Today we're going to talk about a super important structure in graph theory that shows up again and again in algorithm problems — **Union-Find** (also known as Disjoint Set Union).

What is union-find?

In graph problems, we often face questions like:

- Which nodes are connected?
- Are two nodes in the same group?
- How many disconnected parts (components) does a graph have?

At the key of these problems is one basic task: **merging groups and checking group membership**. And that's exactly what **Union-Find** (also called Disjoint Set Union, or DSU) is built for — efficiently managing these group relationships.

So, what exactly is Union-Find?

Union-Find is a data structure with two key operations, as its name suggests: union and find

- `union(x, y)` — Merge two elements into the same group. For example, in the island problem on LeetCode, we can merge adjacent pieces of land into one island.
- `find(x)` — Quickly find which group (or "leader") a particular element belongs to. It tells you the "root" or representative of the group.

To make Union-Find even faster, we usually add two classic optimizations:

- **Path Compression**: When searching for a representative of an element, all points on the search path are directly linked to the root node, making subsequent searches faster.
- **Union by Rank (or Size)**: When merging two trees, always attach the shorter tree under the taller one. This keeps the trees shallow, improving performance. I will leave this knowledge in next week.

With both optimizations, each operation becomes nearly constant time — technically **O(α(n))** ["Big O of alpha of n"], where α is the inverse Ackermann function. This grows so slowly that we treat it as **O(1)** in practice. That's why Union-Find is such a powerful tool for problems involving connected components.

Starting from the "simplest idea"

Before diving into the code, let's ask: Why do we even need Union-Find?

Imagine you've never heard of it before. Suppose we want to group these elements. What's the most basic way to do that?

The First naive idea I came out is to use a dictionary to map group IDs to their members

```
group_dict = {
    1: [0, 3, 5],
    2: [1, 4],
    3: [2]
}
```

This is simple and works fine if you just want to see what's in each group.

But now let’s say you want to know: "Which group does element 3 belong to?" To find out, you’d need to scan the whole dictionary:

```
for group_id, members in group_dict.items():
    if 3 in members:
        print(group_id)
```

In the worst case, that takes **O(n)** time — not ideal.

Now I will try my second idea: using an array to record group ID for each element.

```
belong = [0, 0, 1, 0, 2, 1]
```

Now looking up the group of element 3 is super fast — just check `belong[3]` — which takes **O(1)** time.

But what if we want to merge group 1 and group 0? We’d have to:

- Scan the entire array
- Change every value from 1 to 0

This is also a **O(n)** scan.

So here’s the problem: we want to support both **fast lookup** and **fast merging**, but our simple approaches can only give us one or the other — not both.

This is where Union-Find comes in.

Union-Find solves this by giving each group a leader. Each element points to its "boss", and the root boss represents the whole group. When we want to know which group an element belongs to, we walk up the chain to find its leader. When we want to merge two groups, we simply point one root to the other.

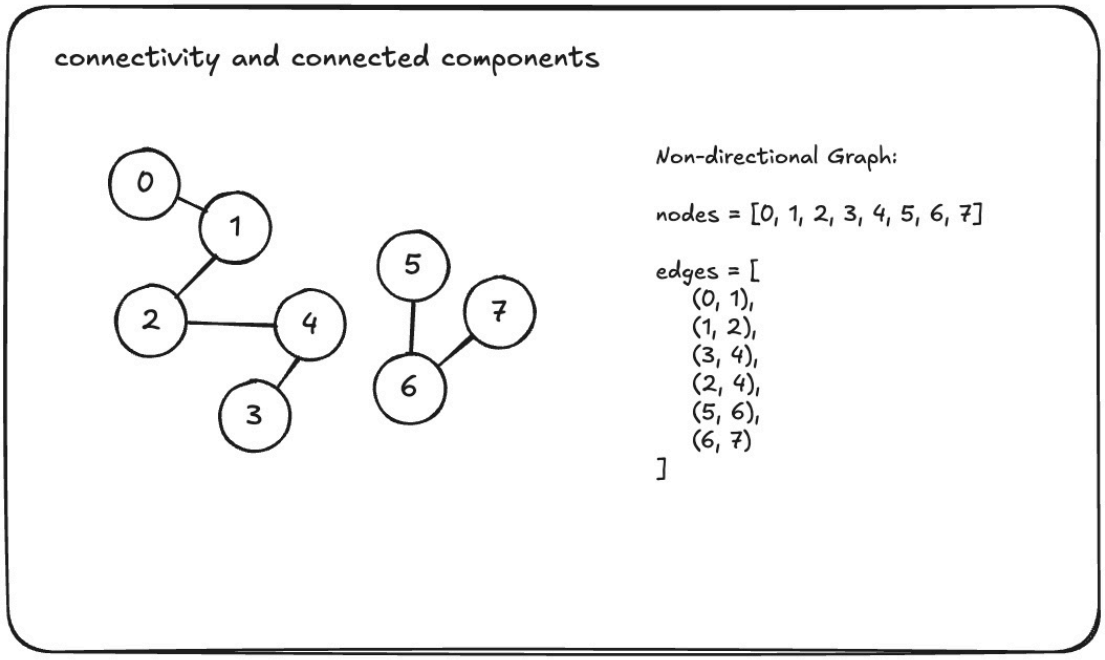
With this setup, both `find()` and `union()` are super fast — and the whole structure is much more efficient than our naive approaches.

Understand Process

Now Let's walk through what **Union-Find** (or DSU) is really doing — and how to build it — step by step, using diagrams and examples.

Connectivity & Connected Component

Before diving into Union-Find, let's first understand two key graph concepts:



We want to know how to use code to:
1. Group these nodes → find your biggest boss

2. Check if node a and node b are connected?
   How many connected components are in the graph? → if biggest boss is the same, these 2 persons are belong to the same gang

What is Connectivity?

