Solution: Redundant Connection

Let's solve the Redundant Connection problem using the Union Find pattern.

We'll cover the following

Statement

We're given an undirected graph consisting of n nodes. The graph is represented as an array called `edges`, of length n, where `edges[i] = [a, b]` indicates that there is an edge between nodes `a` and `b` in the graph.

Return an edge that can be removed to make the graph a tree of n nodes. If there are multiple candidates for removal, return the edge that occurs last in `edges`.

Constraints:

- $3 \leq n \leq 1000$
- `edges.length`$= n$
- `edges[i].length` $= 2$
- $1 \leq$ `a` $<$ `b` $\leq n$
- $a \neq b$
- There are no repeated edges.
- The given graph is connected.
- The graph contains only one cycle.

Solution

So far, you have probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive approach

We can solve this problem using techniques like DFS or BFS, however, the naive approach using DFS or BFS has a time complexity of $O(n^2)$ in the worst-case scenario. This is because the DFS or BFS algorithm will explore the graph by visiting each node and its edges, which may result in visiting many nodes multiple times and not necessarily finding the redundant connection in an efficient manner.

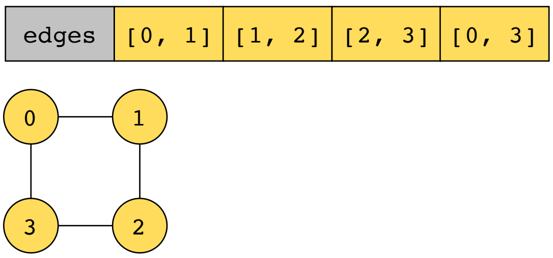
Optimized approach using union find

We are going to solve this problem with the help of the union find pattern and will be using union by rank and path compression.

- **Union by rank:** We connect the nodes that come afterward to the nodes that came before them. This allows us to add new nodes to the subset of the representative node of the larger connected component. Using ranks to connect nodes in this manner helps reduce the depth of the recursion call stack of the `find` function, since the trees within each disjoint set remain relatively shallow.
- **Path compression:** On each `find` operation on a node of a tree, we update the parent of that node to point directly to the root. This reduces the length of the path of that node to the root, ensuring we don't have to travel all the intermediate nodes on future `find` operations.

The slides below illustrate how the algorithm runs:

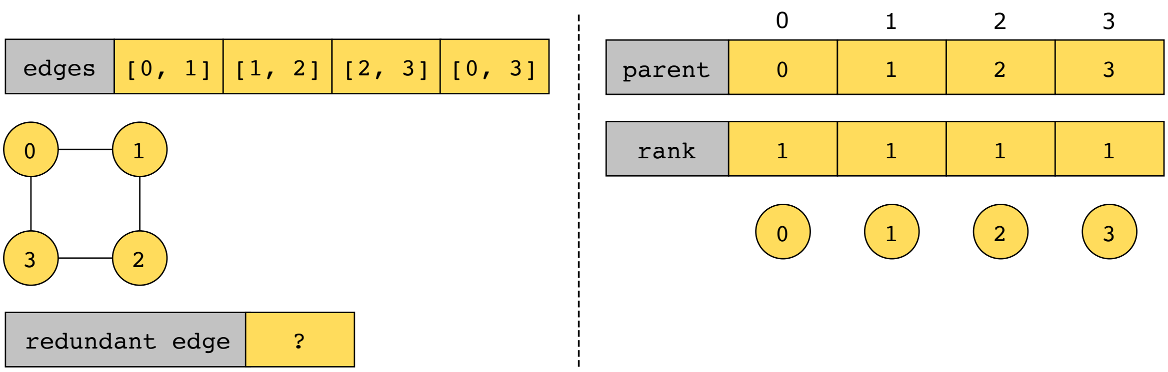
We will first make union find data structures from the **edges** array.



1 of 10

This creates **parent** and **rank** arrays with length equal to the number of nodes in the graph, i.e. 4.

