Quiz 1 Solutions

- Don't Panic.
- Write your name on every page.
- The quiz contains sven problems. You have 100 minutes to earn 100 points.
- The quiz contains 20 pages, including this one and 3 pages of scratch paper.
- The quiz is closed book. You may bring one double-sided sheet of A4 paper to the quiz. (You may not bring any magnification equipment!) You may **not** use a calculator, your mobile phone, or any other electronic device.
- Write your solutions in the space provided. If you need more space, please use the scratch paper at the end of the quiz. Do not put part of the answer to one problem on a page for another problem.
- Read through the problems before starting. Do not spend too much time on any one problem.
- Show your work. Partial credit will be given. You will be graded not only on the correctness of your answer, but also on the clarity with which you express it. Be neat.
- Good luck!

Problem #	**Name**	**Possible Points**	**Achieved Points**
1	Sorting Jumble	12	
2	Algorithm Analysis	16	
3	Sorting Tweets	12	
4	Buggy Code I: Gravitational Waves	12	
5	Buggy Code II: Math	12	
6	Sorting Strange Sequences	14	
7	The Marble Factory	22	
Total:		100	

Name: ______________________________ Student id.: ______________________________

Please circle your discussion group:

Kai Yuan Mon. 2-4	Choyuk Tues. 10-12	Curtis Tues. 12-2	Jonathan Tues. 2-4	Davin Tues. 4-6	Shi-Jie Thurs. 10-12	Leonard Thurs. 4-6

Problem 1. Sorting Jumble. [12 points]

The first column in the table below contains an unsorted list of Silicon Valley companies. The last column contains a sorted list of companies. Each intermediate column contains a partially sorted list of companies.

Each intermediate column was constructed by beginning with the unsorted list at the left and running one of the sorting algorithms that we learned about in class, stopping at some point before it finishes. Each algorithm is executed exactly as described in the lecture notes. (One column has been sorted using a sorting algorithm that you have not seen in class.)

Identify, below, which column was (partially) sorted with which algorithm. *Hint: Do not just execute each sorting algorithm, step-by-step, until it matches one of the columns. Instead, think about the invariants that are true at every step of the sorting algorithm.*

Unsorted	**MergeS**	**InsertionS**	**SelectionS**	**BubbleS**	**ShellSort**	**QuickSort**	**Sorted**
eBay	Apple	Apple	Apple	Cisco	eBay	Apple	Apple
Cisco	Cisco	Cisco	Cisco	eBay	Cisco	Cisco	Cisco
Oracle	eBay	eBay	eBay	Apple	Apple	eBay	eBay
Salesforce	LinkedIn	Oracle	Facebook	LinkedIn	Google	Intel	Facebook
Yelp	Oracle	Salesforce	Google	Microsoft	HP	Quora	Google
VMware	Salesforce	VMware	HP	Facebook	Facebook	Oracle	HP
Apple	VMware	Yelp	Oracle	Oracle	Oracle	LinkedIn	Intel
LinkedIn	Yelp	LinkedIn	LinkedIn	Google	Intel	Microsoft	LinkedIn
Microsoft	Facebook	Microsoft	Microsoft	PayPal	Microsoft	Facebook	Microsoft
Facebook	Microsoft	Facebook	Salesforce	HP	VMware	PayPal	Oracle
PayPal	Google	PayPal	PayPal	Quora	PayPal	Google	PayPal
Google	PayPal	Google	Yelp	Intel	LinkedIn	HP	Quora
Twitter	Twitter	Twitter	Twitter	Salesforce	Twitter	Salesforce	Salesforce
HP	HP	HP	VMware	Twitter	Yelp	Twitter	Twitter
Quora	Quora	Quora	Quora	VMware	Quora	VMware	VMware
Intel	Intel	Intel	Intel	Yelp	Salesforce	Yelp	Yelp
Unsorted	**A**	**B**	**C**	**D**	**E**	**F**	**Sorted**

2pts each

Problem 2. Algorithm Analysis [16 points]

For each of the following, choose the best (tightest) asymptotic function T from among the given options. Some of the following may appear more than once, and some may appear not at all. **Please write the letter in the blank space beside the question.**