In an **undirected graph**, two nodes are **connected** if there's a path between them — even if the path goes through multiple jumps. If nodes can reach each other, we say they belong to the same **connected region**.

What is a Connected Component?

A **connected component** is a group of nodes that are all reachable from one another. A graph may have several components — like islands, isolated groups, or subnetworks.

And that's where **Union-Find** comes in — it's a data structure built specifically to **track and maintain connected components** efficiently.

You can think of a graph as a company and the nodes as employees.

Each employee has a **boss**. By following the chain of command upward, you'll eventually reach the **CEO** — the top representative of that team (i.e., component). We can use an array like `boss[x]` to represent who x reports to. Then we can build and update this structure using Union-Find operations.

In real problems, we often care about two main things:

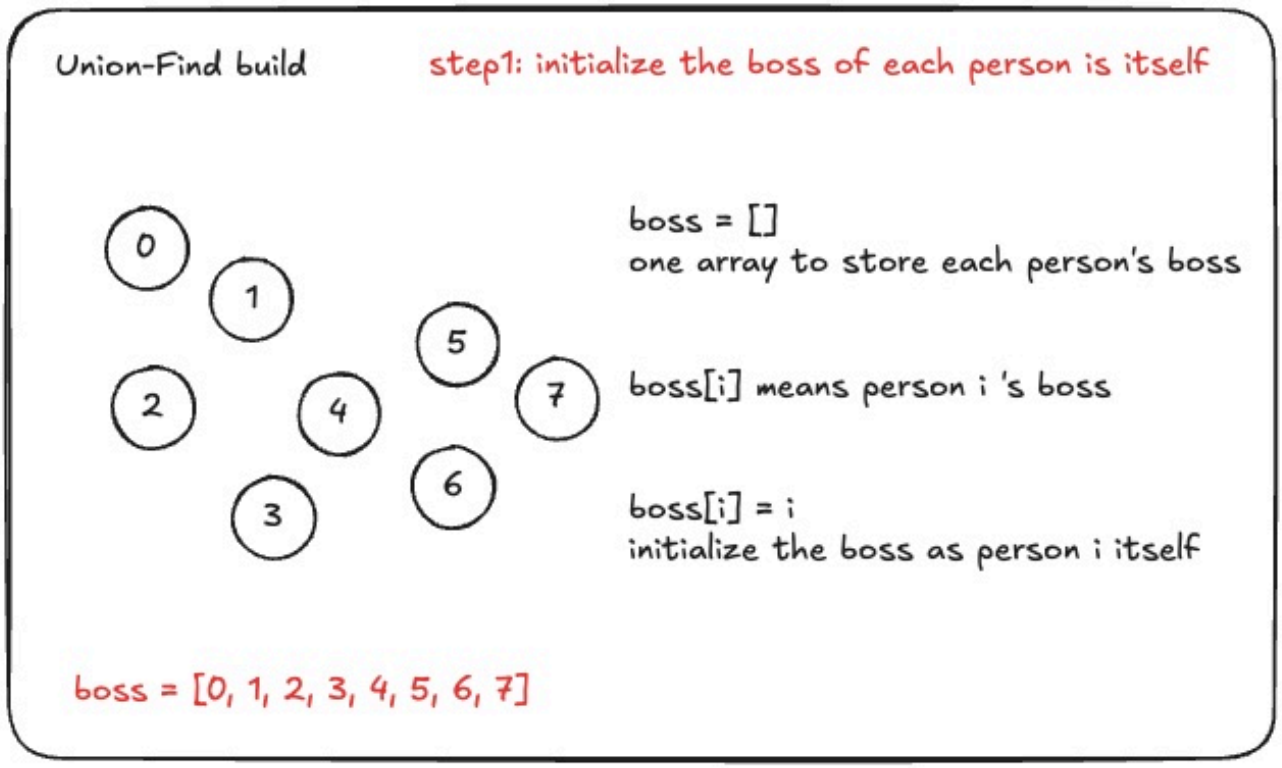
1. **How to group connected nodes together**
2. **How to check if two nodes belong to the same group (component)**

Explain Codes

Let's now break down the Union-Find code structure. Once you understand what each piece does, the whole thing becomes very natural to apply.

Step 1: Initialization

Initially, every node is isolated — no connections yet. So we create an array `boss[]`, where `boss[i] = i`, meaning every person is their own boss. In other words, each node starts out as a standalone group, like a one-person startup.



