UNION FIND DATA STRUCTURE

Davide Cologni

(davide.cologni@unive.it)

Cooperative Problem Solving and Competitive Programming

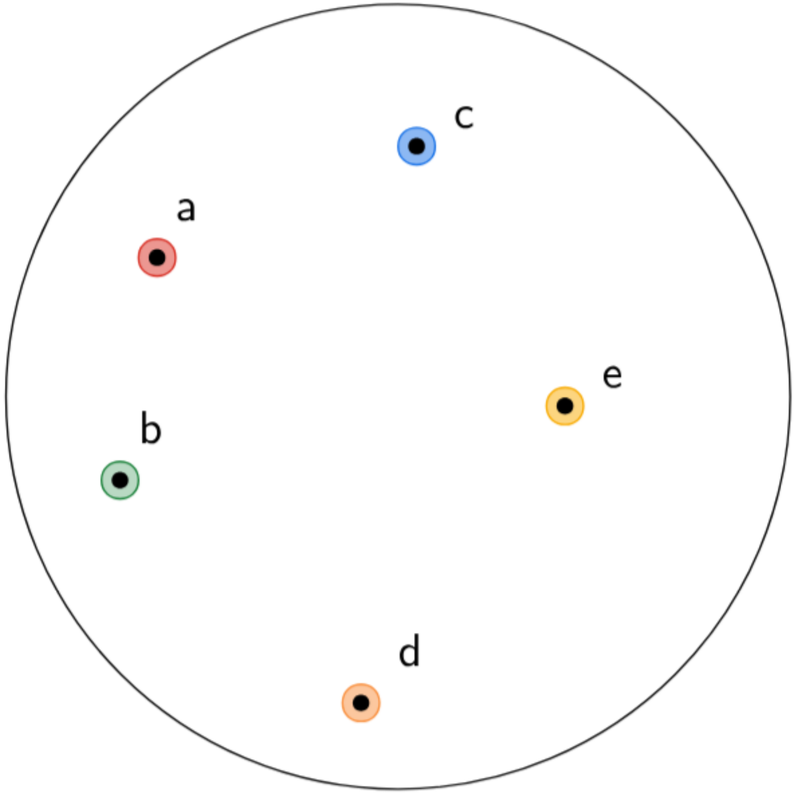
THE PROBLEM

You have a static set of n items and you want to represent a partition of it.

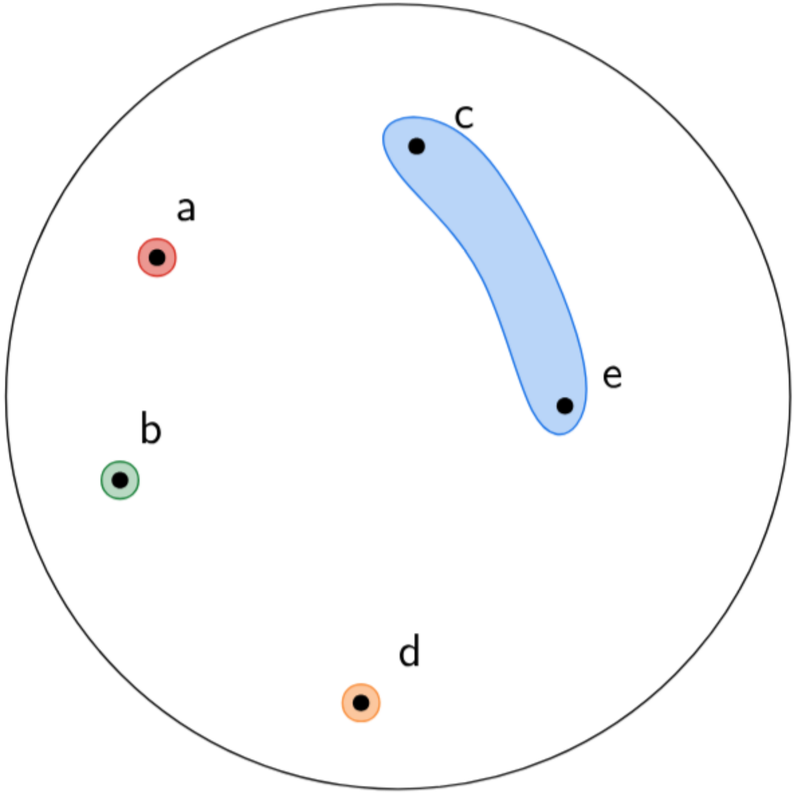
You need to perform two operations efficiently:

- **union(a, b):** Join the two partitions that contains these elements.
- **find(a, b):** Are these two elements in the same partition?

