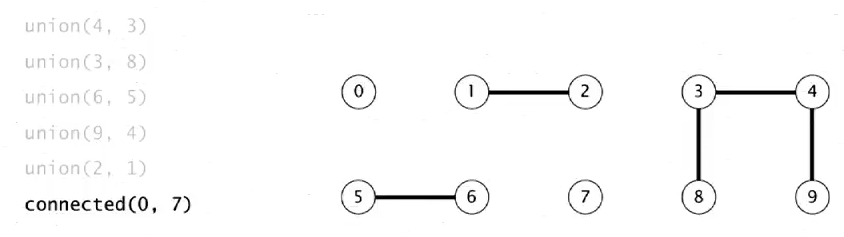
Dynamic Connectivity

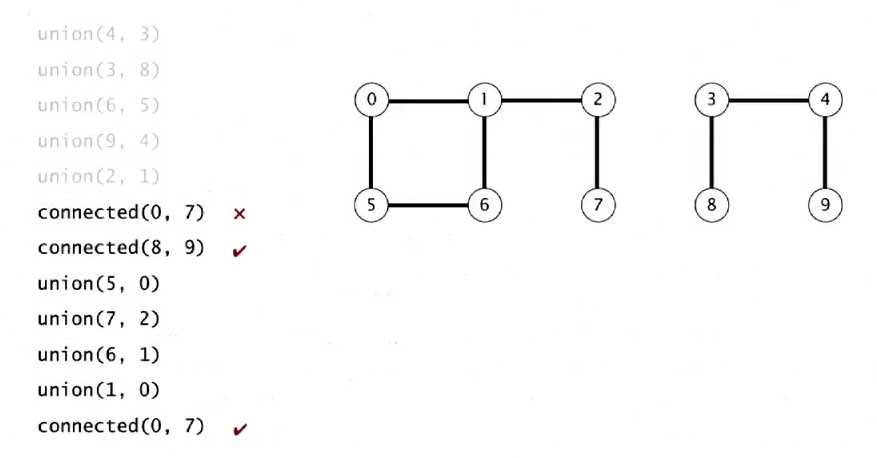
We will look at 2 classic algorithms here, quick find and quick union. **This doesn’t find actually the path between the two objects; it just answers if there is a path or not.**

* Given a Set of N objects:
  + **Union Command**: Connect 2 object.
  + **Find/connected query**: is there a path connecting two objects.

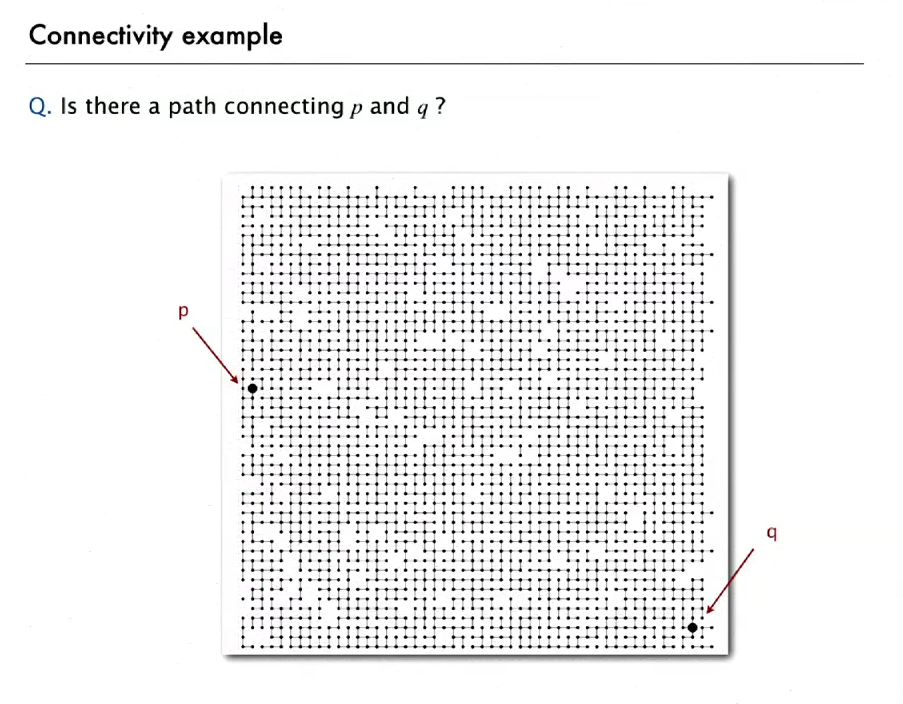
**Example:**



It will return false as 0 and 7 isn’t connected.