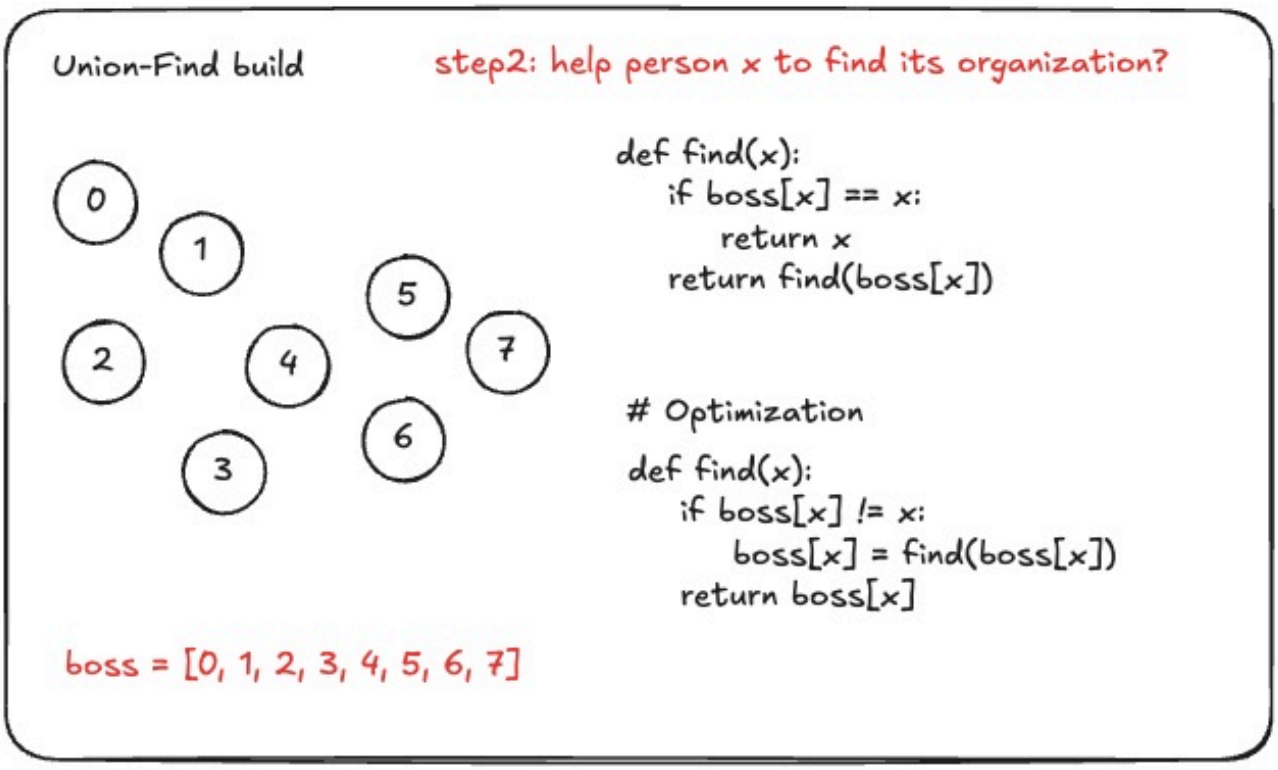
Step 2: The Find Operation

`find(x)` is the heart of Union-Find. It tells us which group x belongs to by finding its top-most boss — the **root** of the tree.

Here's the naive version: keep going up until `boss[x] == x`.

We then improve it with **path compression**: while searching for the root, we **flatten** the tree by making each node point directly to the root. This technique flattens the structure by making every node along the path point directly to the root, reducing the depth of the tree.

In terms of the company analogy, it's like restructuring the hierarchy so that all employees along the chain now report directly to the top-level boss, making future lookups more efficient.

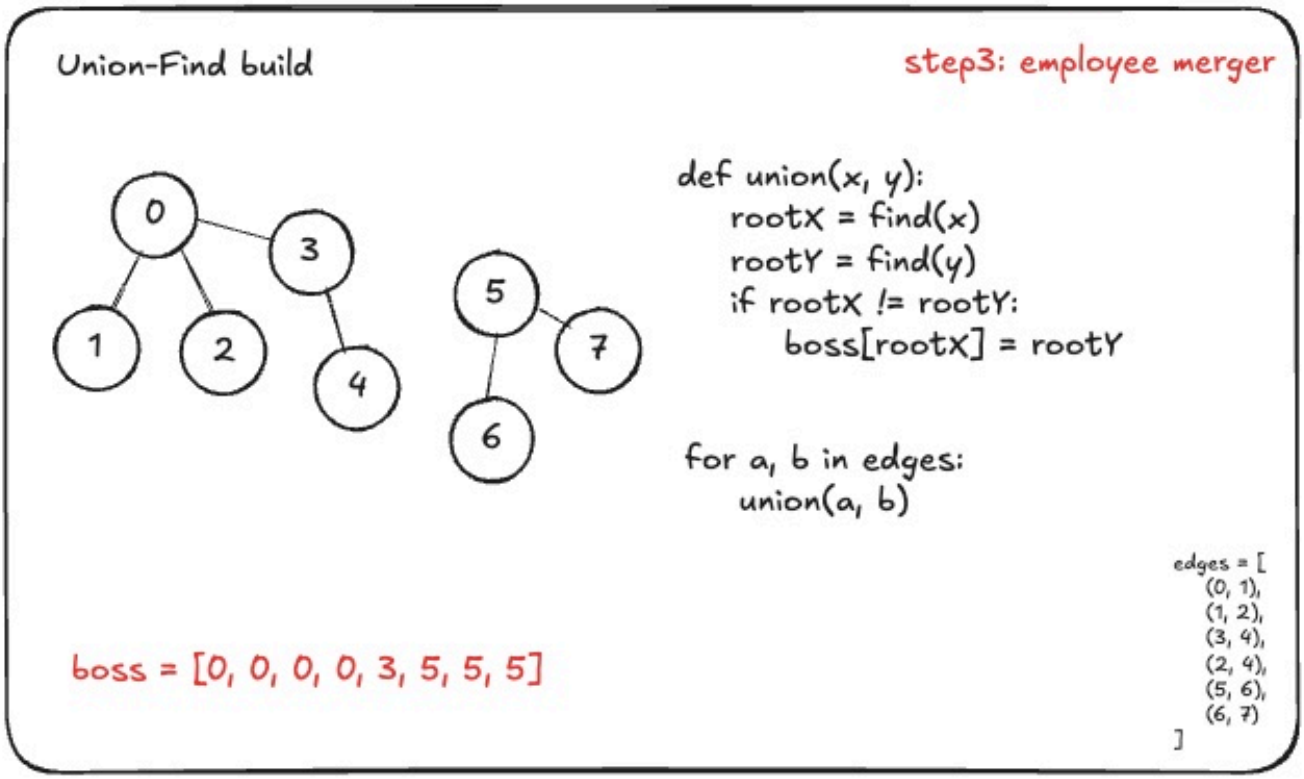


Step 3: The Union Operation

Now we want to merge groups — for example, when we find a connection between node `x` and node `y`.

