# Union-find problem

## 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 mlogn or better and use extra space proportional to n.

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

## 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.

## 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) take logarithmic time or better in the worst case.

## Percolation

(see week 1 programming assignment online or local copy: [Programming Assignment 1\_ Percolation.html](Programming%20Assignment%201_%20Percolation.html))

# Running time and memory

## 3-SUM in quadratic time

Design an algorithm for the 3-SUM problem that takes time proportional to n2 in the worst case. You may assume that you can sort the n integers in time proportional to n2 or better.

## 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 distinct integer values, determines whether a given integer is in the array.

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

## Egg drop

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

* Version 0: 1 egg,  tosses.
* Version 1:  eggs and  tosses.
* Version 2:  eggs and  tosses.
* Version 3: 2 eggs and  tosses.
* Version 4: 2 eggs and tosses for some fixed constant c.

# Stacks and Queues

## Queue with two stacks

Implement a queue with two stacks so that each queue operations takes a constant amortized number of stack operations.

## Stack with max

Create a data structure that efficiently supports the stack operations (push and pop) and also a return-the-maximum operation. Assume the elements are reals numbers so that you can compare them.

## Java generics

Explain why Java prohibits generic array creation.

## Deque and Randomized Queues

(see week 2 programming assignment online or local copy: [Programming Assignment 2\_ Deques and Randomized Queues.html](Programming%20Assignment%202_%20Deques%20and%20Randomized%20Queues.html))

# Elementary sorts

## Intersection of two sets

Given two arrays **a[]** and **b[]**, each containing ***n*** distinct 2D points in the plane, design a subquadratic algorithm to count the number of points that are contained both in array a[] and array b[].

## Permutation

Given two integer arrays of size *n*, design a subquadratic algorithm to determine whether one is a permutation of the other. That is, do they contain exactly the same entries but, possibly, in a different order.

## Dutch national flag

Given an array of *n* buckets, each containing a red, white, or blue pebble, sort them by color. The allowed operations are:

* *swap*(*i*,*j*): swap the pebble in bucket *i* with the pebble in bucket *j*.
* *color*(*i*): color of pebble in bucket *i*.

The performance requirements are as follows:

* At most *n* calls to *color*().
* At most *n* calls to *swap*().
* Constant extra space.

*(see Quicksort lection notes for solution)*

# Mergesort

## Merging with smallest auxiliary array

Suppose that the subarray a[0] to a[n−1] is sorted and the subarray a[n] to a[2∗n−1] is sorted. How can you merge the two subarrays so that a[0] to a[2∗n−1] is sorted using an auxiliary array of length *n* (instead of 2*n*)?

## Counting inversions

An *inversion* in an array *a*[] is a pair of entries *a*[*i*] and *a*[*j*] such that *i*<*j* but *a*[*i*]>*a*[*j*]. Given an array, design a linearithmic algorithm to count the number of inversions.

## Shuffling a linked list

Given a singly-linked list containing *n* items, rearrange the items uniformly at random. Your algorithm should consume a logarithmic (or constant) amount of extra memory and run in time proportional to *n*log*n* in the worst case.

# Quicksort

## Nuts and bolts

A disorganized carpenter has a mixed pile of *n* nuts and *n* bolts. The goal is to find the corresponding pairs of nuts and bolts. Each nut fits exactly one bolt and each bolt fits exactly one nut. By fitting a nut and a bolt together, the carpenter can see which one is bigger (but the carpenter cannot compare two nuts or two bolts directly). Design an algorithm for the problem that uses *n*log*n* compares (probabilistically).

## Selection in two sorted arrays

Given two sorted arrays *a*[] and *b*[], of sizes *n*1 and *n*2, respectively, design an algorithm to find the *kth* largest key. The order of growth of the worst case running time of your algorithm should be log*n*, where *n*=*n*1+*n*2.

* Version 1: *n*1=*n*2 and *k*=*n*/2
* Version 2: *k*=*n*/2
* Version 3: no restrictions

## Decimal dominants

Given an array with *n* keys, design an algorithm to find all values that occur more than *n*/10 times. The expected running time of your algorithm should be linear.

# Priority Queues

## Dynamic median

Design a data type that supports insert in logarithmic time, find-the-median in constant time, and remove-the-median in logarithmic time.

## Randomized priority queue

Describe how to add the methods **sample**() and **delRandom**() to our binary heap implementation. The two methods return a key that is chosen uniformly at random among the remaining keys, with the latter method also removing that key. The **sample**() method should take constant time; the **delRandom**() method should take logarithmic time. Do not worry about resizing the underlying array.

## Taxicab numbers

A *taxicab* number is an integer that can be expressed as the sum of two cubes of integers in two different ways: *a^*3+*b^*3=*c^*3+*d^*3. For example, 1729=9*^*3+10*^*3=1*^*3+12*^*3. Design an algorithm to find all taxicab numbers with *a*, *b*, *c*, and *d* less than *n*.

* Version 1: Use time proportional to (*n^*2) log*n* and space proportional to *n^*2.
* Version 2: Use time proportional to (*n^*2) log*n* and space proportional to *n*.

# Elementary Symbol Tables

## Java autoboxing and equals()

Consider two double values a and b and their corresponding <tt>Double</tt> values x and y.

* Find values such that (a==b) is true but x.equals(y) is false.
* Find values such that (a==b) is false but x.equals(y) is true.

// 1: a = 0.0 / 1; b = 0.0 / (-1);

// 2: a = 0.0 / 0.0; b = 0.0 / 0.0;

## Check if binary tree is BST

Given a binary tree where each Node contains a key, determine whether it is a binary search tree. Use extra space proportional to the height of the tree.

## Inorder traversal with constant extra space

Design an algorithm to perform an inorder traversal of a binary search tree using only a constant amount of extra space.

// Morris traversal. No stack, no recursion

## Web tracking

Suppose that you are tracking *n* web sites and *m* users and you want to support the following API:

* User visits a website.
* How many times has a given user visited a given site?

What data structure or data structures would you use?

# Balanced Search Trees

## Red–black BST with no extra memory

Describe how to save the memory for storing the color information when implementing a red–black BST.

## Document search

Design an algorithm that takes a sequence of *n* document words and a sequence of *m* query words and find the shortest interval in which the *m* query words appear in the document in the order given. The length of an interval is the number of words in that interval.

## Generalized queue

Design a generalized queue data type that supports all of the following operations in logarithmic time (or better) in the worst case.

* Create an empty data structure.
* Append an item to the end of the queue.
* Remove an item from the front of the queue.
* Return the *ith* item in the queue.
* Remove the *ith* item from the queue.

# Hash Tables

## 4-SUM

Given an array *a*[] of *n* integers, the 4-SUM problem is to determine if there exist distinct indices *i*, *j*, *k*, and *l* such that *a*[*i*]+*a*[*j*]=*a*[*k*]+*a*[*l*]. Design an algorithm for the 4-SUM problem that takes time proportional to *n*2 (under suitable technical assumptions).

## Hashing with wrong hashCode() or equals()

Suppose that you implement a data type OlympicAthlete for use in a java.util.HashMap.

* Describe what happens if you override hashCode() but not equals().
* Describe what happens if you override equals() but not hashCode().
* Describe what happens if you override hashCode() but implement public boolean equals(OlympicAthlete that) instead of public boolean equals(Object that).