This is a Week 5 midterm/quiz from CS2020 2016.

Beware that it may cover some material that we have not seen in CS2040S, and there are topics we have covered in CS2040S that were not on this quiz.

Quiz 1

- 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

432165

One of my favorite questions on sorting. Do you remember how these sorting algorithms work?

A hint: you do not actually need to remember most of the step-by-step details of these algorithms if you can just remember the invariants.

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	**A**	**B**	**C**	**D**	**E**	**F**	**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**

Please write the proper number in the blank space beside the letter:

A _ 4	1. BubbleSort
B _ 3	2. SelectionSort
C _ 2	3. InsertionSort
D _ 1	4. MergeSort (top down, sorts top half before bottom half)
E _ 6	5. QuickSort (with first element as the pivot)
F _ 5	6. None of the above

A little bit of theory practice…

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.

Problem 2.a.

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

$T(n) =$ F

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;
}
```

$T(n) =$ G

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 $O(1)$.

$T(n) =$ C

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));
    }
}
```

$T(n) =$ E

Another fun little sorting puzzle…

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  :  ____________________

AlphaSort    :  ____________________
```

Explain briefly (in 1-2 sentences) why:

[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.

```
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 t) {
            int diff = evaluateTweet(m_tweet) - t.evaluateTweet(t.m_tweet);
            if (diff == 0) {
                    if m_tweet == null return -1;
                    return m_tweet.compareTo(t.m_tweet);
            } else {
                    return diff
            }
    }

}
```

We haven't spent as much time in CS2040S on Java and OOP as we did in CS2020. (However, we have spent some time on Java.)

Again, this is a pretty tricky Java focused question (and one that very few people got complete right).

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.

Bug	Line number	Compile error or crash?	Explain the bug
1.			
2.			
3.			
4.			

[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.

```java
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;
    }
}
```

And some more Java! In CS2020, we spent a lot of time in the first 3 weeks on Java (because we had to cover all the material from CS2030 as well as CS2040 in one class).

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.

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.*

```
public class Operator {

    public int calculate(int a, int b){
        if (this instanceof Plus) {
            return (a + b);
        }

        if (this instanceof Minus) {
            return (a - b);
        }
        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 {




}
class Minus extends Operator {




}
class Multiply extends Operator {




}
```

More sorting algorithms. By now you really know how all those sorting algorithms work, right?

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: O(n^2) Extra space: O(1)

Brief explanation:

InsertionSort select an element in array order and sort it into the right place in the prefix.
Array achieve this by slotting the element in place.
Using a linked list, the element that is currently being sorted can be detached from the unsorted list and placed in a sorted linked-list in the right location.

MergeSort: Running time: O(nlogn) Extra space: O(n)

Brief explanation:

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: Selection sort

Explanation:

Makes the minimum number of swaps. But alot of comparison.

And, as always, a good problem to practice your problem solving techniques!

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.

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

Iterate through the list of orders and calculate the number of boxes required for each order. Return the total sum of boxes

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 ~~num~~+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?

O(M*n)

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.)

At the end of every for loop, the size of the box is such that we need <= T boxes (num <= T). The loop will stop if the size reaches the turning point such that num > T, meaning that S + 1 is the smallest size and any smaller values of S will result in more than T boxes

As box capacity decreases, num of boxes increases

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:

Binary search to find turning point for S

What is the running time of your approach?

nlogM

That was the end of the midterm quiz. Here are a few more problems from later in the year.

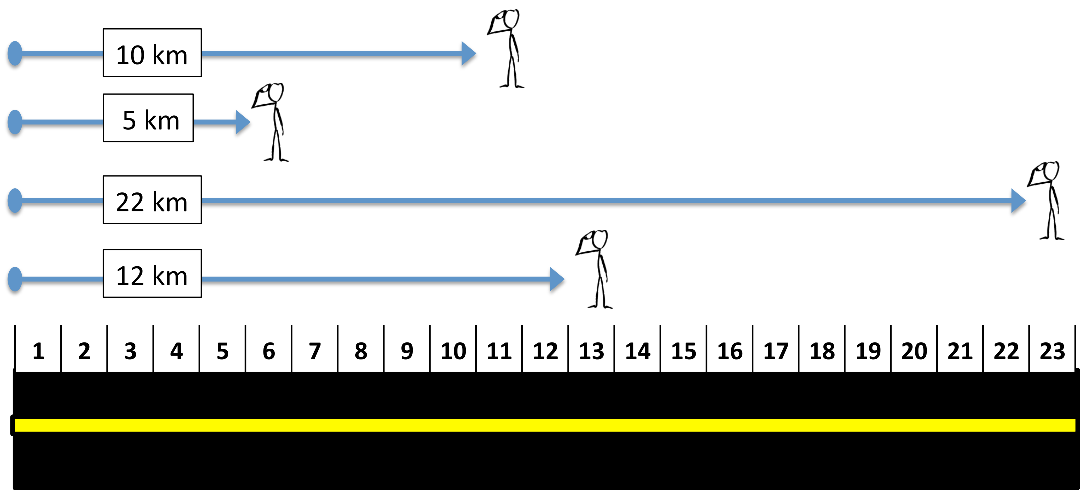
Problem 3. The Ultra Ultra Ultra Marathon. [20 points]

A group of n runners decide to run an ultra ultra ultra marathon 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:

This problem should look similar to some problems we have seen recently!



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.

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

Build an AVL tree, with each tree node storing the left.weight + right.weight + 1
addRunner() is similar to insert() while howManyRunner(v) will search for predecessor(v)
then get the weight of the right

Rank tree - an AVL tree augmented with weights ---- reuse order statistics tree

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

`addRunner` : O(logn)

`howManyRunners` : O(logn)

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