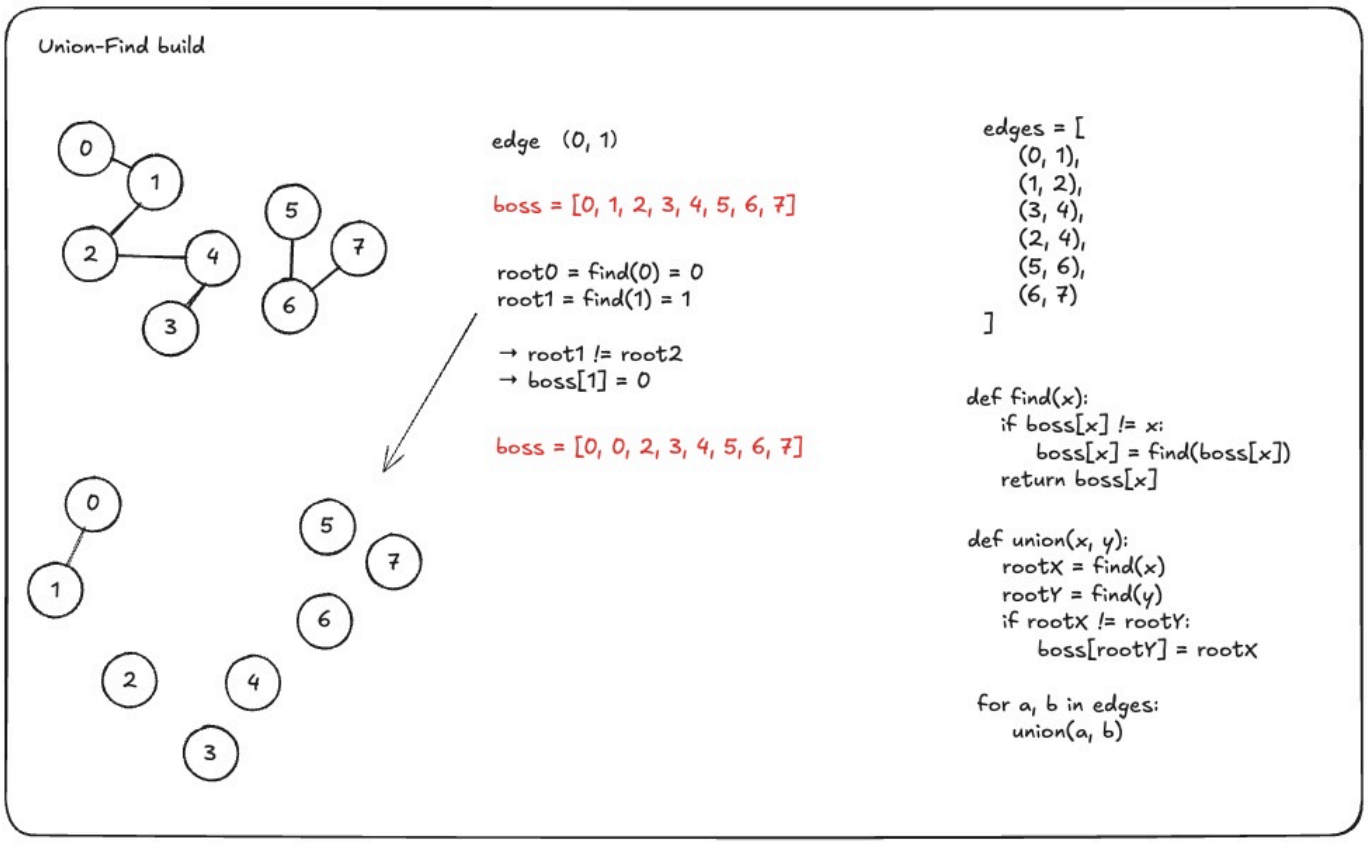
We first find the **root** of each node using `find(x)` and `find(y)`. If they have different roots, we merge them by making one boss point to the other.

This is like a company merging two departments under one boss. You loop through all edges in the graph and perform a union on the connected nodes.



How to build Union-Find?

Let’s walk through a concrete example.