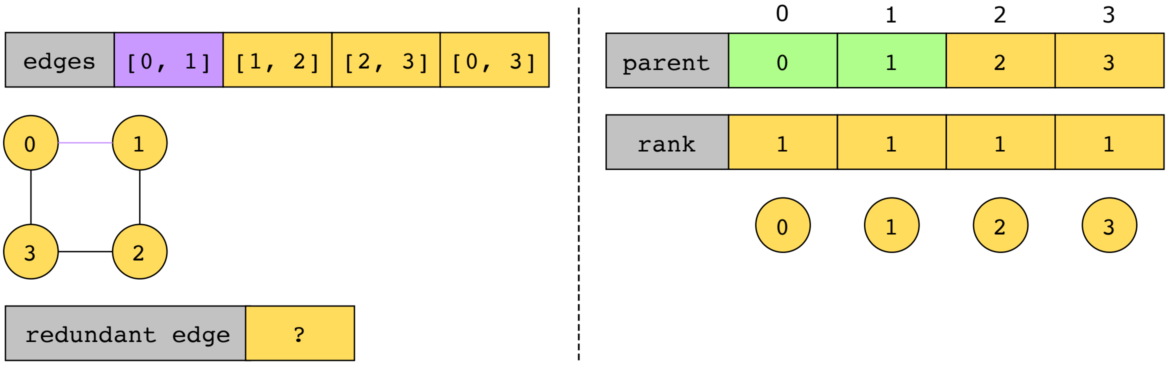
Next, we start traversing the **edges** array to identify the redundant edge.



2 of 10

edge: **[0, 1]**

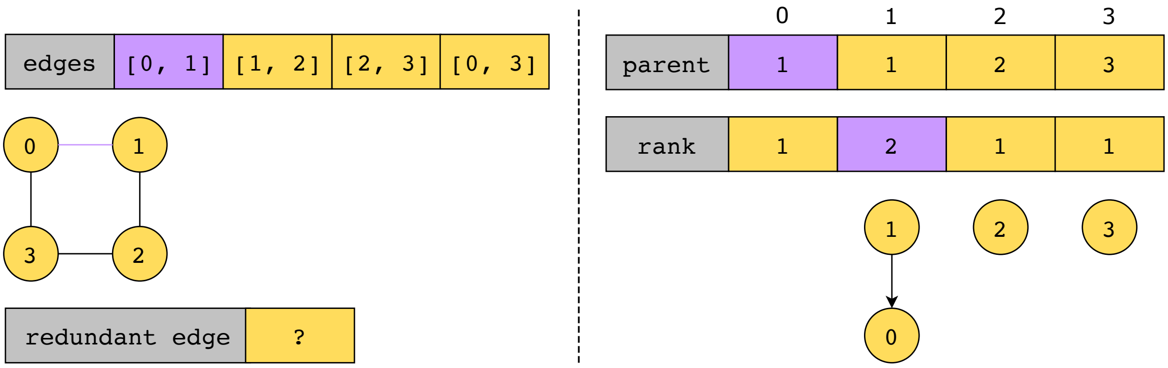
For the above edge, we check if nodes **0** and **1** have the different parents.