THE PROBLEM

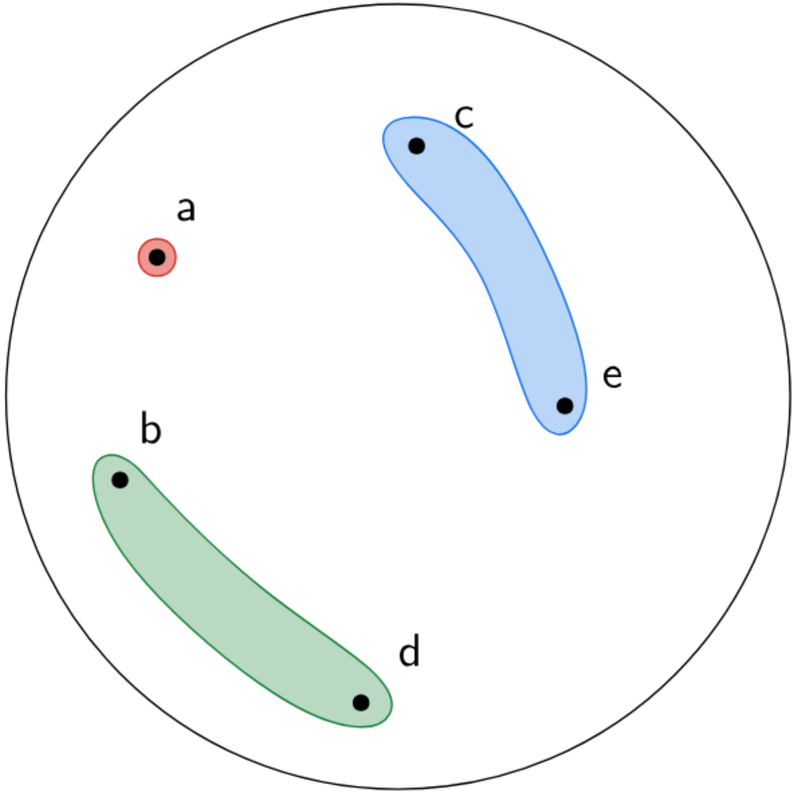


THE PROBLEM



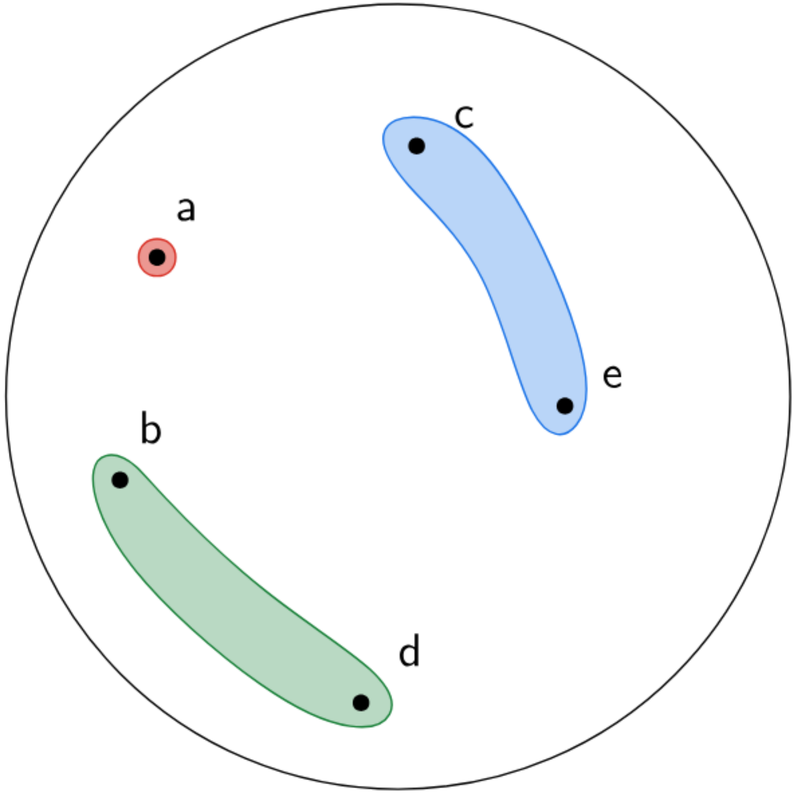
union(c, e)

THE PROBLEM



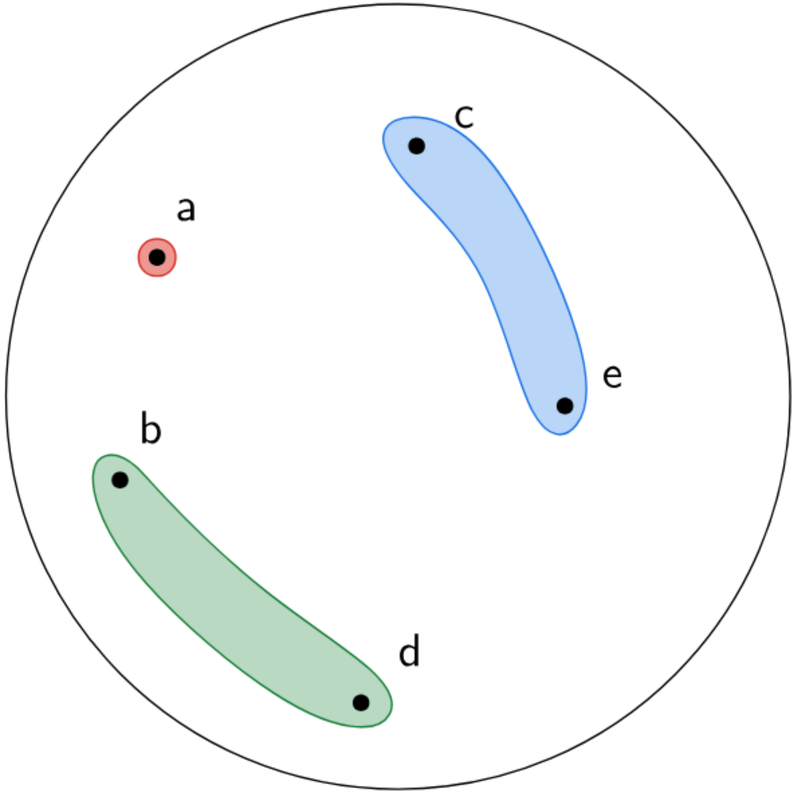
union(b, d)