A. $\Theta(1)$ B. $\Theta(\log n)$ C. $\Theta(n)$ D. $\Theta(n \log n)$

E. $\Theta(n^2)$ F. $\Theta(n^3)$ G. $\Theta(2^n)$ H. None of the above.

4pts each

Problem 2.a.

$$T(n) = \left(\frac{n^2}{17}\right)\left(\frac{n}{4}\right) + \frac{n^3}{n-7} + n^2 \log n$$

$T(n) =$ **Sol:** F : $O(n^3)$

Problem 2.b. The running time of the following code, as a function of n:

```
public static int loopy(int n){
    int j = 1;
    for (int i=0; i<n; i++) {
        for (int k=j; k>0; k--) {
            System.out.println("Hello.");
        }
        j *= 2;
    }
    return j;
}
```

$T(n) =$ **Sol:** G : $O(2^n)$

Problem 2.c. $T(n)$ is the running time of a divide-and-conquer algorithm that divides the input of size n into two equal-sized parts and recurses on both of them. It uses $O(1)$ work in dividing/recombining the two parts (and there is no other cost, i.e., no other work done). The base case for the recursion is when the input is of size 1, which costs $O(1)$.

$T(n) =$ $\boxed{\textbf{Sol: } \text{C} : O(n)}$

Problem 2.d. The running time of the following code, as a function of n:

```
public static int recursiveloopy(int n){
    for (int i=0; i<n; i++) {
        for (int j=0; j<n; j++) {
            System.out.println("Hello.");
        }
    }

    if (n == 1) {
         return 1;
    }
    else {
        return (recursiveloopy(n/2));
    }
}
```

$T(n) =$ $\boxed{\textbf{Sol: } \text{E} : O(n^2)}$

Problem 3. Sorting Tweets [12 points]

Tweets are strings of 140 characters, and they vary significantly in quality. Some tweets are great and some are useless. Assume that you have already implemented a function:

`evaluateTweet(String tweet)`

that returns an integer between 1 and 10. (You do not need to implement the `evaluateTweet` method.) Ten indicates a tweet of the highest quality, and one a tweet of the lowest quality.

Your goal is to sort a list of tweets so that all tweets are ordered first by quality and then alphabetically:

- If tweet t_i has higher quality than tweet t_j, then it always comes later in the list.
- If tweets t_i and t_j have the same quality, then they are ordered alphabatetically as in standard dictionary order.

For example, your sorted list of tweets[1] might be as follows:

```
1 : "Can the height of a binary search tree be less than that of a red-black tree?"
5 : "Delete a consecutive range of leaves from a binary tree"
5 : "Generating uniformly distributed random numbers using a coin"
7 : "Approximation for vector bin packing"
7 : "How is the space hierarchy theorem different for non space constructible functions?"
```

Here the number represents the quality, and it is followed by the string associated with the tweet.

[1]Tweets taken live from the StackExchange twitter feed.

Problem 3.a. Assume that Alice is writing a special sorting function `QualitySort` that will sort your list of tweets by quality from smallest to largest, and Bob is writing a special sorting function `AlphaSort` that will sort your list of tweets alphabetically from smallest to largest.

You will sort your list by first executing one of these sorting algorithms and then the other. (You will not modify the list in any other way except by executing `QualitySort` and `AlphaSort`.)

The Pointy-Haired Boss, strangely, requires that one of the sorting algorithms is implemented with MergeSort (in the usual top-down manner) and the other is implemented with QuickSort (using a random pivot).[2] You get to choose: which function will use MergeSort and which will use QuickSort?

`QualitySort` : ______________ **Solution:** MergeSort

`AlphaSort` : ______________ **Solution:** QuickSort

2 pts each

Explain briefly (in 1-2 sentences) why:

Solution: The list will be properly sorted if we sort *first* by alphabet and *second* by quality. However, the `QualitySort` will only maintain the alphabetical order for tweets of the same priority if it is stable. Hence you want to use MergeSort (the stable sort) for QualitySort, and QuickSort (which is not stable) for AlphaSort.

4 pts with the word stable
2 pts with any answer

For this page, 2 pts "mercy" marks for any answers given.

[2]He believes that this will help ensure the robustness and diversity of the codebase.

