**Question 1**

**Social network connectivity.** Given a social network containing *N* members and a log file containing *M* timestamps at which times pairs of members formed friendships, design an algorithm to determine the earliest time at which all members are connected (i.e., every member is a friend of a friend of a friend ... of a friend). Assume that the log file is sorted by timestamp and that friendship is an equivalence relation. The running time of your algorithm should be *M*log*N* or better and use extra space proportional to *N*.

# [Social Network Connectivity](http://jznest.wordpress.com/2014/02/15/algorithms-part-i-social-network-connectivity-with-union-find/)



So we move on to a better version of Union-Find, the Weighted Union-Find algorithm. It uses a tree structure to represent the [connected components](http://en.wikipedia.org/wiki/Connected_space). So if all members are connected, there should be only one root element left, which means there is only one array element at index i such that array[i] == i. OK, so we go and check how many elements satisfy this condition after each union. However, this takes O(N) time for each check too.

Well, all we care is the number of roots left, right? Remember initially there are N objects and each represents its own tree. So we have N roots at the beginning. Each time we union, we get one less root. So, why not use a variable count to track the number of roots and change its value along the way of doing the union operations? And to check if all members are connected, we just check if count == 1? This takes constant time and O(MlogN + M) = O(M(logN + 1)) = O(MlogN).

<http://www.johncorser.com/princeton-algorithms-part-1-week-1/>

So here’s how I’d answer this one: The question really provokes  a union find algorithm (well, a weighted quick union anyway). What makes it a little bit different is you only care about finding out whether the number of root nodes is only 1, meaning that all users are in the same friendship tree.

What I would do is add a count variable to the data structure, set equal to the number of nodes/elements of the initial array. Then, I would decrement the count variable each time a union was formed. When the count variable becomes 1, that is how many root nodes remain in the forest, meaning everyone is part of the same friendship tree!

This would be accomplished with the algorithmic efficiency of weighted quick sort, which is already at the acceptable level of MlogN.

**Question 2**

**Union-find with specific canonical element.** Add a method find() to the union-find data type so that find(i) returns the largest element in the connected component containing i. The operations, union(), connected(), and find() should all take logarithmic time or better.

For example, if one of the connected components is {1,2,6,9}, then the find() method should return 9 for each of the four elements in the connected components.

<http://www.johncorser.com/princeton-algorithms-part-1-week-1/>

This question really has the same answer as the one above. The trick to accomplishing this in logarithmic or better time is to keep track of the state along the way. By creating a variable each tree that stores the value of the root of that tree, the find() operation could be completed in constant time.

<http://jznest.wordpress.com/2014/02/16/algorithm-part-i-union-find-with-specific-canonical-element/>

My thoughts and solutions:

The approach is similar to the one for the [previous post](http://jznest.wordpress.com/2014/02/15/algorithms-part-i-social-network-connectivity-with-union-find/). Let’s first look at some naive solutions.

1. If we implement the naive union-find as the base algorithm, the find() method would require O(N) time since we have to linearly scan all elements in the same component. So this is not acceptable.

2. So how about using the weighted quick union-find as the base? Where do we go next? Well, a connected component is represented by a tree. We could find the root of an object in O(logN) time and then use a breadth-first search to scan all objects in the tree to find the max. Unfortunately that’s not efficient enough.

3. As I mentioned, we use a similar approach–**that is to keep track of things along the way of the union operations.** Can we keep track of the max object in a given component and update it when doing union? In that way find() will only take constant time and the other methods will have the same time cost as before.

**Question 3**

**Successor with delete.** Given a set of N integers S={0,1,…,N−1} and a sequence of requests of the following form:

* Remove x from S
* Find the *successor* of x: the smallest y in S such that y≥x.

design a data type so that all operations (except construction) should take logarithmic time or better.

(Java Code on github at the bottom of the post. )

My thoughts and solutions:

Well, I have experienced some frustration on this one. Since it is under the context of Union-Find, it is very likely that we need to use Union-Find algorithm. Unfortunately, I didn’t figure out how to apply Union-Find. One step back, if there was no context at all, I probably would not think of using Union-Find at all…But we know now…Again, this suggests that we should practice more problems as the knowledge base, otherwise we won’t have any previous experience to connect with when facing a new problem.

This problem uses a similar approach as the previous post in solving the maximum object in the component. To fit in the union-find frame, the remove operation acts like union, well, with what? The answer is, if we remove x at index i in an array, it is the same as union (array[i], array[i+1]) (except for the last element). And the successor of x would be the largest object in the component which x is a member of. Just try this yourself.

I still can’t figure out how to relate this problem and union-find in the first place. This is the real challenge. If someone has an intuitive idea of how this works, please, leave a comment and share your brilliant idea :)

**Question 4**

**Union-by-size.** Develop a union-find implementation that uses the same basic strategy as weighted quick-union but keeps track of tree height and always links the shorter tree to the taller one. Prove a lg*N* upper bound on the height of the trees for *N* sites with your algorithm.

<http://jznest.wordpress.com/2014/02/17/algorithm-i-part-i-union-by-height/>

This one should be simple if you know how to union by size. Just change the array that holds the size of tree to an array that holds the height of the tree. And if a shorter tree connects to a taller tree, the height of the new tree is the same as the old tall tree. So when does the tree height increase? Well, the answer is that when two trees with the same height union together.