Step 1: No Connections Yet

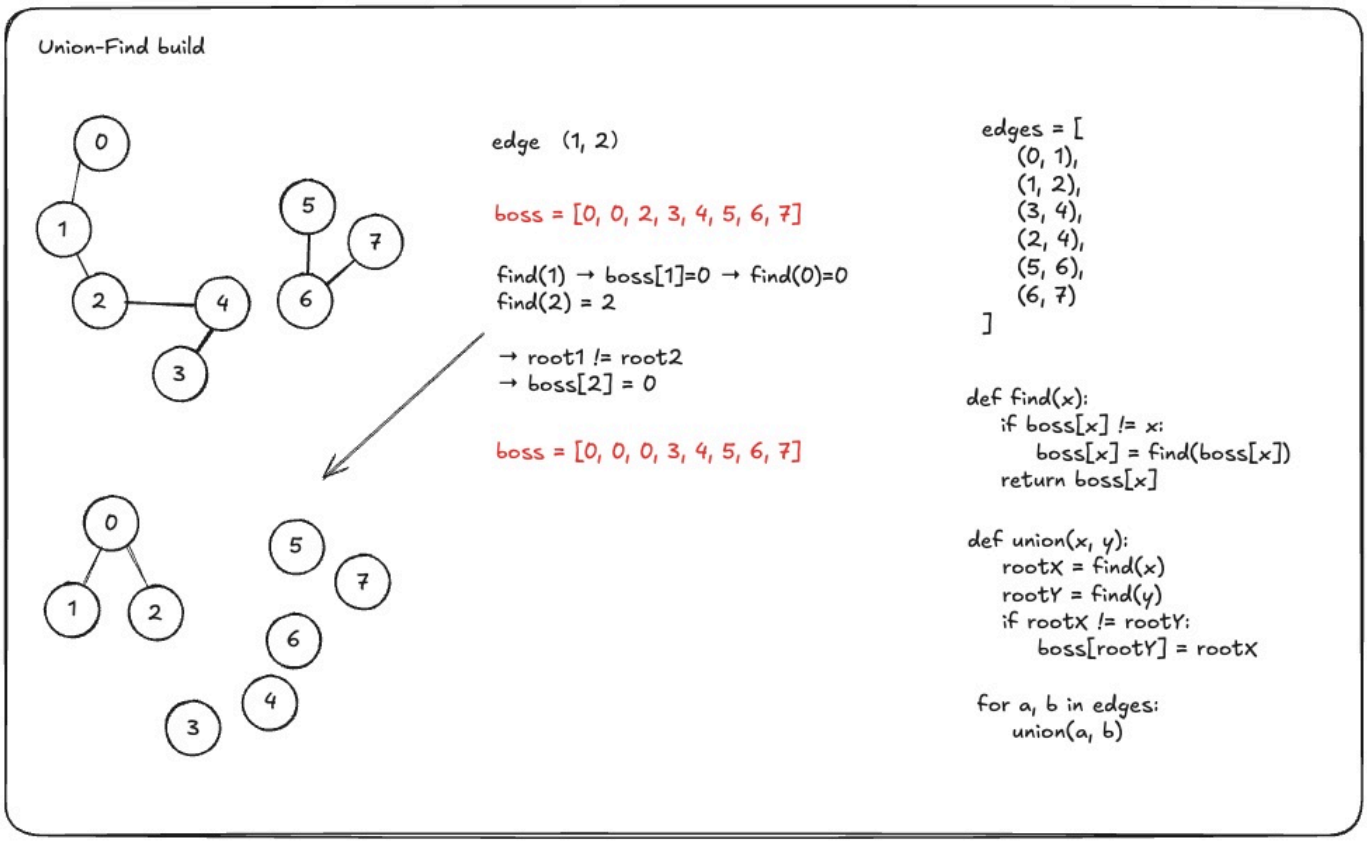
Initially, all nodes are their own bosses: Each node is its own little company. `boss[i] = i`。

Step 2: Connect (0, 1)

`find(0) = 0`, `find(1) = 1` → different roots

Merge: make 1 report to 0 → `boss[1] = 0`

We got a new array.

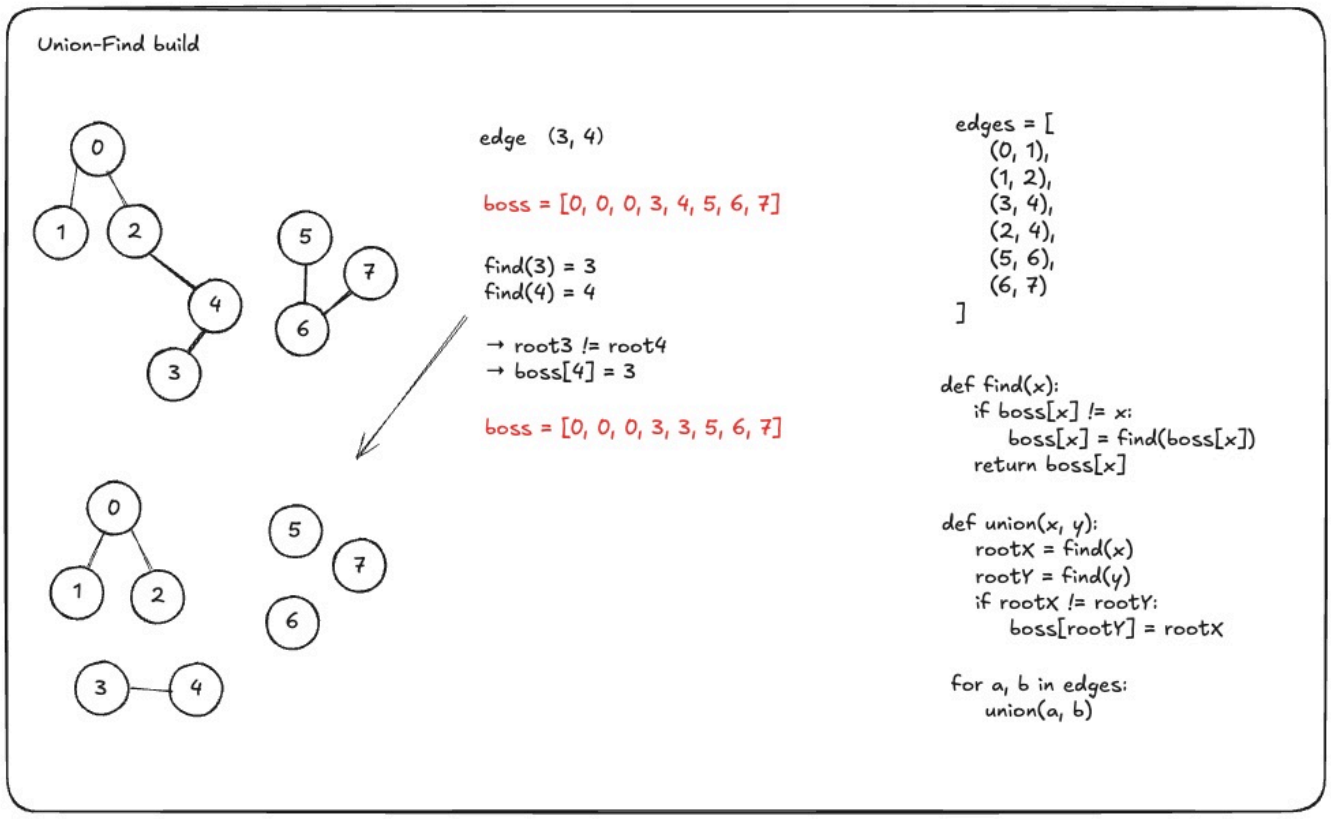


Step 3: Connect (1, 2)