Problem 3.b. You decide to ignore the Pointy-Haired Boss and *not* use Alice and Bob's sorting methods. (*Notice this part is different from Part (a).*) Instead, you decide to use the generic `Arrays.sort` method provided in the Java libraries, which will sort an array of any class that supports the Comparable interface. You will use the following code to sort your tweets:

```
Tweet[] listOfTweets = getTweets(); // retrieves tweets from Twitter
Arrays.sort(listOfTweets); // sorts the list from smallest to largest
```

To do this, you must implement a `Tweet` class that supports the Comparable interface in just the right way so that a single invocation of `sort` will result in a properly sorted list.

Assume that you have already coded all the functionality in your Tweet class *except* for the Comparable interface. Your job is to modify the code below for the Tweet class so that it fully supports the Comparable interface. Your implementation should be such that we can sort the tweets properly (by both quality and alphabetically) with a single call to the sort function.

Solution:

```
public class Tweet implements Comparable<Tweet>
{
  String m_tweet = null;

  Tweet(String tweet) {
      m_tweet = tweet;
  }
  private static int evaluateTweet(String t){
      // implementation removed
  }
  // All other Tweet class implementation removed

  public int compareTo(Tweet other) {
      int qOther = evalutateQuality(other);
      int qThis = evaluateQuality(this);

      if (qThis > qOther) {
          return 1;
      }
      else if (qThis < qOther) {
          return -1;
      }
      else {
          return this.m_tweet.compareTo(other);
      }
  }
}
```

basically 4 pts
-1 pts if use
"m_tweet > other"
directly

Problem 4. Buggy Code: Measuring Gravitational Waves [12 points]

Last night, in an exciting new discovery, physicists announced the first direct detection of gravitational waves, first predicted by Einstein almost 100 years ago! On the next page, you will find some buggy code that *failed* to detect gravitational waves. The main method (which is correct Java code) for the buggy graviational wave detector class looks like this:

```
public static void main(String[] args){
    GravitationalWaveDetector mho = new GravitationalWaveDetector(1000, 1000);
    System.out.println("The maximum wave size is: " + mho.maxData());
}
```

Please identify each of the bugs in the `GravitationalWaveDetector` class on the next page. Only identify bugs that will prevent compilation, or will cause the program to crash.[3] There are four severe bugs. Fill in the following table, specifying for each bug the line number, whether it causes the program to fail to compile or to crash, and briefly describe the problem.

Basically 1 point for each box

Bug	Line number	Compile error or crash?	Explain the bug
1.	28	Compile error	Missing **void** in the method signature for recordData.
2.	23	Compile error	Missing **super(s)** in the constructor. (No empty constructor in parent class.)
3.	38	Compile error	Cannot directly compare GWdata. Need to compare value, not classes.
4.	30	Crash	Variable **data** never initialized. Null deference on head of list.

[3] Please do not identify warnings, such as the requirement that each class is in its own file, or that some variables may be unused. Do not identify stylistic problems, such as missing comments or bad variable names. Do not identify problems that cause the computation to produce a different answer than you think it should. Only identify problems that either prevent the program from compiling or cause it to crash.

```
public class Interferometer {
    protected int size;

    Interferometer(int s){
        size = s;
    }
}

public class GravitationalWaveDetector extends Interferometer {
    private class GWData {
        private int value;
        private GWData next;

        GWData(int v) {
            value = v;
        }
    }

    private int height;
    private GWData data;

    GravitationalWaveDetector(int s, int h){
        height = h;
        size = s;
        recordData(s*h);
    }

    private recordData(int n){
        for (int i=0; i<n; i++){
            data.next = new GWData(i);
        }
    }

    public int maxData(){
        GWData reading = data;
        GWData max = reading;
        while (reading != null){
            if (reading > max) {
                max = reading;
            }
            reading = reading.next;
        }
        if (max != null) {
            return max.value;
        }
        return 0;
    }
}
```

Problem 5. Buggy Code (continued): Math [12 points]

The code on the next page is intended to implement a calculator. Unlike the previous problem, this code compiles and runs without crashing.

Assume for this problem that both `getIntegerKey` and `getOperatorKey` are correctly implemented, and the code is simply omitted here. The `getIntegerKey` method returns the number that the user pressed on the keypad of the calculator. The `getOperatorKey` method returns the operator that the user pressed on the keypad. (All the legal operators are given here.) You may also assume that all the values and answers are integers.