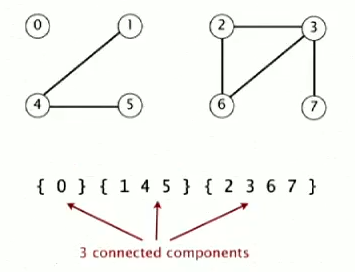


But in this case, it will return true, since both are connected

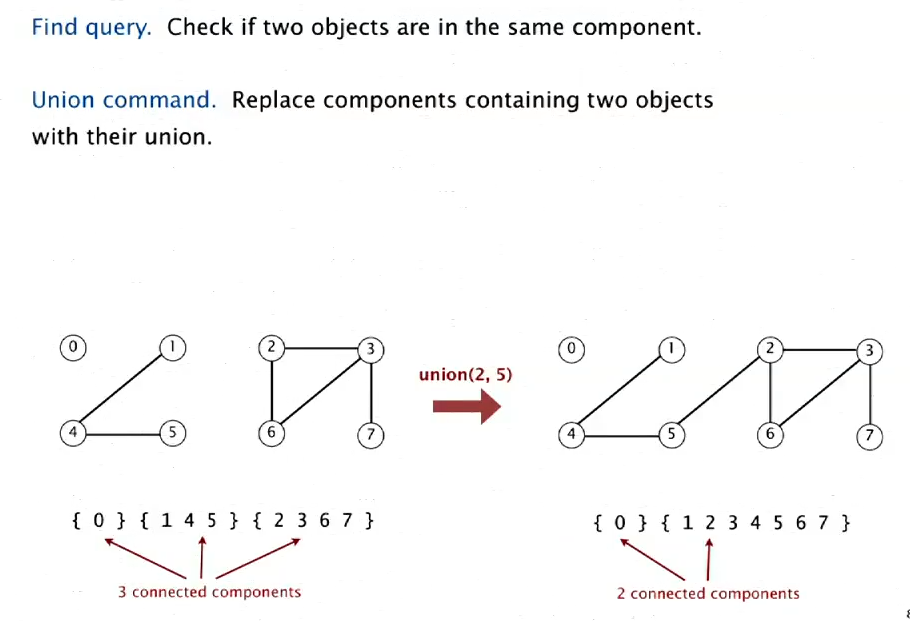


Practical Example

* **Connected Components:** Maximal set of objects that are mutually connected.

****

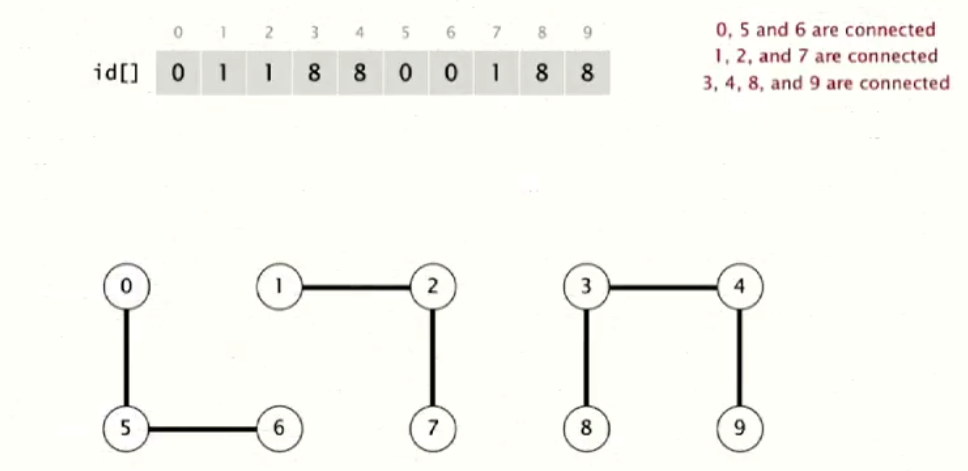
* **Implementing the operations**

****

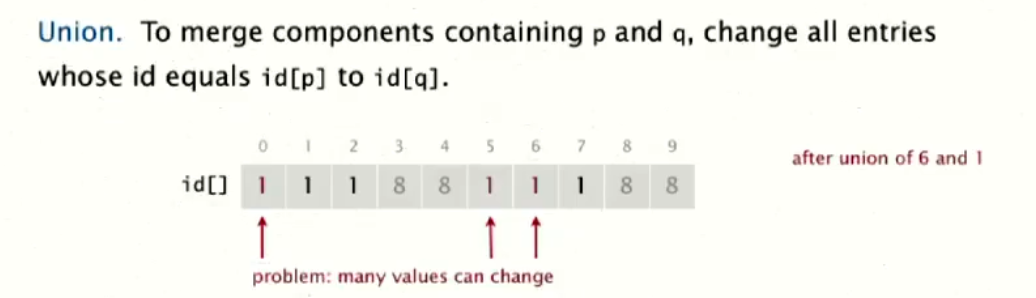
**Quick Find**

To solve the dynamic connectivity problem, the first solution we will look into is quick find. This is known as eager approach.

