Module 7 of 12

Disjoint set union (DSU)

Union-find in twelve lines: a parent array, path compression and union by size — components, cycle detection and Kruskal's minimum spanning tree at contest limits.

Many graph problems only ever ask one question while edges keep arriving: are these two vertices connected yet? Running a BFS per question is O(n) each time; a disjoint set union answers it in effectively constant time.

The lesson builds the contest DSU — a parent array where each group is a tree named by its root, `find` with path compression, `unite` by size — and animates both rules on seven elements, so you can see a tree flatten. Then the habits: `unite` returning `false` is a cycle test, a running component counter, and why the recursive `find` is safe once union by size bounds the height.

The exercises count components and the largest group after every union, and find the first edge that closes a cycle. The timed checkpoint is Kruskal's minimum spanning tree with weights up to 10^9 and a stream of online "same set?" queries.

CONTENTS

1	Disjoint set union — union by size and path compression	14 min
✓	Checkpoint — Disjoint set union	45 min

Exercises in this module compile on Clang 18 · C++20. Reference solutions and the answer key follow each lesson.

LESSON 1 · 14 MIN

Disjoint set union — union by size and path compression

A DSU answers one question fast while groups keep merging: **are `a` and `b` connected yet?** Contest uses: components after each edge, cycle detection, Kruskal's minimum spanning tree, offline "merge these, then ask".

The structure

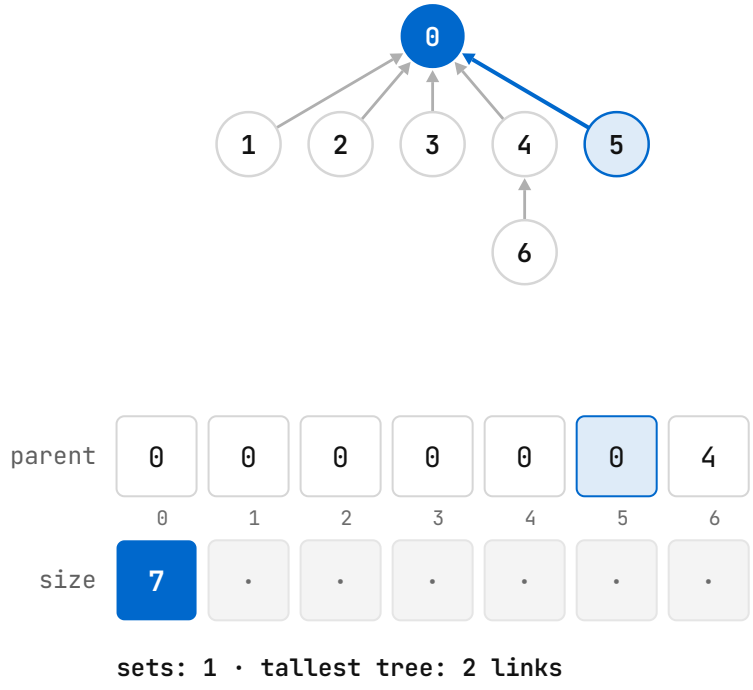
- Each group is a **tree**; `p[x]` is one step from `x` towards its root, and a root is its own parent.
- The root **names** the group: `a` and `b` are together exactly when `find(a) == find(b)`.
- `sz[r]` is the number of elements under root `r` — read only at a root.

```
struct DSU {
    vector<int> p, sz;
    DSU(int n) : p(n), sz(n, 1) { iota(p.begin(), p.end(), 0); }
    int find(int x) { return p[x] == x ? x : p[x] = find(p[x]); }
    bool unite(int a, int b) {
        a = find(a), b = find(b);
        if (a == b) return false;          // already together
        if (sz[a] < sz[b]) swap(a, b);     // a is the larger root
        p[b] = a;
        sz[a] += sz[b];
        return true;
    }
};
```

Two rules keep it fast

- **Union by size**: the smaller root goes under the larger. An element only sinks one level when its group at least doubles, so a tree is never taller than log2 n.
- **Path compression**: `p[x] = find(p[x])` re-points every node it walks straight at the root.
- Together: amortised O(α(n)) per call — effectively constant. Either one alone is already O(log n).