Problem 5.a. Explain the problem with this code, and given an example of when it prints the wrong answer to the screen.

Solution: The problem here is that we check, using `instanceof` whether the operator is Plus or Minus, but we omitted Multiply. Hence if the user inputs “8 x 4” this code will return the answer 0.

Problem 5.b. Modify the code (by crossing out and adding new code directly on the next page) so that it is correct, and also so that it does not use the `instanceof` keyword anywhere in the code. When correct, the program should apply the proper operator (returned by the `getOperatorKey` method) to the two integers (returned by the `getIntegerKey` method calls) and print out the correct answer. *Do not modify the main method.*

```java
public class Operator {

    public int calculate(int a, int b){
        // Really, it would be better to throw an exception here
        return 0;
    }

    public static int getIntegerKey() {
        // implementation omitted
        // return key pressed by user
    }

    public static Operator getOperatorKey() {
        // implementation omitted
        // return operator pressed by user
    }

    public static void main(String[] args){
        int first = getIntegerKey();
        Operator op = getOperatorKey();
        int second = getIntegerKey();
        System.out.println("The answer: " + op.calculate(first, second));
    }
}

class Plus extends Operator {
    public int calculate(int a, int b) {
        return (a+b);
    }

}
class Minus extends Operator {
    public int calculate(int a, int b) {
        return (a-b);
    }

}
class Multiply extends Operator {
    public int calculate(int a, int b) {
        return a*b;
    }

}
```

Problem 6. Sorting Strange Sequences. [14 points]

Problem 6.a. In class, we always assumed that we were sorting an array. What if you are sorting a linked list? (For the purpose of this problem, assume you have a standard doubly-linked list with a head and a tail pointer. It is a single linked list, not a SkipList.) For each sorting algorithm below:

- Give the asymptotic running time when executed on a linked list.
- Give the asymptotic amount of extra space needed *outside* the linked list.
- Briefly explain how the implementation differs on a linked list from an array.

(Remember how accessing a linked list differs from accessing an array.)

InsertionSort: Running time: Extra space:

2 + 2
+ 1

Brief explanation:

Solution: InsertionSort still runs in $O(n^2)$ time and $O(1)$ extra space. Notice that all the steps of the algorithm involve swapping adjacent items (which is easy to implement in a doubly linked list) and scanning the array from one end to the other (which is easy to do in a doubly-linked list).

MergeSort: Running time: Extra space:

2 + 2
+ 1

Brief explanation:

Solution: MergeSort still runs in $O(n \log n)$ time, but now it only requires $O(1)$ extra space: we don't need an extra array to do the merge in! The key part that needs to be reimplemented is the Merge operation.

To do a MergeSort, first, we can the linked list in $O(n)$ time to find the midpoint. We then break the list into two lists at the midpoint, and recursively MergeSort the two halves. To do the merge, we start at the beginning of our two lists and weave the two lists together: iteratively insert the first item from the second list into the first list by scanning the first list until we find the right place; then continue with the remaining items from the second list.

Some stated
O(log n)
space for
stack/keep
track of
pointers. still
give full
marks for that

Problem 6.b. Imagine you are sorting a set of large items on disk. For example, each item in the array is a 500MB video file (and you want to store the files on disk in the proper order).

Comparing two items is very fast: you only need to read a small amount of the file to determine the proper order. In general, reading the array is very fast. But moving or copying the files is very, very slow. (For example, each swap of two elements in the array might take 10 or 20 seconds!) Which sorting algorithm that we have studied in class should you use? Why?

Sorting algorithm: **Solution:** SelectionSort

2
+ 2

Explanation:

Solution: Here we want an algorithm that minimizes the number of swaps. SelectionSort has at most n swaps, and hence is the most efficient possible. Both MergeSort and QuickSort have potentially many more swaps (even if they have fewer comparisons).