THE PROBLEM



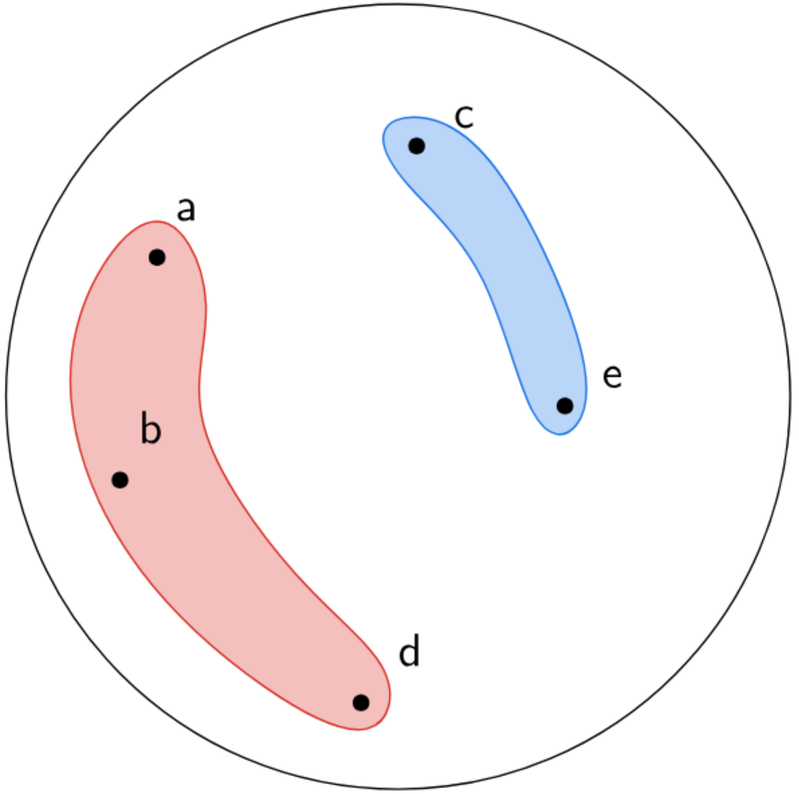
find(b, d) = True

THE PROBLEM



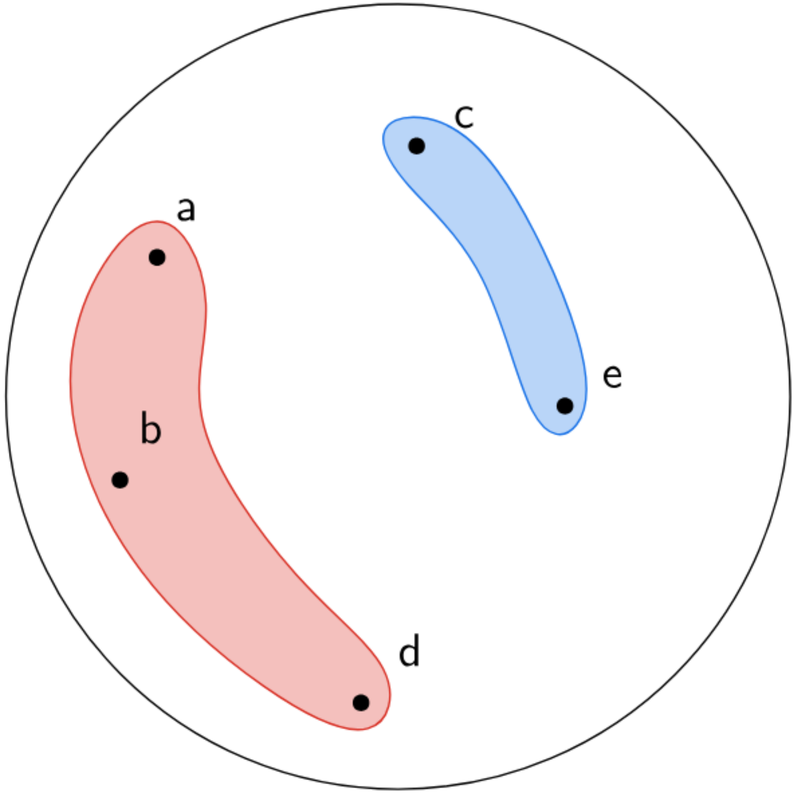
find(a, b) = False

THE PROBLEM



union(a, d)

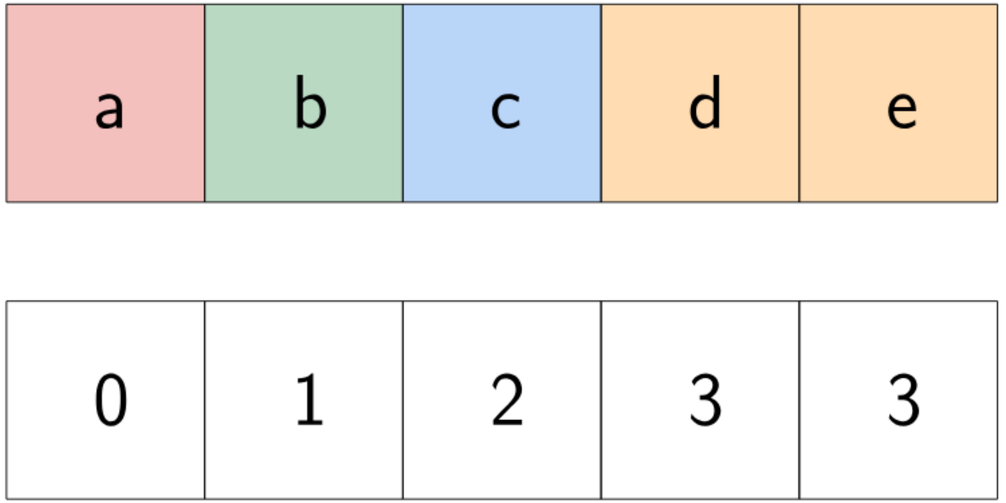
THE PROBLEM



find(a, b) = True

NAIVE APPROACH

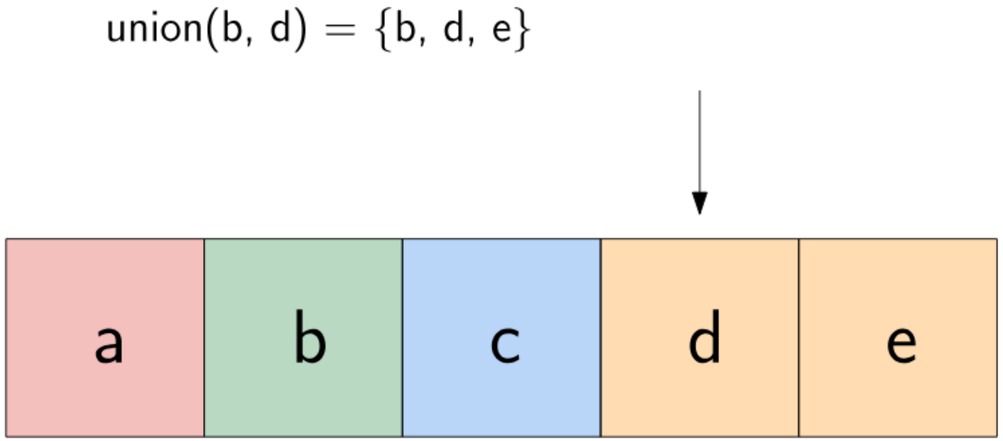
Keep an array: `group[i]` = the group ID of element `i`.



NAIVE APPROACH

Find is instant — just compare two values.