3 of 10

edge: **[0, 1]**

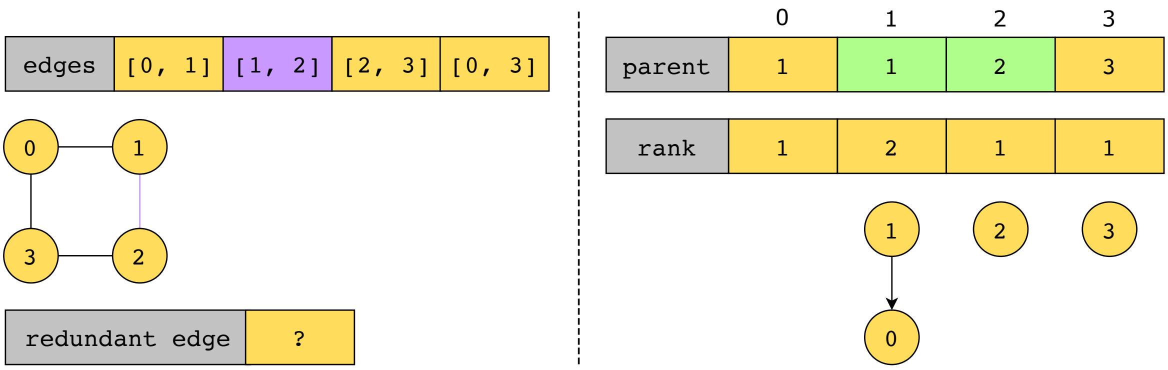
Since they do, the edge is not redundant, so we connect both nodes through the **union(0, 1)** method.



4 of 10

edge: **[1, 2]**

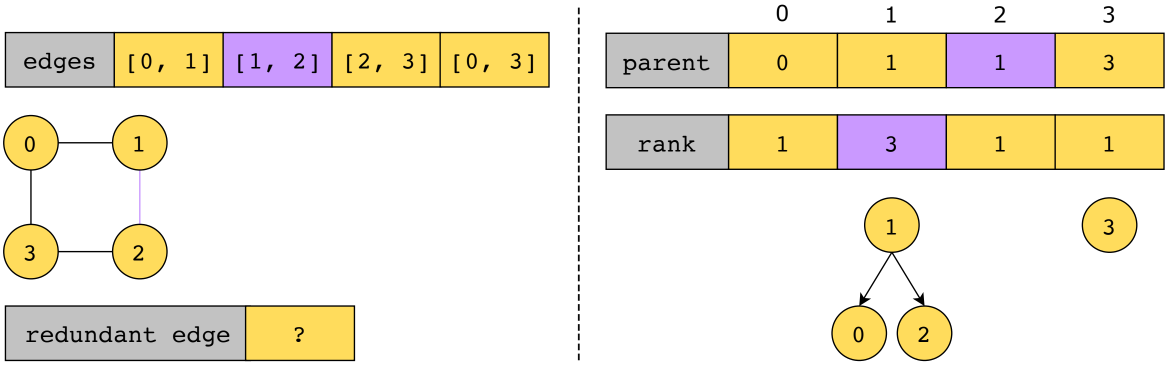
For the above edge, we check if nodes **1** and **2** have different parents.



5 of 10

edge: **[1, 2]**

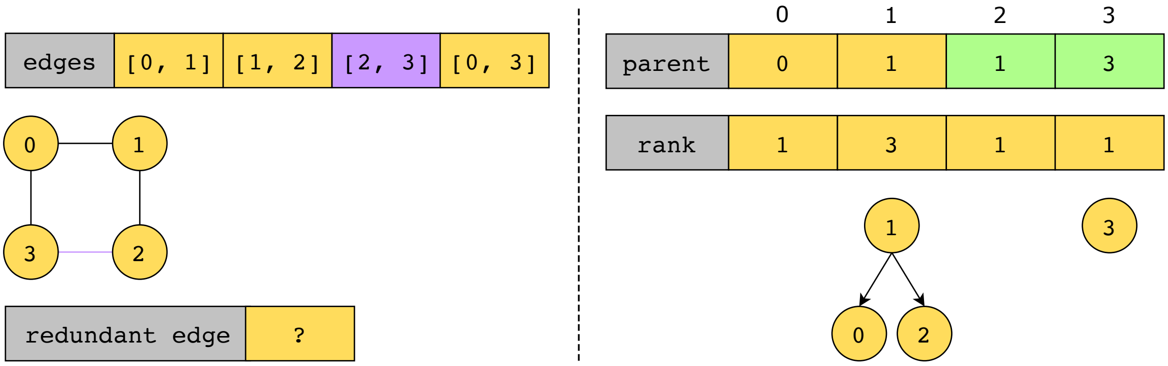
Since they do, the edge is not redundant, so we connect both nodes through the **union(1, 2)** method.