Problem 7. The Marble Factory [22 points]

After graduating from NUS, you decide to open a marble factory! Back in the olden days, before Minecraft and PlayStations, everyone used to play with marbles. Maybe they will again! First, you purchase a large warehouse for shipping boxes of marbles to your customers. Assume that every order contains at most M marbles. You are given a list of orders for the day, for example:

To ship:

```
Order 1: 150 marbles
Order 2: 573 marbles
Order 3: 1 marble
Order 4: 4325 marbles
```

All the boxes in your warehouse are the same size S. Each order is packed into the minimum number of boxes. For example, if $S = 40$, then Order 1 of 150 marbles take 4 boxes. (Two different orders cannot be combined in the same box.)

Problem 7.a. Give a linear time algorithm `numBoxes(S, orders)` that calculates the exact number of boxes of size S you need to ship all the requests on the list `orders`. Use pseudocode and briefly explain (in one to two sentences) how your algorithm works.

Solution: Here we simply perform a linear scan of the list of orders, for each order calculating the number of boxes needed needed for that order and maintaining the sum.

Solution:

```
numBoxes(S, orders)
   sum = 0
   for i = 0 to (length(orders)-1) do
       boxes = ceiling(orders[i] / S)
       sum = sum + boxes
   return sum
```

Problem 7.b. Your warehouse can ship at most T boxes per day. (The loading dock is not big enough to ship more.) Moreover, it is generally cheaper to ship smaller boxes than bigger boxes. Therefore we want to find the smallest size S so that we can pack all our orders in at most T boxes. Consider the following algorithm, which takes as input a list of `orders`:

```
1.  int findSize(Orders[] orders, int M, int T) {
2.      if (orders.length > T) return -1; // IMPOSSIBLE
3.
4.      for (int S=M; S >= 1; S--) {
5.          int num = numBoxes(S, orders);
6.          if (num > T) return S+1  ;
7.      }
8.      // If we arrive here, it means that even packing
9.      // one marble per box, we need no more than T boxes
10.     return 1;
11. }
```

Assume that there are n orders. Recall that M is the maximum size of any order.

What is the (tight) asymptotic running time of this algorithm as a function of n and M?

Sol: $O(nM)$

Explain briefly (in one or two sentences) why this algorithm works. (In particular, explain why it is safe to return on line (6) without examining any of the smaller values of S.)

Solution: The key point here is that the number of boxes you need is non-increasing in S: the bigger the value of S, the fewer boxes you need (with a minimum of n boxes when each box has size M). Since the maximum box size is M and the minimum box size is 1, the for loop on line 4 explores all the possible box size.

To put it in the opposite direction, as we decrease S (in the for loop on line 4), the number of boxes we need only increases. Therefore, if for some value of S we discover that we need too many boxes (i.e., the number of boxes $> T$), then if decrease S further, we will only need *more* boxes. Hence in that case, it is impossible that a smaller value of S is better. Thus the value we return on line 6 is the minimum acceptable value.

(A more careful proof would also point out that the initial value of M is always an acceptable solution, if we enter the for loop. Otherwise it is impossible.)

Problem 7.c. Give a more efficient algorithm for finding the smallest value of S that allows us to pack all the orders in at most T boxes. Assume that the total number of marbles in all the orders is very large, and that T is also very large. For example, you might think of $T > n^2$, where n is the number of orders. Analyze your solution as a function of n and M (and not any other parameters).

In less than eight words, what is your basic approach: **Solution:** Binary search

What is the running time of your approach? **Solution:** $O(n \log M)$

Solution: As we already argued in the previous part, the number of boxes is non-increasing in S. Or to put it differently, imagine that (hypothetically) compute an array `Boxes[M]` where `Boxes[i]` is equal to the number of boxes we need if the boxes have size i. Then this array is sorted from biggest to smallest (with many duplicate values). However, we do not want to *actually* compute this array because that would take $O(nM)$ time.

Instead, we want to do a binary search in this imaginary array, searching for the value T. We want to find the smallest value in this array that is $\leq T$.