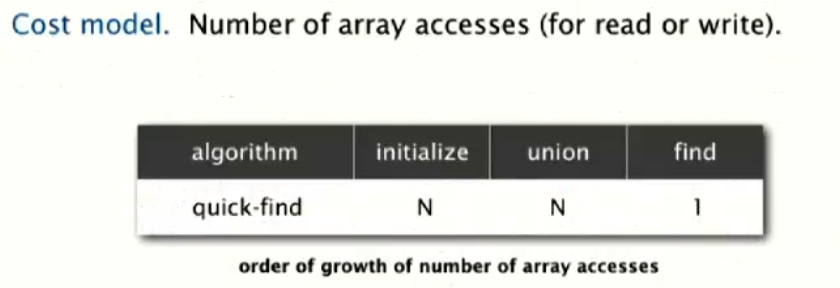
* Here we will use an integer array, the index number will indicate the objects and the entries inside the array will indicate if they are connected or not.



* For union (p. q) we need to change all the entries which id is equal to id[p] to id[q]. (For example, check web). Basically, change the **{first component’s (p) value + all its matching component value}** to **2nd component’s(q) value**.

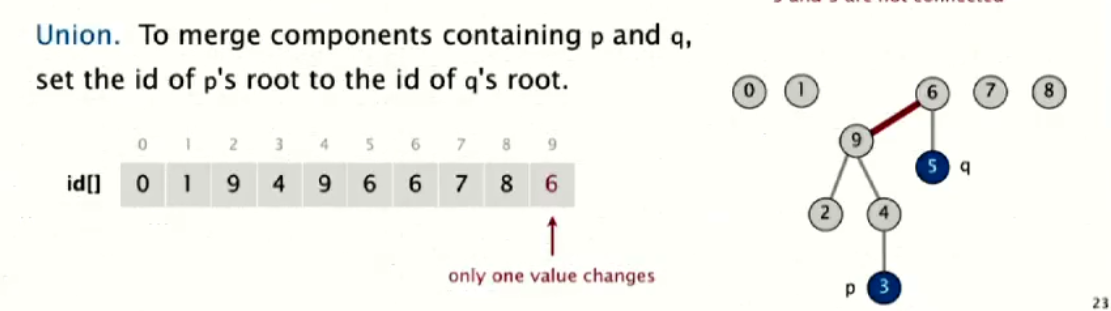


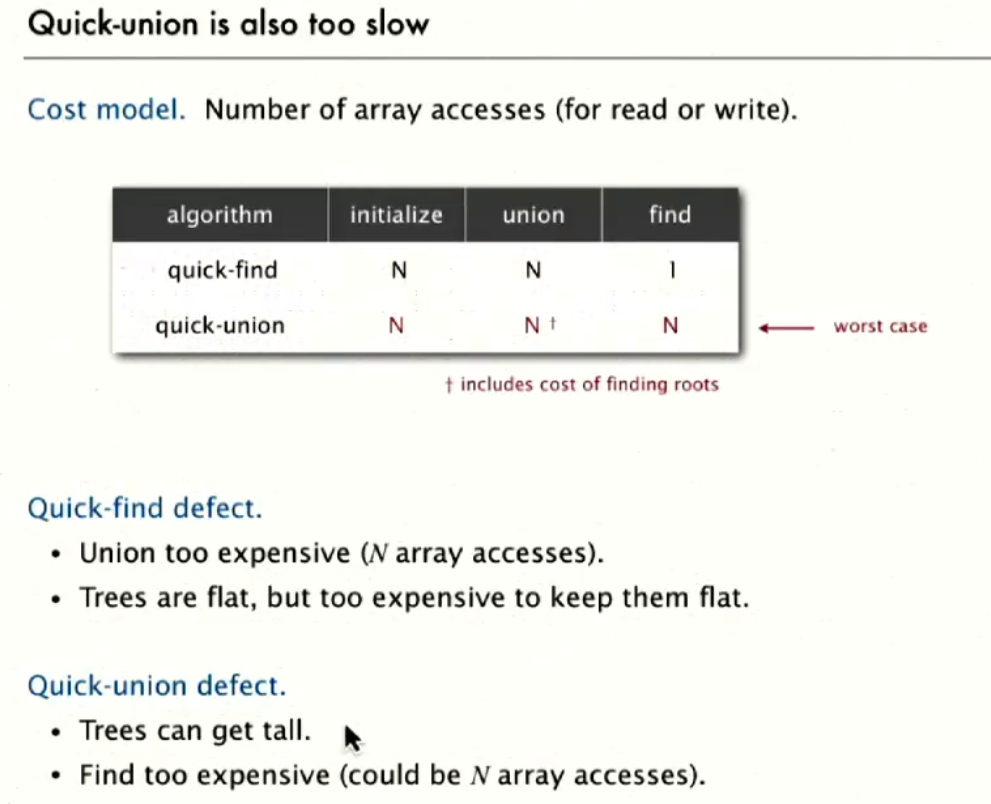
But here lies the problem of the quick find, to connect two component we have to do expensive operations.