Union by size and path compression



1. Seven elements, seven sets: every element is its own root, so parent[i] = i and size[i] = 1. The size row is only read at a root.
2. unite(0, 1): Both sets have size 1, a tie, so 0 stays the root and size[0] becomes 2.
3. unite(2, 3): Both sets have size 1, a tie, so 2 stays the root and size[2] becomes 2.
4. unite(0, 2): Both sets have size 2, a tie, so 0 stays the root and size[0] becomes 4. Now 3 sits two links below its root — the only kind of path compression can shorten.
5. unite(4, 5): Both sets have size 1, a tie, so 4 stays the root and size[4] becomes 2.
6. unite(4, 6): The set of 6 is smaller, so its root 6 goes under 4 and size[4] becomes 3.
7. unite(3, 6) starts with find(3), which walks 3 → 2 → 0 to the root 0.
8. Path compression: on the way back every node of that walk points straight at 0, so parent[3] is now 0 and the next find(3) is one step.
9. find(6) is 6 → 4. The set of 0 has 4 elements and the set of 4 has 3, so 4 goes under 0: one set of 7.
10. A later find(5) walks 5 → 4 → 0 once and flattens it. Union by size keeps every tree at most log2 n tall; path compression flattens what is walked, and together a find costs almost O(1).

Contest habits

- `unite` returning `false` **is** the cycle test: the edge joined two vertices that were already connected.
- Keep a running `components` counter: start at `n`, subtract one on every successful `unite`.
- The recursive `find` is fine at 2 × 10^5 *with* union by size (depth ≤ 18). Without it, a chain of 10^6 can overflow the stack.
- Index from 0 or convert 1-based input once at read time — never mix.

More practice

- Number of Provinces
- Redundant Connection
- Number of Connected Components in an Undirected Graph
- Most Stones Removed with Same Row or Column
- Min Cost to Connect All Points

EXERCISE 1

Components after every union

There are `n` people, numbered `1` to `n`, each in a group of their own. `q` times, two people `u` and `v` join groups (if they are already in the same group, nothing changes). After every join, print two numbers: how many groups there are, and the size of the largest group.

Input: `n q`, then `q` lines `u v` (`1 ≤ n ≤ 2·10^5`, `1 ≤ q ≤ 2·10^5`, `1 ≤ u, v ≤ n`; `u = v` is allowed). **Output:** `q` lines `<groups> <largest>`.

```
6 5
1 2
3 4
2 1
2 4
5 6
```

prints

```
5 2
4 2
4 2
3 4
2 4
```

INPUT 1

```
6 5
1 2
3 4
2 1
2 4
5 6
```

OUTPUT 1

```
5 2
4 2
4 2
3 4
2 4
```

REFERENCE SOLUTION

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

struct DSU {
    vector<int> p, sz;
    DSU(int n) : p(n + 1), sz(n + 1, 1) { iota(p.begin(), p.end(), 0); }
    int find(int x) { return p[x] == x ? x : p[x] = find(p[x]); }
    bool unite(int a, int b) {
        a = find(a), b = find(b);
        if (a == b) return false;
        if (sz[a] < sz[b]) swap(a, b);
        p[b] = a;
        sz[a] += sz[b];
        return true;
    }
};

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n, q;
    cin >> n >> q;
    DSU d(n);
    int groups = n, largest = 1;
    while (q--) {
        int u, v;
        cin >> u >> v;
        if (d.unite(u, v)) {
            --groups;
            largest = max(largest, d.sz[d.find(u)]);
        }
        cout << groups << ' ' << largest << '\n';
    }
    return 0;
}
```

EXERCISE 2

The first edge that closes a cycle

An undirected graph on vertices `1` to `n` receives `m` edges one at a time. Print the 1-based index of the first edge whose two endpoints were already connected before it arrived — the first edge that closes a cycle. A self-loop `v v` closes a cycle at once. Print `-1` when no edge does.

Input: `n m`, then `m` lines `u v` (`1 ≤ n ≤ 2·10^5`, `0 ≤ m ≤ 2·10^5`). **Output:** one integer.

```
5 6
1 2
2 3
4 5
3 4
1 3
2 5
```

prints

```
5
```

Edge 5 (`1 3`) arrives after `1 2` and `2 3` have already connected 1 and 3.

INPUT 1

```
5 6
1 2
2 3
4 5
3 4
1 3
2 5
```

OUTPUT 1

```
5
```

REFERENCE SOLUTION

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

struct DSU {
    vector<int> p, sz;
    DSU(int n) : p(n + 1), sz(n + 1, 1) { iota(p.begin(), p.end(), 0); }
    int find(int x) { return p[x] == x ? x : p[x] = find(p[x]); }
    bool unite(int a, int b) {
        a = find(a), b = find(b);
        if (a == b) return false;
        if (sz[a] < sz[b]) swap(a, b);
        p[b] = a;
        sz[a] += sz[b];
        return true;
    }
};

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n, m;
    cin >> n >> m;
    DSU d(n);
    for (int i = 1; i <= m; ++i) {
        int u, v;
        cin >> u >> v;
        if (!d.unite(u, v)) {
            cout << i << '\n';
            return 0;
        }
    }
    cout << -1 << '\n';
    return 0;
}
```

QUIZ

1. After `p[x] = find(p[x])` runs on the walk `5 → 3 → 1 → 0` (0 the root), what is `p[5]`?

A. 3

B. 1

C. 0

D. 5

2. Why does union by size keep every tree at most log2 n tall?

A. An element only sinks one level when its group at least doubles, which can happen at most log2 n times

B. It sorts the elements before uniting them

C. Path compression is applied to every element after each union

D. It limits each root to two children

3. Edges arrive one at a time. Which test tells you edge `(u, v)` closes a cycle?

A. `sz[u] == sz[v]`

B. `find(u) == find(v)` before uniting

C. `p[u] == v`

D. `find(u) != find(v)` after uniting

4. Starting from `n` singletons, you perform `k` calls of `unite` and `s` of them return `true`. How many groups are there?