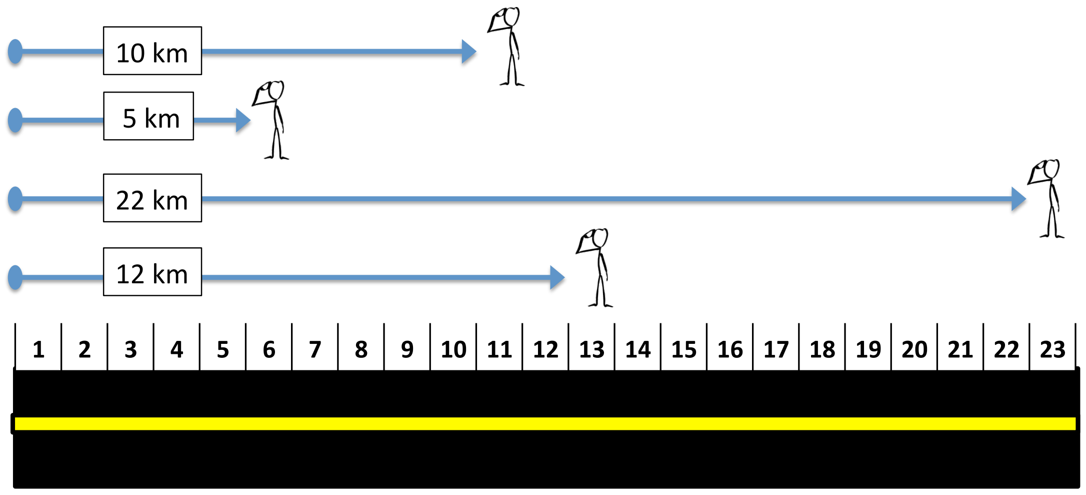
```
binarySearch(orders, T)
   // First make sure it is not impossible
   if (numBoxes(1, orders) < 0) return -1

   // Now do binary search
   return binarySearch(orders, 1, M)

binarySearch(orders, min, max)
   // Base case
   if (min >= max)
       return min

   // Find the midpoint
   mid = min + (max-min)/2
   boxes = numBoxes(mid, orders)

   // Recurse, maintain the invariant:
   // There is always a feasible size in the range [min, max]
   if (boxes > T)
       return binarySearch(orders, mid+1, max)
   else
       return binarySearch(orders, min, mid)
```

Problem 3. The Ultra Ultra Ultra Running Man. [20 points]

A group of n runners decides to run an ultra ultra ultra race of M kilometers, a distance so far that no one can complete it in a lifetime of running. Each runner runs as far as he/she can, and then stops. Your job is to build a data structure to calculate statistics about the race. Specifically, your data structure should support two operations:

- `addRunner(int d)` which adds a new runner who ran d kilometers to the data structure.
- `howManyRunners(int d)` which returns the number of runs that made it to distance *at least* d.

Consider the following example:



Here you see four runners, who are added to the data structure as follows:

```
addRunner(10)
addRunner(5)
addRunner(22)
addRunner(12)
```

If you were to then query the data structure: `howManyRunners(11)` it would return the answer 2. If a new runner were added via the operation `addRunner(42)`, then the query `howManyRunners(11)` would return 3.

Assume there are n runners, with a maximum distance of M kilometers. (For simplicity, you may assume that n and M are powers of 2.) Give the most efficient implementation of this data structure. Both operations should be efficient, and both the running time and space usage should depend only on n (not M). Partial credit will be given for solutions whose running time or space depends on M (as long as the implementation is reasonably efficient).

Problem 3.a. Describe your solution briefly in two sentences.

Solution: Use a Rank Tree (implemented as an AVL tree, augmented to store weights). Insert each endpoing into the Rank Tree, and the count is exactly the number of runners who end after a certain point.

Problem 3.b. What is the running time of each operation in your data structure?

`addRunner`	:	$\boxed{}$	**Solution:** $O(\log n)$
`howManyRunners`	:	$\boxed{}$	**Solution:** $O(\log n)$

Problem 3.c. Provide more details explaining how your data structure works (and why it works). Give pseudocode as necessary for clarity.

Solution: Consider the following observation: the number of runners who make it to location j is exactly the number of runners that finish at location j or later. Hence, we build an AVL tree augmented to support `Rank` queries, exactly as discussed in class. For each runner, insert the kilometer on which the runner stops in the data structure. In order to determine the number of runners that make it to kilometer at least j:

- First find the successor of j in the AVL tree; let ℓ be the successor of j. (If j exists in the tree, then $\ell = j$.)
- Query the rank of ℓ. Let k be the rank of ℓ. (Assume that the first element has rank 0 and the last element has rank $n - 1$.)
- Let n be the total number of runners inserted. (Keep track of this separately.)
- Return $n - k$.