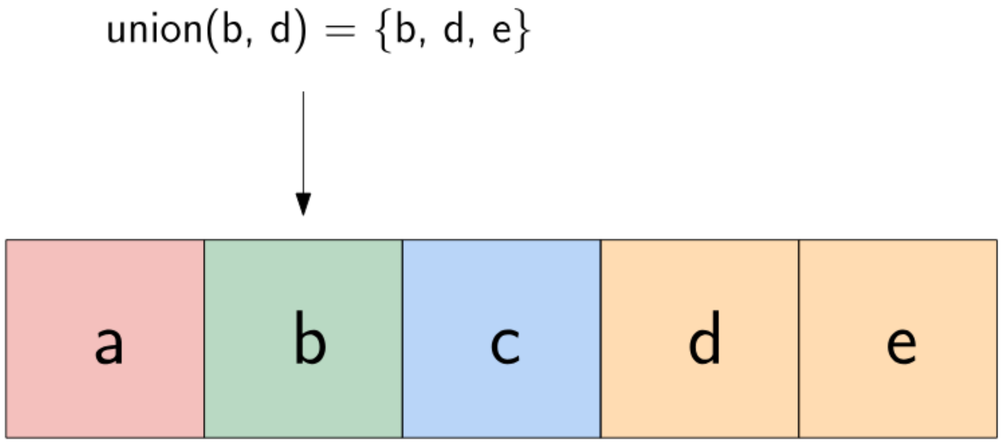
Union is slow $O(n)$ — you must search and rewrite every element of one group.



NAIVE APPROACH

Find is instant — just compare two values.

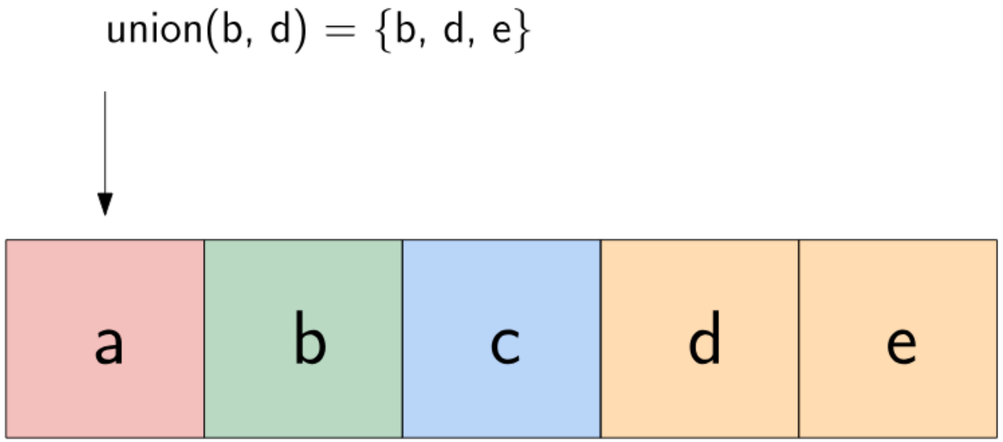
Union is slow $O(n)$ — you must search and rewrite every element of one group.



NAIVE APPROACH

Find is instant — just compare two values.

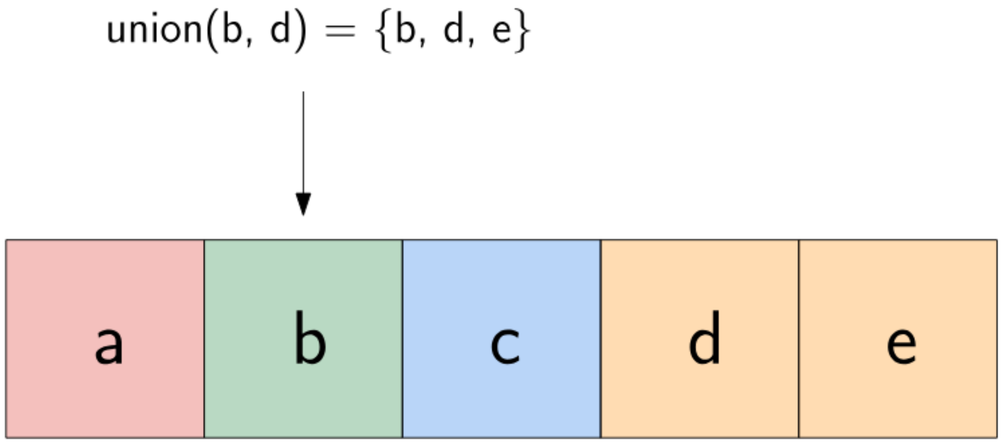
Union is slow $O(n)$ — you must search and rewrite every element of one group.



NAIVE APPROACH

Find is instant — just compare two values.

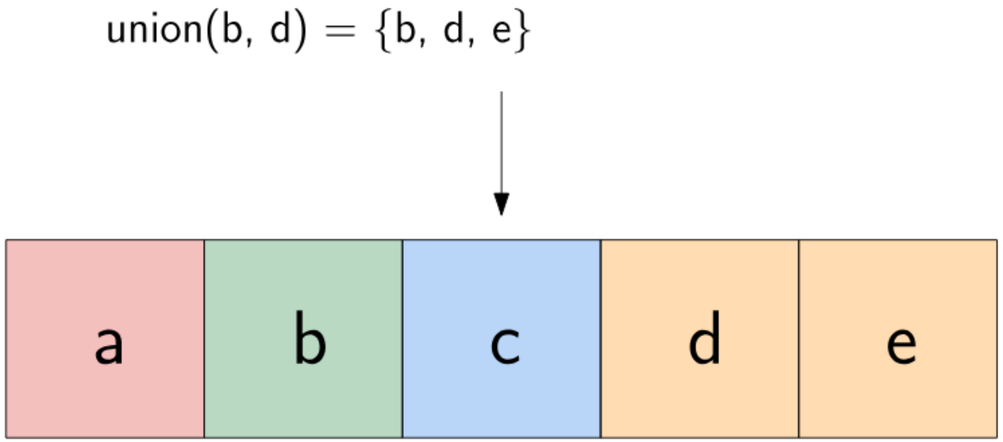
Union is slow $O(n)$ — you must search and rewrite every element of one group.



NAIVE APPROACH

Find is instant — just compare two values.