A. `n - k`

B. `n - s`

C. `k - s`

D. `n`

ANSWER KEY · DISJOINT SET UNION — UNION BY SIZE AND PATH COMPRESSION

1. C Path compression re-points every node it walked at the root, so `p[5]`, `p[3]` and `p[1]` all become `0`. `3` was the old parent; `5` would make 5 a root of its own.

2. A When `x`'s root goes under another, the new group is at least twice the size of `x`'s old one. A group cannot double more than log2 n times before holding all n elements. No sorting or child limit is involved, and the bound holds without path compression.

3. B Same root before the edge means u and v were already connected, so the edge adds a second route — a cycle. Equal sizes say nothing about connection; `p[u] == v` checks one direct link only; after uniting, the roots are always equal.

4. B Only a successful unite merges two groups into one; a failed one (already together) changes nothing. So the count is `n - s`, not `n - k`.

CHECKPOINT · 45 MIN

Checkpoint — Disjoint set union

A 45-minute timed checkpoint on module 7: the DSU, its two speed rules, and the problems it solves.

How it works. Ten questions and two programs, in 45 minutes from when you start the timer. 70% on the questions and both programs accepted clears the module.

Before you start, make sure you can answer these from memory:

- What `find` returns, and what path compression changes on the way back.
- Why union by size keeps every tree at most log2 n tall.
- What `unite` returning `false` means for an edge of a graph.
- How Kruskal's algorithm decides whether to take an edge, and how it knows the graph was disconnected.

The programs are Kruskal's minimum spanning tree with weights up to 10^9, and a stream of union and "same set?" queries answered online.

EXERCISE 1

Kruskal's minimum spanning tree

A weighted undirected graph has vertices `1` to `n` and `m` edges. Print the total weight of a minimum spanning tree, or `IMPOSSIBLE` when the graph is not connected. (With `n = 1` the tree has no edges and weighs `0`.)

Input: `n m`, then `m` lines `u v w` (`1 ≤ n ≤ 10^5`, `0 ≤ m ≤ 2·10^5`, `1 ≤ w ≤ 10^9`; self-loops and parallel edges may appear). **Output:** one line.

```
4 5
1 2 3
2 3 1
3 4 4
1 4 2
1 3 5
```

prints

```
6
```

The tree takes `2-3` (1), `1-4` (2) and `1-2` (3).

INPUT 1

```
4 5
1 2 3
2 3 1
3 4 4
1 4 2
1 3 5
```

OUTPUT 1

```
6
```

REFERENCE SOLUTION

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

struct DSU {
    vector<int> p, sz;
    DSU(int n) : p(n + 1), sz(n + 1, 1) { iota(p.begin(), p.end(), 0); }
    int find(int x) { return p[x] == x ? x : p[x] = find(p[x]); }
    bool unite(int a, int b) {
        a = find(a), b = find(b);
        if (a == b) return false;
        if (sz[a] < sz[b]) swap(a, b);
        p[b] = a;
        sz[a] += sz[b];
        return true;
    }
};

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n, m;
    cin >> n >> m;
    vector<array<int, 3>> edges(m);
    for (auto& [w, u, v] : edges) cin >> u >> v >> w;
    sort(edges.begin(), edges.end());
    DSU d(n);
    ll total = 0;
    int taken = 0;
    for (auto& [w, u, v] : edges)
        if (d.unite(u, v)) {
            total += w;
            ++taken;
        }
    if (taken == n - 1) cout << total << '\n';
    else cout << "IMPOSSIBLE\n";
    return 0;
}
```

EXERCISE 2

Union and query, online

There are `n` elements `1` to `n`, each in its own set. Process `q` operations in order:

- `U a b` — merge the sets of `a` and `b`;
- `Q a b` — print `YES` if `a` and `b` are in the same set right now, `NO` otherwise.

Input: `n q`, then `q` lines (`1 ≤ n, q ≤ 2·10^5`). There is at least one `Q`. **Output:** one line per `Q`.