This data structure uses space $O(n)$ and has running time $O(\log n)$ for both operations.

There are several other less good solutions. The simplest solution, which is *not ok*, is to use an array of size M with one slot per day. In this case, the space is $\Theta(M)$, adding an event is $O(n)$, and querying the number of events is $O(1)$.

Alternatively, you might use an interval tree, exactly as described in class. For each runner, add an interval to the tree. To query the number of runners for a specific kilometer, repeatedly query for an interval containing that kilometer, deleting each interval as it is found. Once you have found all the intervals that intersect the given day, reinsert them. This data structure has space $O(n)$, adding an event has cost $O(\log n)$, and querying a day has cost $O(n \log n)$.

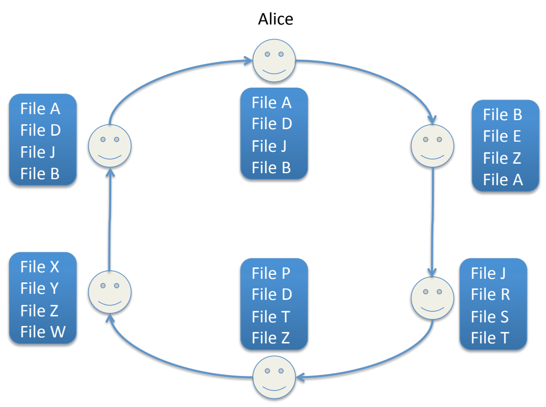
To do better, imagine a data structure consisting of a binary tree built recursively as follows: the root contains the interval $[1, M]$, the left sub-tree is built recursively for the interval $[1, M/2]$ and the right sub-tree is built recursively for the interval $[M/2 + 1, M]$. A node at height j is responsible for for an interval of size 2^j (where a leaf has height 0). Notice that this tree has height $\log M$. This tree takes $O(M)$ space. Each node in the tree stores a count of the number of events that fully cover the interval the node is responsiblef or. To insert an event, we do a tree walk as follows:

Assume we are current at node u in the tree, which is responsible for the interval $[u_1, u_2]$. Assume the right child of u is responsible for the interval $[v, u_2]$. Assume we are inserting the interval $[1, x]$. If u is a leaf, then increment the count at u if $x == u_1 == u_2$. If $x \geq u_2$, then increment the count of events at u and halt. If $x \geq v$ and $x < u_2$, then increment the count of events at $u.left$ and recurse in the right sub-tree. If $x < v$, then recurse in the left sub-tree.

Notice that the interval is now broken up into $O(\log n)$ pieces, with one node in the tree responsible for each piece. To query the number of events on a given day, start at the appropraite leaf of the tree and walk up to the root, summing the counts observed. Thus, inserting an event or querying the count has running time $O(\log M)$.

Problem 4. Peer-to-Peer Networking [20 points]

Imagine a peer-to-peer network consisting of n users arranged in a ring. Each user has k (large) files. Files are duplicated in the network, i.e., each file may be stored by multiple users. (Each file is only stored at most once by each user.) Assume that the files are very, very large (especially compared to n and k).



Alice wants to know how many times each of her files appears on the network. (Perhaps if a given file appears too many times, we should delete some of the copies; if a given file does not appear enough times, we should increase the number of copies.) In the above example, $k = 4$, $n = 6$, and each of Alice's files appears three times in the network.

Alice will send messages around the ring to try to answer this question. The goal is use the minimum amount of communication. We will consider a simplified networking model:

- Alice creates a data structure locally.
- Alice sends it to her right neighbor, who receives an exact copy of the data structure.
- When a user receives the data structure, they can modify it and send it on to his/her right neighbor in the same manner.
- When Alice receives a copy of the data structure from her left neighbor, she stores it locally overwriting the original version).

We assume that each transmission of the data structure costs *exactly* the size of the data structure as it is stored locally. (That is, we will not consider data compression or any other tricks for saving bandwidth.) Thus if the data structure has size m, then the total communication cost of sending the data all the way around the ring once is $m \cdot n$. For example, if the data structure is an array of k integers, and each integer is stored in $\log n$ bits, then the total communication cost is $kn \log n$. (Whether an entry in the array is "empty" or not does not matter.)