****

**Quick Union**

Because union operation is too slow in quick find, Quick union provide a better solution to it. It is known as lazy approach, where we don’t work where don’t have to.

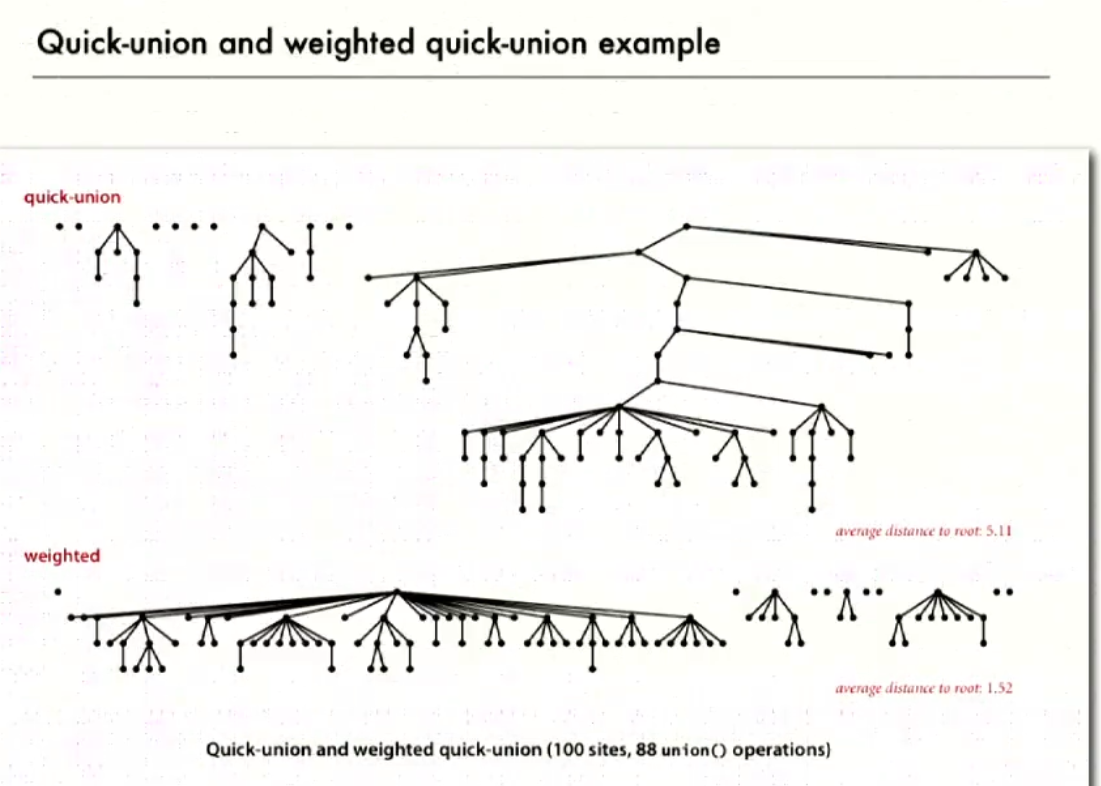
* The data structure here is an array with N size. The index denotes the object and the value inside denotes the root of the object. 
* To find connectivity is checking if p and q have same root or not.
* To connect two component p and q, we have to change the root of p to the root of q. 