```
5 6
Q 1 2
U 1 2
U 3 4
Q 2 1
U 2 4
Q 1 3
```

prints

```
NO
YES
YES
```

INPUT 1

```
5 6
Q 1 2
U 1 2
U 3 4
Q 2 1
U 2 4
Q 1 3
```

OUTPUT 1

```
NO
YES
YES
```

REFERENCE SOLUTION

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

struct DSU {
    vector<int> p, sz;
    DSU(int n) : p(n + 1), sz(n + 1, 1) { iota(p.begin(), p.end(), 0); }
    int find(int x) { return p[x] == x ? x : p[x] = find(p[x]); }
    bool unite(int a, int b) {
        a = find(a), b = find(b);
        if (a == b) return false;
        if (sz[a] < sz[b]) swap(a, b);
        p[b] = a;
        sz[a] += sz[b];
        return true;
    }
};

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n, q;
    cin >> n >> q;
    DSU d(n);
    while (q--) {
        char op;
        int a, b;
        cin >> op >> a >> b;
        if (op == 'U') d.unite(a, b);
        else cout << (d.find(a) == d.find(b) ? "YES" : "NO") << '\n';
    }
    return 0;
}
```

CHECKPOINT QUESTIONS

1. In the contest DSU, what does `p[r] == r` mean?

A. r is a root and names its set

B. r has no elements under it

C. r has been deleted

D. r was never united

2. What is the amortised cost per operation of a DSU with both path compression and union by size?

A. O(log n)

B. O(α(n)), effectively constant

C. O(1) worst case for every call

D. O(n)

3. Which `unite` makes a chain of length n when called as `unite(2,1), unite(3,2), …, unite(n,n-1)`?

A. One that always hangs the second argument's root under the first's

B. One that hangs the smaller set's root under the larger's

C. One with path compression and union by size

D. None of them can build a chain

4. `sz[x]` is correct for which elements?

A. Every element

B. Only roots

C. Only leaves

D. Only elements united at least once

5. Kruskal's algorithm on a graph with n vertices took only n − 3 edges. What do you know?

A. The graph has a cycle

B. The graph has exactly 3 connected components

C. The tree is still minimum; print the total

D. An edge weight was negative

6. Why does the recursive `find` not overflow the stack at n = 2·10^5 here?

A. The compiler always turns it into a loop

B. Union by size keeps the depth at most about log2 n ≈ 18

C. Path compression runs before the recursion

D. The stack is unlimited on the judge

7. Edges `(1,2) (2,3) (3,1) (4,5)` arrive in that order. Which edge is the first to close a cycle?

A. the 2nd

B. the 3rd

C. the 4th

D. none

8. Kruskal sorts edges of weight up to 10^9 on n = 10^5 vertices. Which type should the total be?

A. int

B. long long

C. double

D. unsigned int

9. You need the number of components after each of q edge additions. What is the total cost with a DSU?

A. O(q · n)

B. O(q α(n)) plus O(n) to initialise

C. O(n^2)

D. O(q log q) for sorting

10. Which problem can a plain DSU NOT answer as edges are added?

A. Are u and v connected?

B. How large is u's component?

C. Did this edge close a cycle?

D. What is the shortest path from u to v?

ANSWER KEY · CHECKPOINT — DISJOINT SET UNION

1. A A root is its own parent; it is the set's name. It may have many elements under it, and it may have been the winning side of many unites.

2. B The inverse Ackermann bound α(n) is at most 4 for any realistic n. A single call can still walk a few links, so it is amortised, not worst-case O(1); either rule alone gives O(log n).

3. A Always putting `find(b)` under `find(a)` makes 1 under 2, 2 under 3, … — a path. Union by size would keep hanging the singleton under the big root, giving a star.

4. B `unite` updates the size only at the surviving root; a non-root's `sz` is a stale value from when it was a root. Always read `sz[find(x)]`.

5. B Each taken edge merges two components: n − (n − 3) = 3 components remain, so there is no spanning tree. Cycles only cause rejected edges, and negative weights are fine for Kruskal.

6. B With union by size no walk is longer than about 18 links, so the recursion is shallow. The recursive form is not tail-recursive (it assigns after the call), and stacks are limited.

7. B After `(1,2)` and `(2,3)`, vertices 1 and 3 share a root, so `(3,1)` — the 3rd edge — finds them already connected.

8. B Up to 10^5 − 1 edges of 10^9 sum to about 10^14, far past int and unsigned int; double would lose exactness past 2^53 in general. long long holds it exactly.

9. B Each addition is one unite and a counter update — near-constant time. No sorting is needed, and no per-query BFS.

10. D A DSU only knows which component an element is in and how big it is. Distances need a traversal — BFS or Dijkstra, the next module.

CodeKairo · Competitive Programming study plan · Module 7: Disjoint set union (DSU). Solve the exercises online to track your progress and earn XP.