Thus for the purpose of this problem, minimizing the size of the data structure is equivalent to minimizing the communication cost.

Problem 4.a. Alice first suggests the following strategy for counting how many times each of her k files appears in the ring:

- She chooses a hash function h that maps each of her files to an integer in the range $[0, \ldots, k^5 n^7]$. (The range is chosen to be particularly large—you should not assume this is the best or optimal choice of range, or attribute any special meaning to the values 5 and 7.)
- Assume h satisfies the simple uniform hashing assumption.
- The data structure consists of two arrays F (for file) and C (for count), each of size k.
- In cell $F[i]$ she stores $h(f[i])$, where $f[i]$ is file i.
- In cell $C[i]$ she stores 1 to indicate the count of 1.
- Whenever a user receives the data structure containing F and C, she checks whether any of her files has the same hash as any of the files in F. If so, she increments the appropriate counter in C.
- In the end, Alice reads the counters in C to determine the number of times each file appears in the ring.

Assuming that you can store a value in the range $[0, \ldots M-1]$ using $\log M$ bits, what is the total (asymptotic) communication cost of this approach:

Solution: The data structure is of size $k \log n + k \log k^5 n^7 = O(k \log k + k \log n)$. Thus the total communication cost is $O(kn \log k + kn \log n)$.

Will the final count received by Alice be: (circle all that are correct)

Possibly too low | **Always exactly right** | **Possibly too high**

Explain why:

Solution: The answer may be too high due to collisions. If any other file in the network has the same hash function as one of Alice's, it will increase the count erroneously, resulting in a count that is too high. (The count cannot be too low.)

Problem 4.b. Alice is worried that the counts may not be correct. (Is Alice right to be worried? See previous page.) Alice decides to try another approach: a hash table.

Alice's new data structure is a hash table of size M, where collisions are resolved with chaining. Her hash table is implemented exactly as discussed in class (or as implemented in the Java 7 library). She inserts each of her files into the hash table, with the file as the key and a count as the value.

For each of her own files, she increments the count to 1. She then sends the hash table around the ring. Each user that receives the hash table looks up all of her files in the table; if a user finds her file in the hash table, she increments the count.

Assume that M is chosen to be sufficiently large so that all of the linked lists in the hash table are of length at most 3. For this part, assume that the size of M is reasonable, and assume that communication costs of $O(M)$ or $O(Mn)$ or even $O(Mnk)$ are reasonable (even though in reality they are not).

Why is this a very bad data structure for solving the counting problem? What mistake did Alice make? Explain briefly, e.g., in at most three to four sentences.

Solution: The problem is that a hash table with chaining stores both the key and the value. That means that each of the Alice's k files is stored in the hash table itself! That is a disaster, leading to very high communication costs.

Problem 4.c. Next, Alice tries to use a Fingerprint Hash Table. She creates a Fingerprint Hash Table with M entries, but stores an integer (of size $\log n$) in each cell (instead of storing just a binary 0/1 value). This integer is used to store the count of the number of files that hash to this cell. (Each file can appear at most n times in the network.) She then sends this Fingerprint Hash Table around the ring, and each user looks up all of her files in the table; if a user finds a count of at least one in a given entry, then she increments the count.

What is a good choice for the value of M (as a function of n and k) to ensure that the expected error on the count is only plus/minus $O(1)$?

Explain your answer briefly:

Solution: Choose $M = nk$. There are at most nk files in the network. Assume a given file of Alice's hashes to slot x. Each time a file in the network is hashed to see if it matches Alice's file, there is a probability of $1/M$ that it chooses slot x. Thus, the expected number of items that will collide with Alice's file is nk/M. By choose $M = nk$, we ensure that the expected number of collisions is 1, as desired.

Assuming the table is of size M, what is the total communication cost of the Fingerprint Hash Table solution?

Explain your answer briefly:

Solution: The Fingerprint hash table is an array of size M containing $\log n$ bits each. Hence the total size of the data structure is $M \log n$, and the total communication cost is $Mn \log n$. Substituting $M = nk$, we get a total cost of $kn^2 \log n$, which you will notice is significantly worse than the solution in Part (a).

Scratch Paper

Scratch Paper

Scratch Paper