- `find(1) → boss[1] = 0`, and `find(0) = 0` → root is 0
- `find(2) = 2`

Merge: `boss[2] = 0`

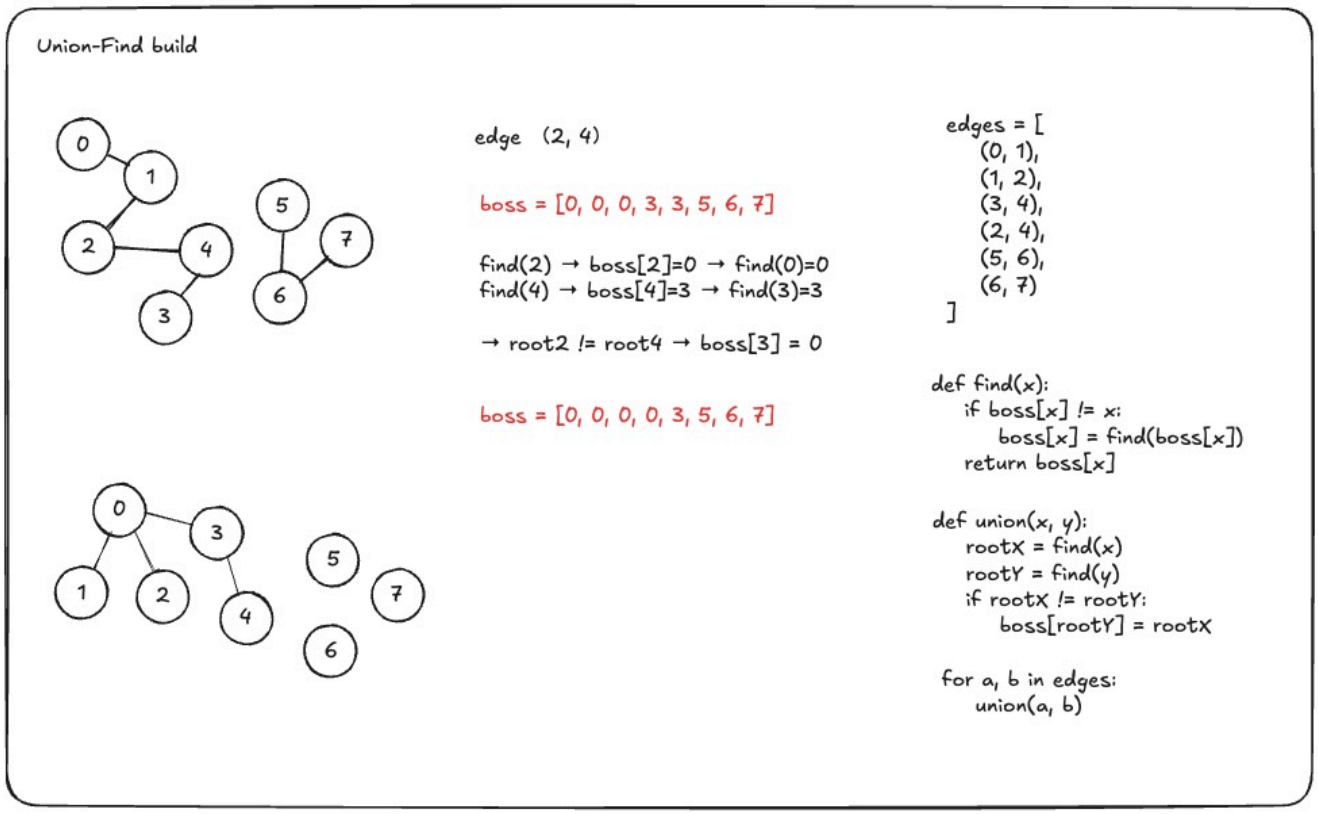
Now we've merged 0, 1, and 2 into one group: `boss = [0, 0, 0, 3, 4, 5, 6, 7]`.



Step 4: Connect (3, 4)

