrow 3$, so `find(2)` returns **3**. Compression sets `parent[2] = 3`; `parent[1]` is already 3. The array becomes [3, 3, 3, 3, 5, 5].
5. In weight order: 3–4 (1) accept, 1–4 (2) accept, 2–4 (3) accept; now all four vertices are connected, and 1–2, 2–3 and 1–3 are rejected. Total **1 + 2 + 3 = 6**.
6. The bound $O(\alpha(n))$ is amortized: it holds for the average over a long sequence. A find that walks an uncompressed path can take up to about $\log_2 n$ steps, and only afterwards does that path become short.

Practice this topic

Four coding problems for this chapter are in the **Practice** tab. Open any of them in any order; the label shows difficulty within this topic.

Connection checks

Apply

Maintain n initially separate vertices. Type 1 joins u and v . Type 2 asks whether u and v are currently connected through joins.

Growing network

Connect

After each undirected edge addition, print the number of connected components and the size of the largest component.

Redundant links

Reason

Process undirected edges in input order. A link is redundant if its endpoints were already connected before adding it (including a self-loop). Count redundant links.

Least-cost backbone

Stretch

Choose edges connecting all n vertices with minimum total weight. Print -1 if connection is impossible. A one-vertex graph needs no edges and costs 0.