6 of 10

edge: **[2, 3]**

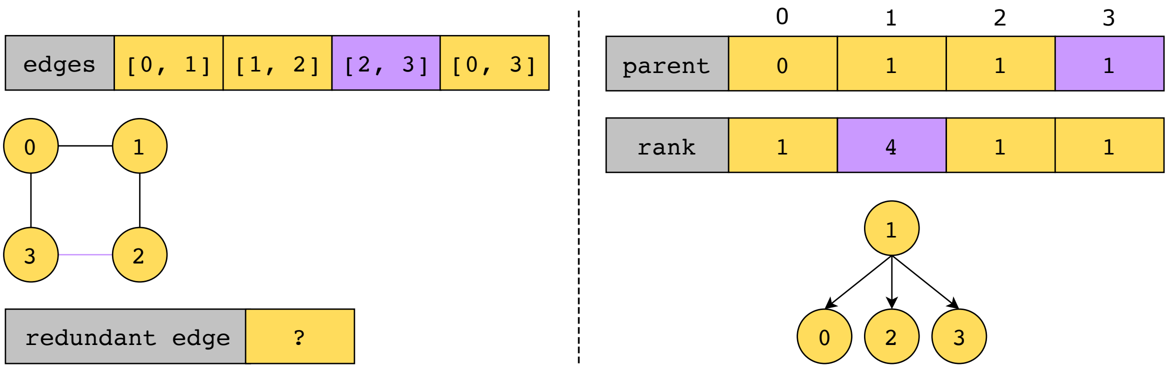
For the above edge, we check if nodes **2** and **3** have different parents.



7 of 10

edge: **[2, 3]**

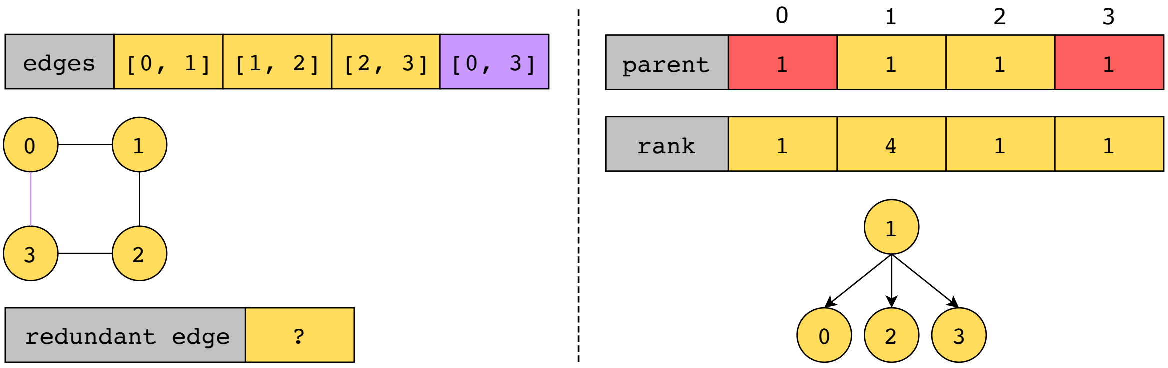
Since they do, the edge is not redundant, so we connect both nodes through the **union(2, 3)** method.