- `find(3) = 3`
- `find(4) = 4`

Merge: `boss[4] = 3`

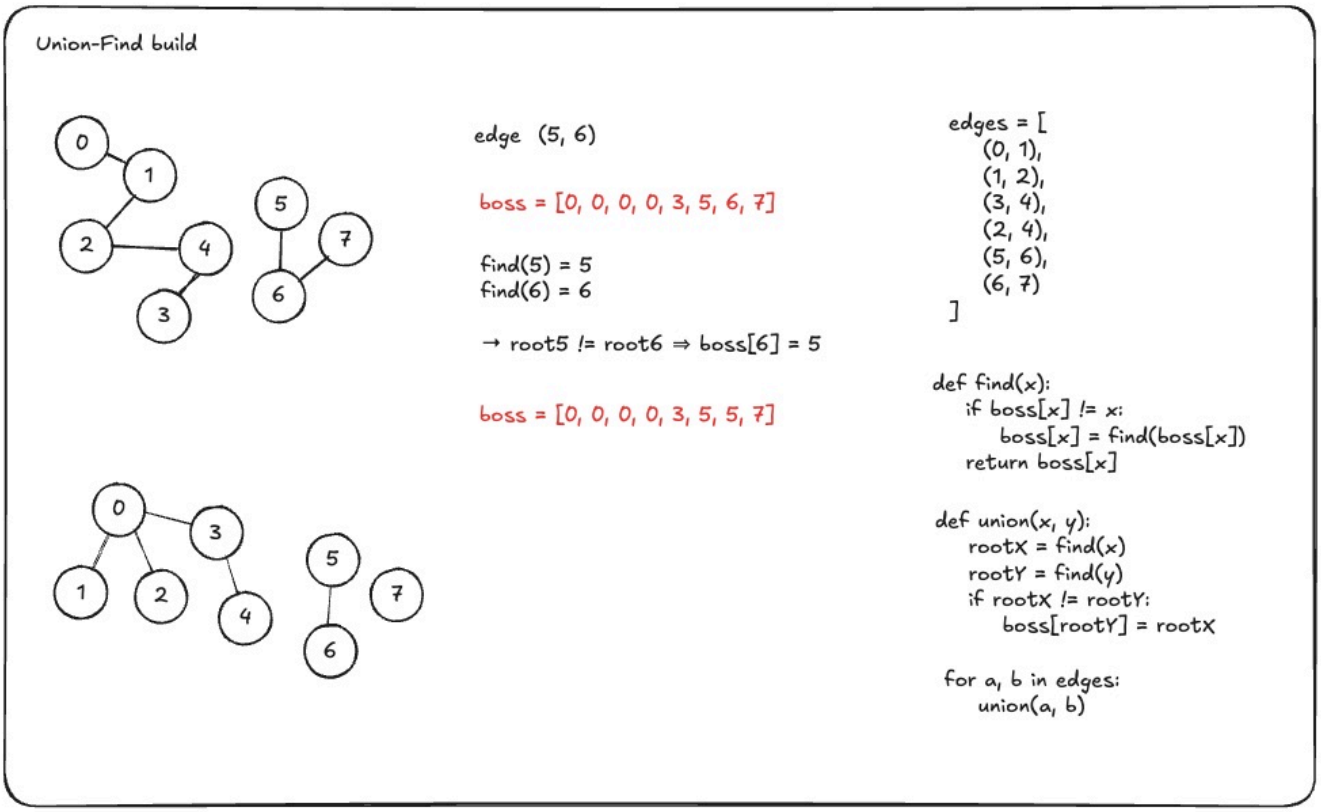


Step 5: Connect (2, 4)

- `find(2)` → 0
- `find(4) → boss[4]=3 → find(3) = 3 → root is 3`

Merge: `boss[3] = 0` → now 3 and 4 are under 0 too

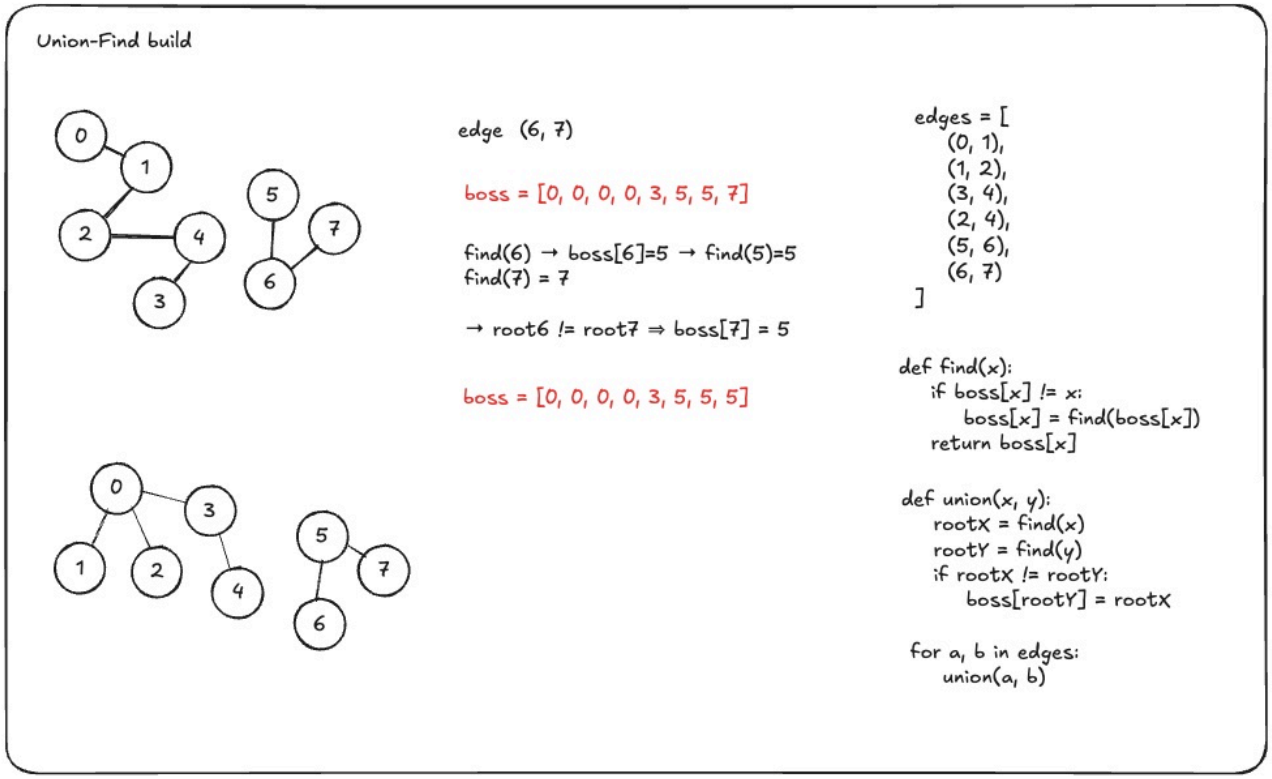
This subgraph already has 5 members connected together.