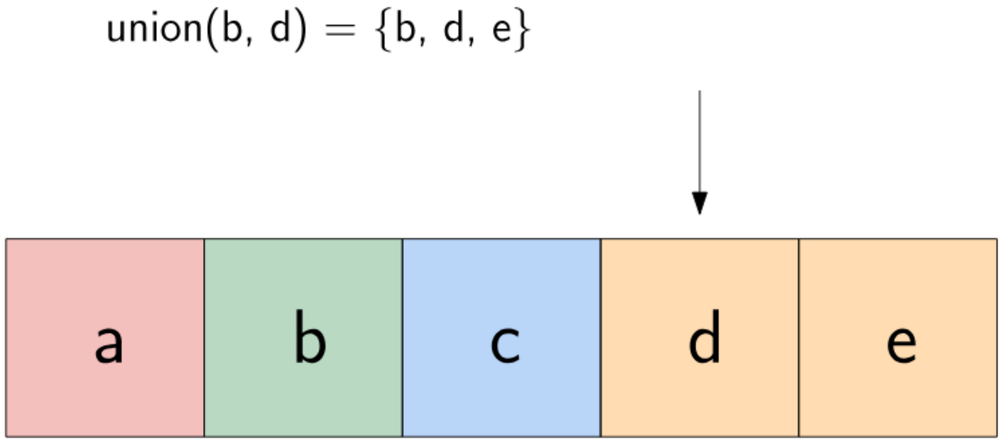
Union is slow $O(n)$ — you must search and rewrite every element of one group.



NAIVE APPROACH

Find is instant — just compare two values.

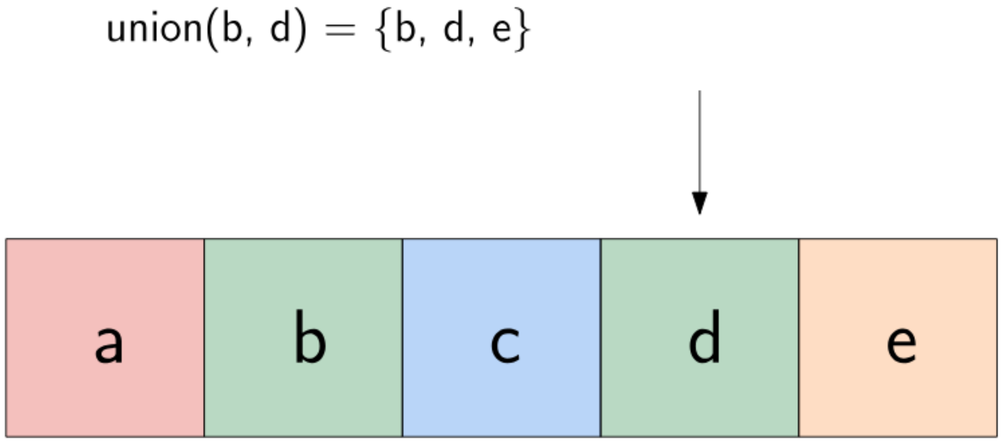
Union is slow $O(n)$ — you must search and rewrite every element of one group.



NAIVE APPROACH

Find is instant — just compare two values.

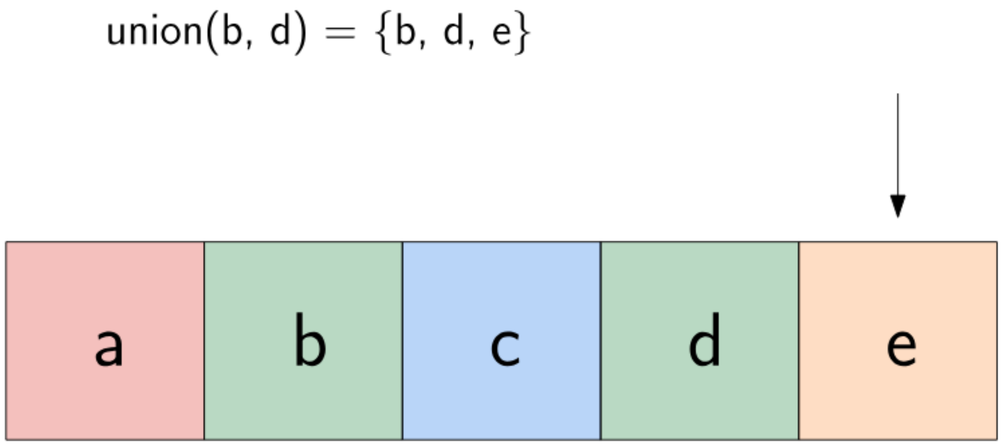
Union is slow $O(n)$ — you must search and rewrite every element of one group.



NAIVE APPROACH

Find is instant — just compare two values.

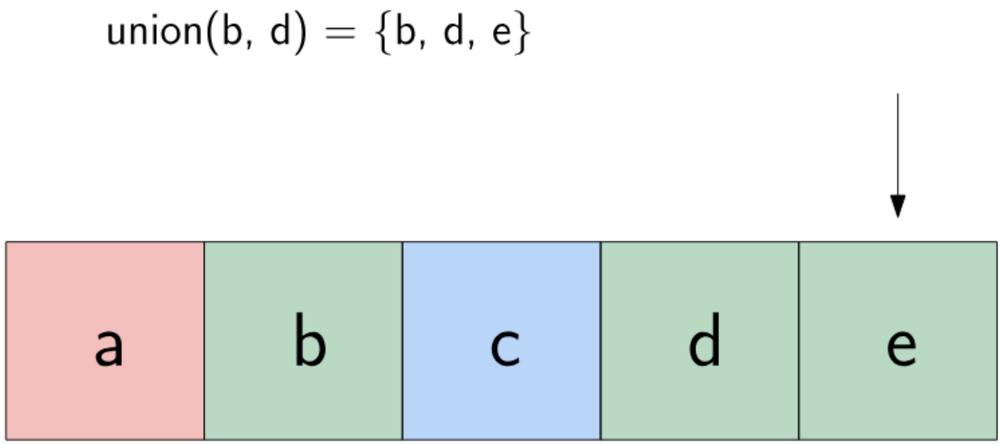
Union is slow $O(n)$ — you must search and rewrite every element of one group.



NAIVE APPROACH

Find is instant — just compare two values.