```
addRunner: traverse down the tree. left child <= , right child >
if empty, set left, set right
Update height. Rebalance tree

howManyRunners(v)
search(v) : if traverse left.weight + right.search(v) ------> rank() <-- black box
if (v in tree) -> return right.weight + 1. (weight of empty child = 0)
else -> return predecessor(v).right.weight
```

We have only just begun hash tables, so some of this might not make sense until next week..

Here we want to compare different approaches to storing data.

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

Imagine a massive array `Data[0..n-1]` of n files, each of which is very big, i.e., at least T bits. Since the files are so big, assume that reading, writing, or hashing a file takes $\Theta(T)$ time. Think of the total size of `Data` (including all the files) as being around 10 terabytes.

You are given a smaller array `S[0..k-1]` containing k files. Each of these files may (or may not) appear in the `Data` array, and may appear in the `Data` array multiple times (i.e., there may be repeats). The goal is to count how many times each of the files in `S` appears in the array `Data`. We want our data structure to be small, ideally not depending on T, which is very large. Even a data structure of size $\Theta(n)$ is too large. And we want our data structure to be fast, since there is so much data to process! Consider the following three strategies for counting:

	Hash Table	**Fingerprint Hash Table**	**Hash Tree**
1.	Create a standard hash table with M slots. Use chaining to resolve collisions.	Create a fingerprint hash table with M slots, where each slot holds a $\log n$-bit counter (instead of just a binary 0/1 value).	Create an empty AVL tree.
2.	Insert each file in S into the hash table, using the file as the key and the count as the value. Initially, the count for each file is 1.	Insert each file into the fingerprint hash table. That is, for each file $S[i]$, set $hash(S[i]) = 1$, i.e., initialize the count to 1.	For each file in $S[i]$, insert $hash(S[i])$ into the AVL tree, using $hash(S[i])$ as the key and the count as the value. Initially, the count for each file is 1.
3.	Scan the *Data* array. For each item $Data[i]$ in the *Data* array, search for $Data[i]$ in the hash table. If you find it, increment the associated count.	Scan the *Data* array. For each item $Data[i]$ in the *Data* array, search for $Data[i]$ in the fingerprint hash table. If the slot $hash(Data[i]) > 0$, then increment the count in that slot.	Scan the *Data* array. For each item $Data[i]$ in the *Data* array, search for $hash(Data[i])$ in the hash tree. If you find it, increment the associated count.
4.	When you are done, subtract 1 from each value in the hash table. The hash table now contains the proper count associated with each file in S.	When you are done, subtract 1 from each value in the fingerprint hash table. The fingerprint hash table now contains the proper count associated with each file in S.	When you are done, subtract 1 from each value in the hash tree. The hash tree now contains the proper count associated with each file in S.

Problem 4.a. What is the problem with the hash table solution? Assume, for now, that the size of the table M is chosen appropriately so that there are at most $O(1)$ collisions per slot. And assume that it is reasonable to scan the entire array `Data`, as there is no other way to determine the count.

The file is used as a key and these file may be very large (in terms of terabytes). This will potentially take up alot of space

Problem 4.b. For the Fingerprint Hashtable, what is a good choice for the value of M?

$\boxed{\text{2k}}$

Explain your answer:

Problem 4.c. What is the total space used by the Fingerprint Hash Table?

$\boxed{\text{2klogn}}$

Problem 4.d. For the FingerPrint Hashtable, is the final count: (circle all that apply)

Possibly too low | [✓ **Always exactly right**] | [✓ **Possibly too high**]

Explain your answer:

Problem 4.e. For the Hash Tree, is the final count: (circle all that apply)

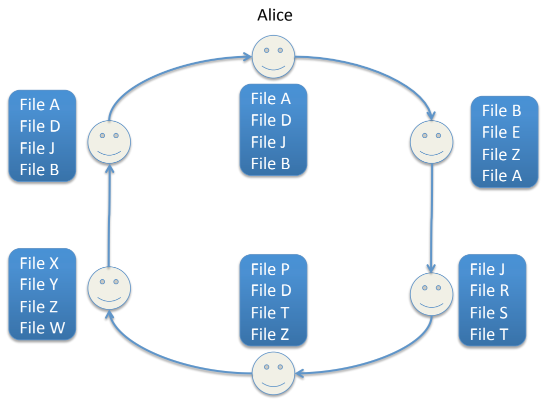
Possibly too low | [✓ **Always exactly right**] | [✓ **Possibly too high**]

Explain your answer:

Problem 4.f. What are the advantages of the Hash-Tree solution, compared to the Fingerprint Hash Table? (You can assume that the hash function used in the Hash-Tree produces a value that can be stored in $O(\log n)$ bits.)

And again… this is about hash tables, and may not make sense until next week.

Problem 5.
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.)



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 is mn. 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 5.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.)
- 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 communication cost of this approach:

O(klogk + klogn)

klog(k5n7) + klog(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:

Problem 5.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 10. For this part, assume that the size of M is reasonable, and that communication costs of $O(M)$ or $O(Mn)$ or even $O(Mnk)$ are reasonable.

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.

Hash table with chaining stores the file as key. This means that each of Alice's k files is stored in the hash table and this leads to high communication costs

Problem 5.c. Next, Alice tries to use a Fingerprint Hash Table. She creates a Fingerprint Hash Table with M entries, but stores a counter of size $\log n$ in each cell (instead of storing just a binary 0/1 value). (Each file can appear at most n times in the network.)

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

nk

Explain your answer briefly:

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

O(Mnlogn)

Explain your answer briefly:

array of size M containing logn bits