8 of 10

edge: **[0, 3]**

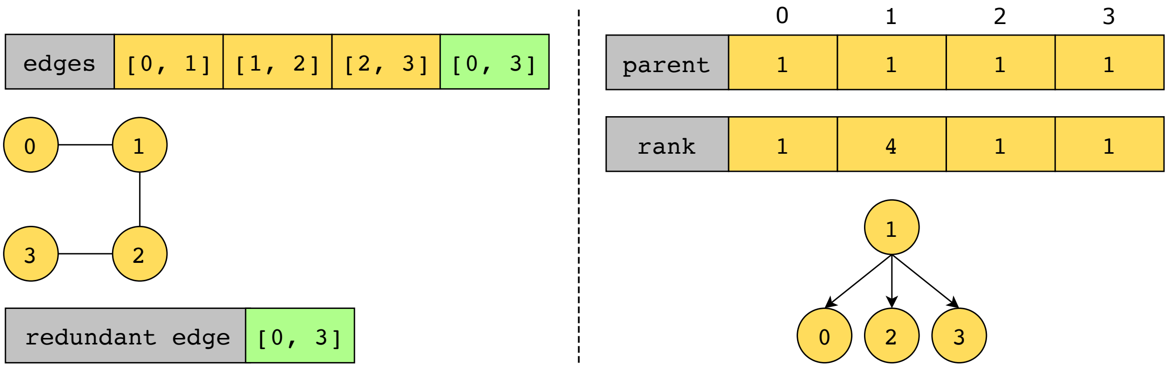
For the above edge, we check if nodes **0** and **3** have different parents.



9 of 10

edge: **[0, 3]**

Since they don't, the edge is already part of a connected component in the Union Find data structures and can therefore be removed to convert the graph into a tree. So we return the redundant edge, **[0, 3]**.



10 of 10

> **Note**: In the following section, we will gradually build the solution. Alternatively, you can skip straight to just the code.

Step by step solution construction

We start with the basic implementation of Union Find (without the use of rank or path compression):

File: `UnionFind.java`

In the `UnionFind` class, we declare the `parent` array with length based on the `edges` array.

File: `main.java`

In the `redundantConnection` function, we'll declare an object of the given class and initialize the `parent` array with default values.

Java

main.java

UnionFind.java

```
import java.util.*;

class UnionFind {

    public int[] parent;

    // Constructor
    public UnionFind(int n) {

        parent = new int[n + 1];
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
        }
    }

    // Function to find which subset a particular element belongs to
    public int find(int v) {
        if (parent[v] != v) {
            return find(parent[v]);
        }
        return v;
    }

    // Function to join two subsets into a single subset
    public void union(int v1, int v2) {

        int p1 = find(v1);
        int p2 = find(v2);
```

Redundant Connection

File: `UnionFind.java`

Next, we need to check if the vertices forming an edge have the same parent. In the `union` function, we'll call the parent of the first vertex `p1` and the parent of the second vertex `p2`. The algorithm is described as follows:

- If both `v1` and `v2` have the same parent, i.e., `p1` is equal to `p2`, the given edge is redundant, so we return FALSE.
- Otherwise, this edge is connecting two vertices that were not already connected. So, we'll update the `parent` list by making a connection based on the current edge and then return TRUE.

File: `main.java`

In the `redundantConnection` function, we traverse the `edges` array and for each edge, we attempt to connect its two vertices, `v1`, and `v2`, through the `union(v1, v2)` method.

Java

main.java

UnionFind.java

```
import java.util.*;

class UnionFind {
```

```
    public int[] parent;

    // Constructor
    public UnionFind(int n) {

        parent = new int[n + 1];
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
        }
    }

    // Function to find which subset a particular element belongs to
    // Returns FALSE if both vertices have the same parent, otherwise, update
    // Returns TRUE if no cycle exits in the graph
    public int find(int v) {
        if (parent[v] != v) {
            return find(parent[v]);
        }
        return v;
    }

    // Function to join two subsets into a single subset
    public boolean union(int v1, int v2) {
```

Redundant Connection

We will now add union by rank and path compression to the Union Find algorithm. We'll make the following changes to the `UnionFind` class:

Union by rank:

- We initialize a `rank` array (set to the length of the `edges` array) with 1s.
- In the `union` function, if the parents, `p1` and `p2` of the vertices are not the same, we make the connection based on the ranks of both parents. This is done in the following way:
  - If `p1`'s rank is greater than `p2`'s rank, we'll update `p2`'s parent with `p1`, and add `p2`'s rank to `p1`'s rank. For example, if `rank[p1]` $= 5$ and `rank[p2]` $= 2$, we'll add `p2` to `p1`'s set and update its rank to $5 + 2 = 7$.
  - Similarly, if `p2`'s rank is greater than `p1`'s rank, we'll update `p1`'s parent with `p2`, and add `p1`'s rank to `p2`'s rank.