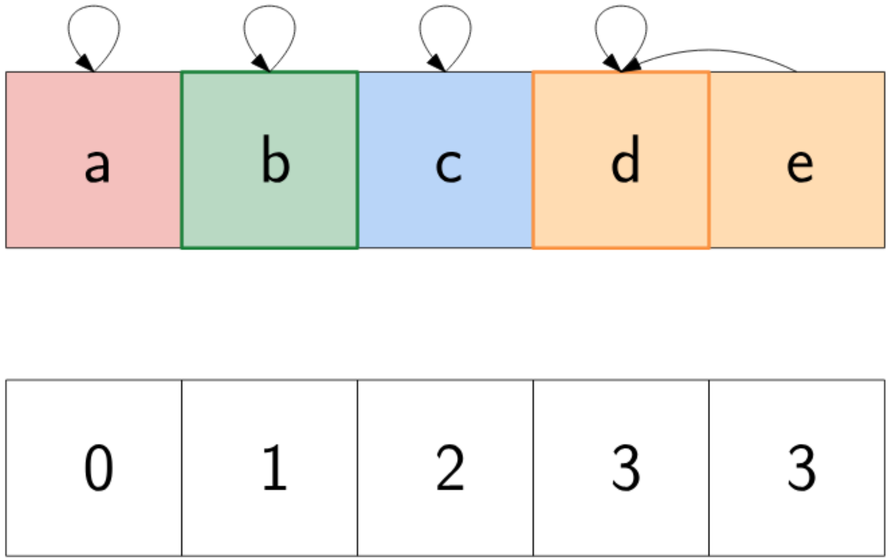
Union is slow $O(n)$ — you must search and rewrite every element of one group.



REPRESENTATIVES

What if each element just points to **another element in its group**?

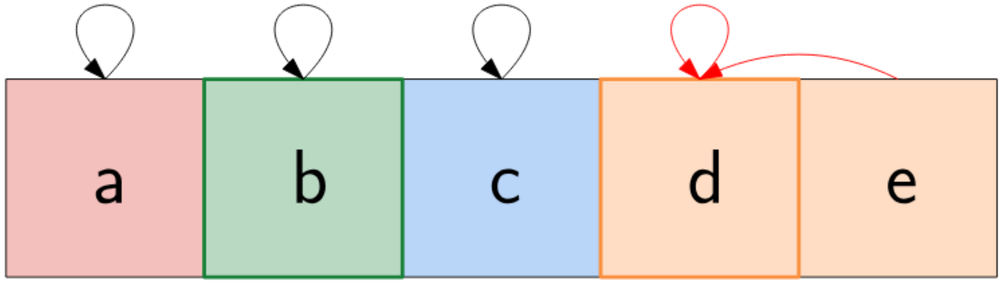
One element per group points to itself — that's the **representative**.



REPRESENTATIVES

Find: the elements are in the same set if they have the same *representative*. To find it, recursively follow the parent pointers.

find(b, e)
representative(b) == representative(e)
b == d
False



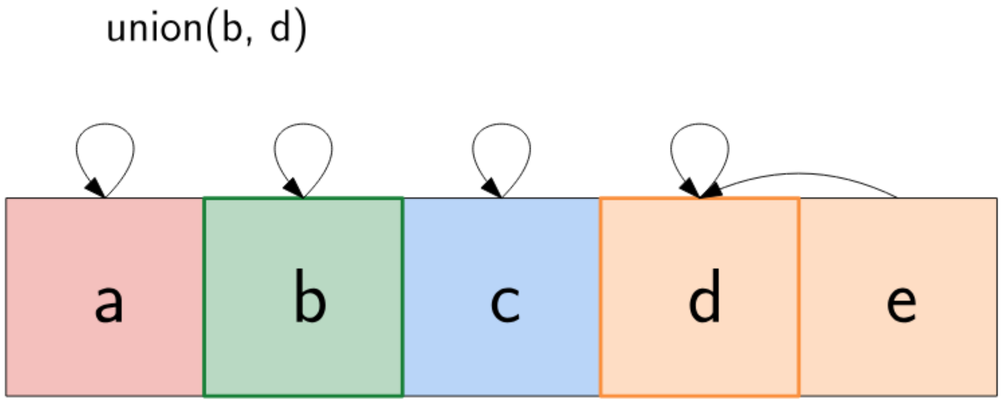
REPRESENTATIVES

```
fn representative(self, item) {
    if self.parent[item] == item {
        return item
    }
    return self.representative(self.parent[item])
}

fn find(self, a, b) {
    return self.representative(a) == self.representative(b)
}
```

REPRESENTATIVES

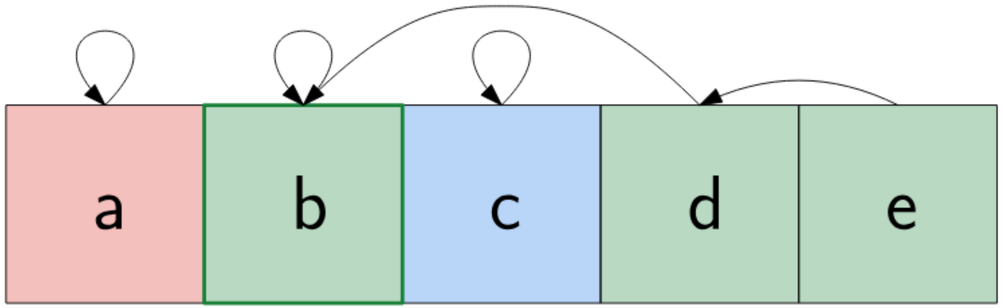
Union: Find both representatives, then choose one representative and set its parent to be the other one. A single operation.



REPRESENTATIVES

Union: Find both representatives, then choose one representative and set its parent to be the other one. A single operation.

$\text{union}(b, d) = \{b, d, e\}$



THE PROBLEM WITH LONG CHAINS

Finding the representative of `e` now means following the entire chain.

The longer the chain, the slower every find becomes.

We store the `rank` of every representative: the length of the longest path to it.