Step 6: Connect (5, 6)

- `find(5) = 5`
- `find(6) = 6`

Merge: `boss[6] = 5` → now 5 and 6 are under 5



Step 7: Connect (6, 7)

- `find(6)` → boss[6]=5 → the root 5
- `find(7) = 7`

Merge → `boss[7] = 5` → now 5, 6 and 7 are in the same group

Final structure: `boss = [0, 0, 0, 0, 3, 5, 5, 5]`

We now have two major connected components:

- Group represented by 0: includes nodes 0, 1, 2, 3, 4
- Group represented by 5: includes nodes 5, 6, 7

This is a basic understanding of the key steps and process of union find. When solving problems, the most efficient way is to remember the construction process of this data structure and use it in the problem to achieve our goal.

When to use it?

Besides, a big part of mastering Union-Find is knowing **when to apply it**.

Unlike sliding window, binary search, or DFS, Union-Find problems often feel less obvious at first glance.

So here's what to watch for:

So here's what to watch for:

- Does the problem talk about **groups**, **regions**, or **connectivity**?
- Are we checking whether two things are in the same group?
- Are we merging groups or building connections dynamically?
- Are there a lot of queries like "are these connected?"
- Do the keywords include: **undirected edges**, **components**, **connectivity**?

If yes, Union-Find might be a great fit.

Practice

Next, let's take a look at how to identify and use union-find through an exercise.

```
The current password is represented by the string `currentPassword`, consisting only of
lowercase Latin letters.

New password requirements have just been released.
The new password, called `newPassword`, must meet two specific requirements:

- It must be a palindrome.
- It must have a period of `k`. That is, `newPassword[i] = newPassword[i+k]` for all `1 ≤ i ≤
length(newPassword) − k`.

The objective is to determine the minimum number of characters that need to be changed in
`currentPassword` to create a `newPassword` of the same length.

---

**Example**

`currentPassword = "abzzbz"`

`k = 3`

Changing the first character of `currentPassword` to 'z' creates a `newPassword` of
`"zbzzbz"` which is a palindrome with a period of 3.

Therefore, the answer is 1.

---

**Function Description**


Complete the function `findMinChanges` in the editor with the following parameters:
```

```python
string currentPassword: the current password
int k: the required period of the new password
```

Let's use 3 minutes to read it.

Key Information Abstraction

We are given a string and asked to modify it so that:

1. It becomes a **palindrome**: `s[i] == s[n - 1 - i]`
2. It satisfies **periodicity with period k**: `s[i] == s[i + k]`

We need to compute the **minimum number of character changes** required to transform `currentPassword` into a `newPassword` of the same length.

Intuition

At first glance, a natural idea would be:

“For each position in the string, check whether it matches its corresponding palindrome / periodic position. If not, change it.”

- For palindrome: `s[i] == s[n - 1 - i]`
- For periodicity: `s[i] == s[i + k]`

This sounds reasonable and can be implemented with two pointers or direct traversal.

But here’s the problem:

Some positions may be involved in **multiple constraints** — for example, a position might be both part of a palindrome and a periodic pair. This creates **interdependency**: we can't treat each constraint separately, we need to consider the **entire group of linked positions**. We must make a **global decision** across the group.

Since the goal is to minimize the number of changes, we should shift our thinking: It’s not about checking one-to-one equalities anymore.

It’s about **grouping all positions that must be equal**, then picking the **most common character** in each group, and changing the rest to it.

This is exactly the things Union-Find is designed for.

So I got my thinking of this problem:

So I got my thinking of this problem:

- Union all positions that must be equal into the same set.
- For each set, count the **most frequent character**.
- Change the rest of the characters to this one to **minimize the number of changes**.

```
'''
Thinking:
- Union all positions that are required to be equal into the same group.
- For each group, count the most frequent character among those positions.
- Change all other characters in the group to that character — the total number of changes
will be minimized.
'''
```

Write Code

```
from collections import defaultdict, Counter

def findMinChanges(currentPassword: str, k: int) -> int:
    n = len(currentPassword)

    # build union find functions
    ## Initilization
    ## each position is its own parent (n individual sets)
    parent = list(range(n))

    ## build find function
    def find(x):
        '''
        Find the root of x.
        Apply path compression to flatten the tree.
        '''
        while parent[x] != x:
            parent[x] = parent[parent[x]]
            x = parent[x]
        return x

    # build
    def union(x, y):
        '''
        Merge the set of x into the set of y
        by attaching the root of x to the root of y
        '''
        parent[find(x)] = find(y)
```

```
    # Apply constraints
    ## period constraints
    ## s[i] == s[i + k]
    for i in range(n - k):
        union(i, i + k)

    ## palindrome constraints
    ## s[i] == s[n - 1 - i]
    for i in range(n // 2):
        union(i, n - 1 - i)

    # Get all groups with the same representative together: the same group means that their characters must be the same
    groups = defaultdict(list)
    for i in range(n):
        root = find(i)
        groups[root].append(i)

    # Count minimum number of changes
    res = 0
    for group in groups.values():
        chars = [currentPassword[i] for i in group]
        freq = Counter(chars)
        # Find most common character in the group
        max_freq = max(freq.values())
        # Change the rest to match it
        res += len(group) - max_freq

    return res
```

This problem can be tricky for beginners mainly because of **when** to apply Union-Find.
Once you realize that some positions are **interdependent** — i.e., they must be the same due to period constraint or palindrome constraint — you can model the problem as a **set of linked positions**.

Union-Find is simply a data structure to **merge** and **track** those linked groups efficiently.
As long as you remember how to build it, you can plug it in as a reliable tool whenever the problem requires grouping or connected components.