Path compression:

- In the `find` function, for a node `v`, we make the found root as the parent of `v` so that we don't have to traverse all the intermediate nodes again on further `find` operations.

Java

main.java

UnionFind.java

```
import java.util.*;

class UnionFind {

    public int[] parent;
    public int[] rank;

    // Constructor
    public UnionFind(int n) {
```

```java

        parent = new int[n + 1];
        rank = new int[n + 1];
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
            rank[i] = 1;
        }
    }

    // Function to find which subset a particular element belongs to
    // Returns FALSE if both vertices have the same parent, otherwise, update
    // Returns TRUE if no cycle exits in the graph
    public int find(int v) {
        if (parent[v] != v) {
            parent[v] = find(parent[v]);
        }
        return parent[v];
    }
```

Redundant Connection

Lastly, we need to return the redundant edge. While traversing the `edges` array, for each edge, we will check if the `union(v1, v2)` method returns FALSE:

- If it does, both `v1` and `v2` will have the same parent. So the current edge will be redundant, and we return it.
- Otherwise, `v1` and `v2` have different parents, so we connect them.

Java

main.java

UnionFind.java

```java
import java.util.*;

class RedundantConnections {

    public static int[] redundantConnection(int[][] edges) {

        // Declaring the parent and rank list with lengths based on the edges
        System.out.println("\n\tDeclaring the parent and rank arrays");
        UnionFind connections = new UnionFind(edges.length);
        System.out.println("\t\tparent: " + Arrays.toString(connections.parer
        System.out.println("\t\trank: " + Arrays.toString(connections.rank));

        // Traversing the edges of the graph to check for the redundant edge
        for (int[] edge : edges) {
            int v1 = edge[0];
            int v2 = edge[1];
            if (!connections.union(v1, v2)) {
                return edge;
            }
        }
        return new int[]{};
    }

    // Driver code
    public static void main(String[] args) {
        int[][][] edges = {
                {{1, 2}, {1, 3}, {2, 3}},
```

Redundant Connection

Just the code

Here's the complete solution to this problem:

Java

main.java

UnionFind.java

```java
class RedundantConnections {

    public static int[] redundantConnection(int[][] edges) {

        UnionFind connections = new UnionFind(edges.length);

        for (int[] edge : edges) {
            int v1 = edge[0];
            int v2 = edge[1];
            if (!connections.union(v1, v2)) {
                return edge;
            }
        }
        return new int[]{};
    }

    // Driver code
    public static void main(String[] args) {
        int[][][] edges = {
                {{1, 2}, {1, 3}, {2, 3}},
                {{1, 2}, {2, 3}, {1, 3}},
                {{1, 2}, {2, 3}, {3, 4}, {1, 4}, {1, 5}},
                {{1, 2}, {1, 3}, {1, 4}, {3, 4}, {2, 4}},
                {{1, 2}, {1, 3}, {1, 4}, {1, 5}, {2, 3}, {2, 4}, {2, 5}}
        };
```

Redundant Connection

Solution summary

To recap, the solution to this problem can be divided into the following two main parts:

- Initialize `parent` and `rank` arrays based on the length of the `edges` array.
- Traverse the `edges` array and for each edge, compare the parents of both vertices:
    - If the parents are the same, the current edge is redundant, so we return it.
    - Otherwise, we connect the two vertices based on their respective ranks.

Time complexity

The time complexity of this solution is $O(n)$, where n is the number of edges.

Explanation:

- We use a loop to traverse the `edges` array, resulting in $O(n)$ iterations.