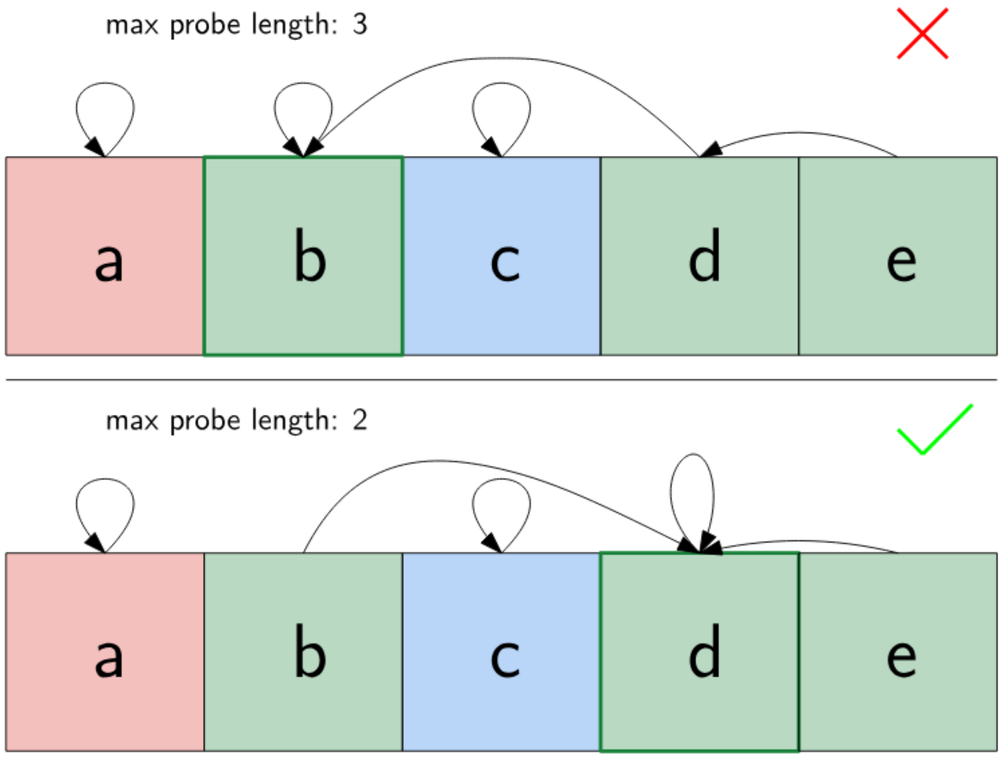
FIX 1: UNION BY RANK

We store the `rank` of every representative: the length of the longest path to it.

When merging, always attach the **shorter chain under the root of the longer one**.

This keeps the chains from growing deep.

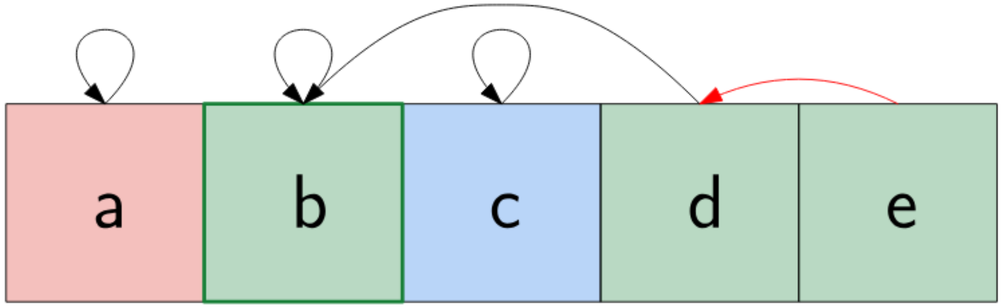
FIX 1: UNION BY RANK



FIX 2: PATH COMPRESSION

When you follow a chain to find the root, **repoint every visited element directly to the root**.

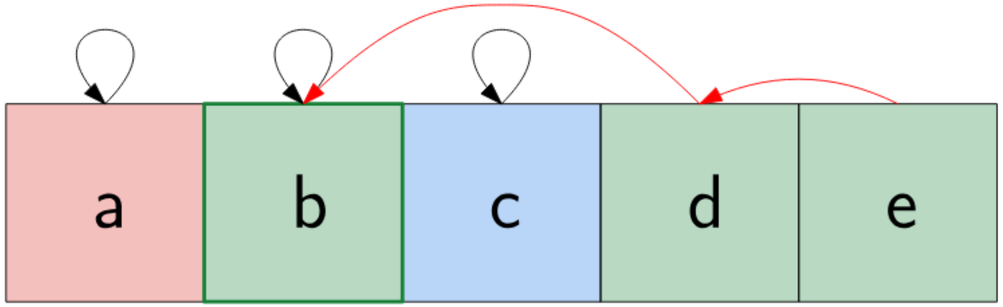
The next find on any of those elements is instant.



FIX 2: PATH COMPRESSION

When you follow a chain to find the root, **repoint every visited element directly to the root**.

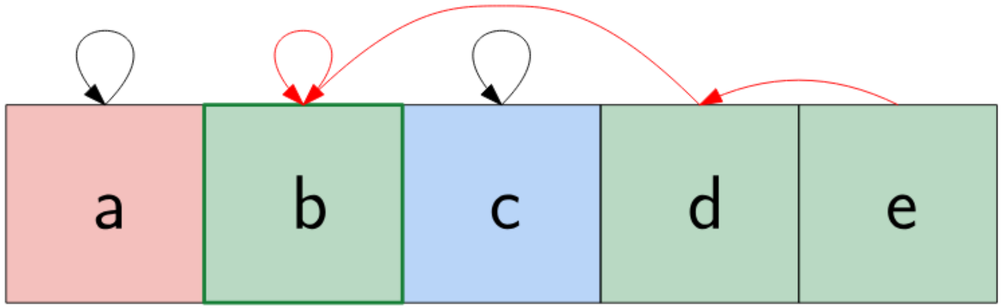
The next find on any of those elements is instant.



FIX 2: PATH COMPRESSION

When you follow a chain to find the root, **repoint every visited element directly to the root**.