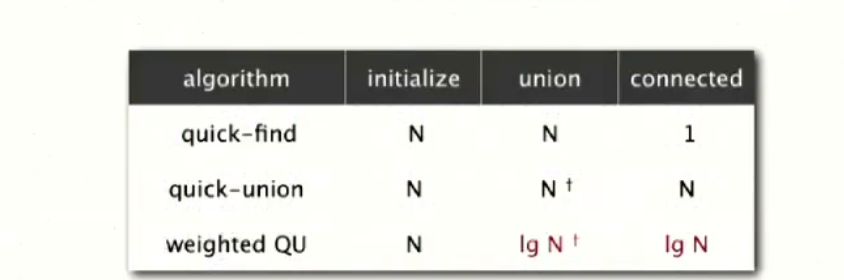
**Quick Union Improvement**

**Improvement 1:**

The problem with quick union is that tree can grow tall, and in that case to do find operation we may have to traverse the entire array. To make an improvement we can use **Weighted quick union.** Here the union operation always makes the smaller tree a child of the bigger tree, regardless of what order we pass them as the argument of the function.



* This uses the same data structure as quick union, but keep an extra array to keep the size of the tree.
* Find is identical.
* There is small modification in the union operation. If the size of the tree of p is bigger, then we assign the root of q to root of p.



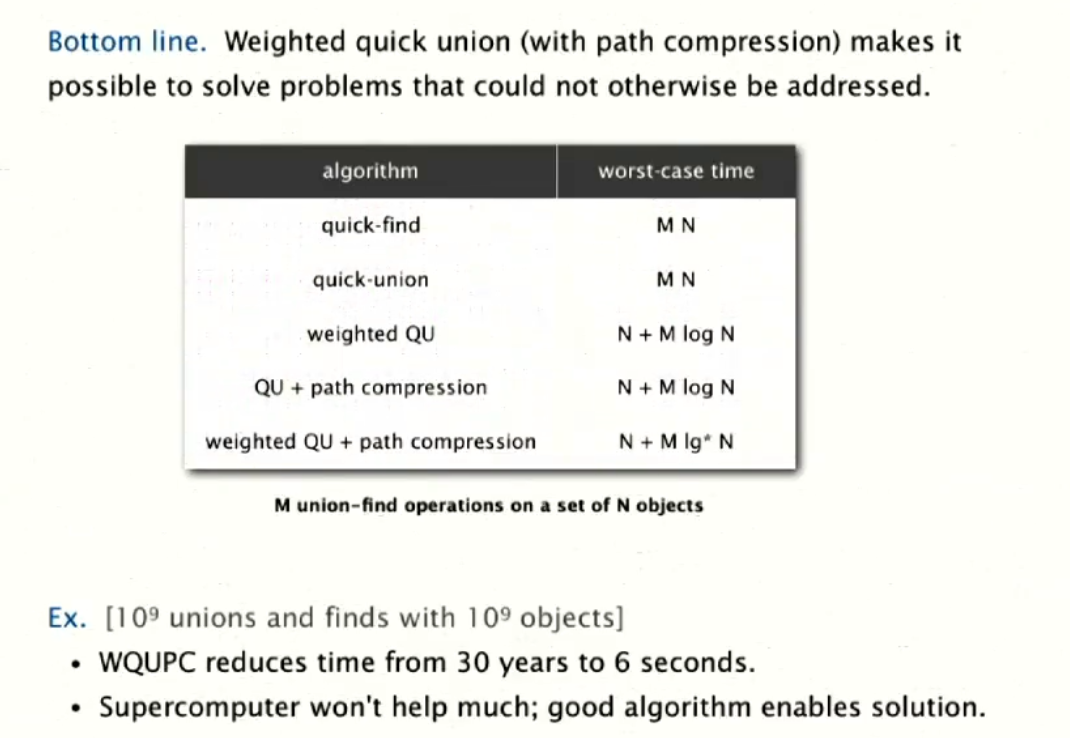
As we can see, the weighted quick union is logN in the operation for union and connect, and that is acceptable algorithm.

**Improvement 2:**

This is as close to the best solution we have for dynamic connectivity problem. It is an improvement over Weighted Quick Unionusing path compression. It is known as **Weighted Quick Union Path Compression.**

The theory behind this improvement is, while we find the root of a given node, we are touching all the nodes from that node to the root. So, we can also do one pass more and point each node to its grandparent. This way we can make the tree almost flat.

This is the best way to solve the dynamic connectivity problem and this is what should be used as it gives almost linear time like performance.

****