Quiz1:

**1.**

**Question 1**

**Social network connectivity.** Given a social network containing n*n* members and a log file containing m*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*m*log*n* or better and use extra space proportional to n*n*.

*Note: these interview questions are ungraded and purely for your own enrichment. To get a hint, submit a solution.*

**1 / 1 point**

in weighted quick union method, we keep the track of size of the tree, we can use this to check if all components are connected or not. we begin with the firs element of the time stamp and call the union function for them, we should notice that it is impossible for all component to be connected before n-1 steps, so after that every time that we call the union function for component of the time stamp we check if the size of the tree of our first element is equal to n or not when the size became equal to n we know that all elements are connected we can stop and report the time in time stamp array, if we reach the end of time stamp array and the size of the first component is not equal to n we must conclude that our elements are not entirely connected even at the end,

**Correct**

*Hint*: union−find.

**2.**

**Question 2**

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

For example, if one of the connected components is \ {1, 2, 6, 9\} {1,2,6,9}, then the {find()}find() method should return 99 for each of the four elements in the connected components.

**1 / 1 point**

in the union find method if we could keep track of the max element of the component along with ids in another list probably the problem is solved. I’m not sure it is the best way but it works. we set the first value for elements of this new list equal to that component assigned to that index, when we change the id of an element we check and if the component that its value us changing was greater than the maximum values assigned we change the maximum value to the new one.

**Correct**

*Hint:* maintain an extra array to the weighted quick-union data structure that stores for each root \mathtt{i}i the large element in the connected component containing \mathtt{i}i.

**3.**

**Question 3**

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

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

design a data type so that all operations (except construction) take logarithmic time or better in the worst case.

**1 / 1 point**

I didn’t understand the question I try it later;

**Correct**

Hint: use the modification of the union−find data discussed in the previous question.

Quiz 2:

**1.**

**Question 1**

**3-SUM in quadratic time.** Design an algorithm for the 3-SUM problem that takes time proportional to n^2*n*2 in the worst case. You may assume that you can sort the n*n* integers in time proportional to n^2*n*2 or better.

*Note: these interview questions are ungraded and purely for your own enrichment. To get a hint, submit a solution.*

**1 / 1 point**

we can sort the n integer in an array from small to large then we extract all of the two combination of these integer and compute the sum of each pair and save it in an array and sort it from large to small, the we begin from the beginning of our first array and binary search our second array for the same and with the opposite sign number, if we didn't found it we keep going and do it for the second item in the first array, but if we found it we increase the number of zero sums and although for the next step in the second array we will not consider the numbers that are between the top and out previous finding, we keep going till we reach the end of one of our arrays,

**Correct**

*Hint:* given an integer \mathtt{x}x and a sorted array \mathtt{a[ ]}a[] of n*n* distinct integers, design a linear-time algorithm to determine if there exists two distinct indices \mathtt{i}i and \mathtt{j}j such that \mathtt{a[i] + a[j] == x}a[i]+a[j]==x.

**2.**

**Question 2**

**Search in a bitonic array.** An array is *bitonic* if it is comprised of an increasing sequence of integers followed immediately by a decreasing sequence of integers. Write a program that, given a bitonic array of n*n* distinct integer values, determines whether a given integer is in the array.

* Standard version: Use \sim 3 \lg n∼3lg*n* compares in the worst case.
* Signing bonus: Use \sim 2 \lg n∼2lg*n* compares in the worst case (and prove that no algorithm can guarantee to perform fewer than \sim 2 \lg n∼2lg*n* compares in the worst case).

**1 / 1 point**

I didn't think about details of the code but I think we can do as i say, first we use binary search to find the index that the trend change, in the binary search that we used to know every time we compare the middle element of remaining of the list with our given number, here every time we take two middle element and see if the first one is greater or the last one, if the second one was greater we should search for our changing trend point in the right half and if not we should look for it in the left half, we keep going until we have only two element and check that index to be sure the trend changed there, when we have that index, we do our binary search for finding the given number, we look at the middle of our array, if the given number was less than that we continue the binary search in the regular way, if we found it se stop if we didn't fount it we do kind of reverse binary search for the decreasing part of our array to see if our number is there, if our number was greater than our middle element we do the binary search for the part between our changing trend index to the middle index.

**Correct**

*Hints*: Standard version. First, find the maximum integer using \sim 1 \lg n∼1lg*n* compares—this divides the array into the increasing and decreasing pieces.

Signing bonus. Do it without finding the maximum integer.

**3.**

**Question 3**

**Egg drop.** Suppose that you have an n*n*-story building (with floors 1 through n*n*) and plenty of eggs. An egg breaks if it is dropped from floor T*T* or higher and does not break otherwise. Your goal is to devise a strategy to determine the value of T*T* given the following limitations on the number of eggs and tosses:

* Version 0: 1 egg, \le T≤*T* tosses.
* Version 1: \sim 1 \lg n∼1lg*n* eggs and \sim 1 \lg n∼1lg*n* tosses.
* Version 2: \sim \lg T∼lg*T* eggs and \sim 2 \lg T∼2lg*T* tosses.
* Version 3: 22 eggs and \sim 2 \sqrt n∼2*n*​ tosses.
* Version 4: 22 eggs and \le c \sqrt T≤*cT*​ tosses for some fixed constant c*c*.

**1 / 1 point**

I don't understand the question, I will try later.

**Correct**

*Hints:*

* Version 0: sequential search.
* Version 1: binary search.
* Version 2: find an interval containing T*T* of size \le 2T≤2*T*, then do binary search.
* Version 3: find an interval of size \sqrt n*n*​, then do sequential search. Note: can be improved to \sim \sqrt{2n}∼2*n*​ tosses.
* Version 4: 1 + 2 + 3 + \ldots + t \; \sim \; \frac{1}{2} t^21+2+3+…+*t*∼21​*t*2. Aim for c = 2 \sqrt{2}*c*=22​.