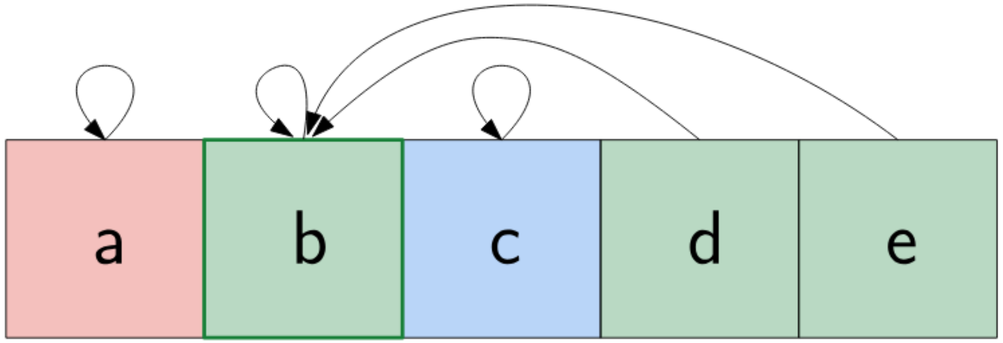
The next find on any of those elements is instant.



FIX 2: PATH COMPRESSION

When you follow a chain to find the root, **repoint every visited element directly to the root**.

The next find on any of those elements is instant.



AMORTIZED COST

Path compression does extra work the first time you traverse a chain — but every subsequent find on those same elements is O(1).

AMORTIZED COST

No single operation is guaranteed $O(1)$, but the amortized cost per operation is $O(\alpha(n))$ — the inverse Ackermann function — which stays below 5 for any input size you will ever encounter in practice.

SUMMARY

Operation	Naive array	Union-Find
Find	O(1)	~O(1)
Union	O(n)	~O(1)

Both improvements follow from the same idea: use pointers to defer work, and repair the structure lazily as you go.

A WARMUP: SATISFIABILITY OF EQUALITY CONSTRAINTS

You are given a list of constraints on variables: either $x_i = x_j$ or $x_i \neq x_j$.

Is there an assignment of natural numbers to variables that satisfies **all** constraints at once?

(Hint: You only need to check for the existence of an assignment; you don't need to find it)

EXAMPLE

Constraints:

$$x_1 = x_2$$
$$x_2 = x_3$$
$$x_1 \neq x_4$$

The equations are satisfiable with:

$$x_1 = 1$$
$$x_2 = 1$$
$$x_3 = 1$$
$$x_4 = 2$$

EXAMPLE 2

Constraints:

$$x_1 = x_2$$
$$x_2 = x_3$$
$$x_1 \neq x_3$$

In this case, there is no assignment that can satisfy all the equations.

SOLUTION

First, group the variables in the same partition for all equality conditions, i.e., $x_i = x_j$.

$$\texttt{union}(x_i, x_j)$$

Then, verify that all inequalities ($x_i \neq x_j$) refer to different groups of variables.

$$\texttt{!find}(x_i, x_j)$$

SECOND PROBLEM

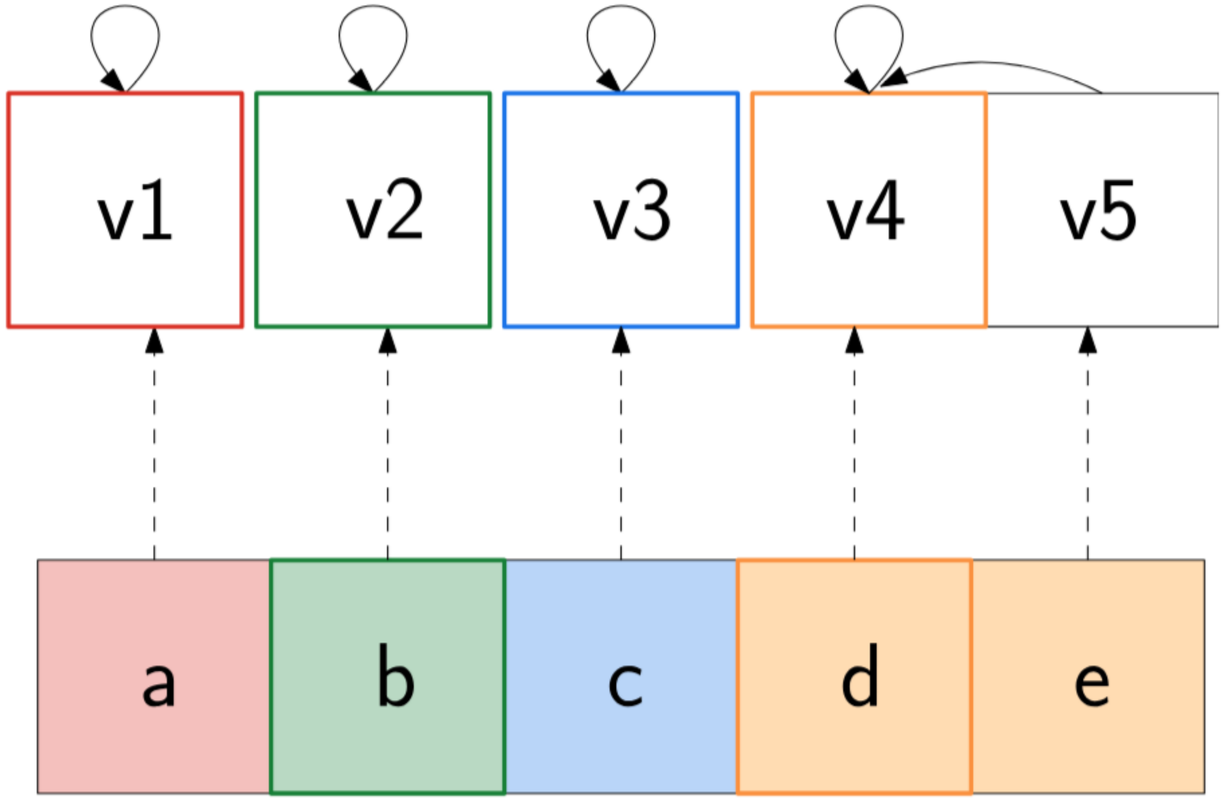
https://open.kattis.com/problems/almostunionfind

SOLUTION

The hard part of this problem is to implement the `move` operation.

SOLUTION

Create an array of “virtual” nodes. Every virtual node points to another virtual element and they behave exactly as the previous union find.



SOLUTION

To `move(e,c)`, find the `c` representative and create a new virtual node linked to it. Then, make `e` link to the new virtual node.