

Name: Sample Solution

Email address (UWNetID): \_\_\_\_\_

## **CSE 332 Autumn 2016 Final Exam**

(closed book, closed notes, no calculators)

**Instructions:** Read the directions for each question carefully before answering. We may give partial credit based on the work you **write down**, so show your work! Use only the data structures and algorithms we have discussed in class so far. Writing after time has been called will result in a loss of points on your exam.

**Note:** For questions where you are drawing pictures, please circle your final answer.

You have 1 hour and 50 minutes, work quickly and good luck!

Total: Time: 1 hr and 50 minutes.

<b>Question</b>	<b>Max Points</b>	<b>Score</b>
1	12	
2	12	
3	10	
4	10	
5	14	
6	10	
7	10	
8	11	
9	11	
<b>Total</b>	<b>100</b>	

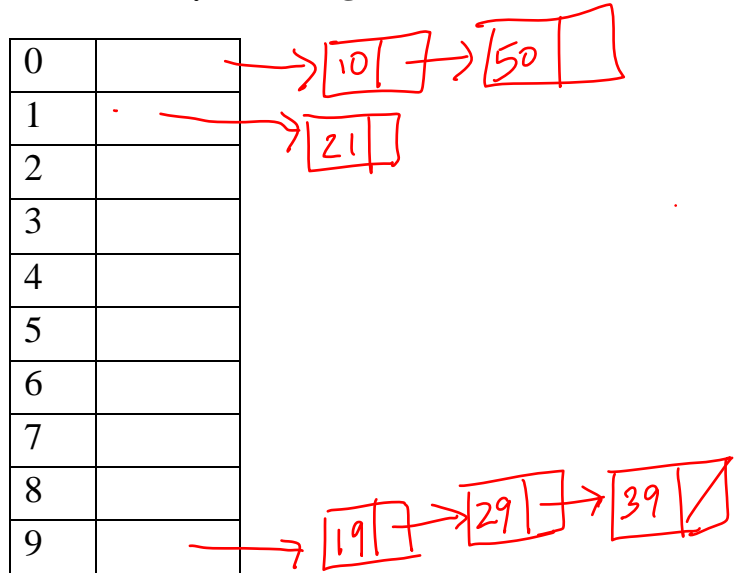
1) [12 points total] Hash Tables

For a) and b) below, insert the following elements in this order: 50, 21, 29, 10, 39, 19. For each table, TableSize = 10, and you should use the primary hash function  $h(k) = k \% 10$ . If an item cannot be inserted into the table, please indicate this and continue inserting the remaining values.

a) Quadratic probing hash table

0	50
1	21
2	
3	39
4	10
5	
6	
7	
8	19
9	29

b) Separate chaining hash table – use a linked list for each bucket where the values are ordered by **increasing value**



c) What is the load factor in Table b)?

0.6

d) In a sentence or two, describe **double hashing**.

The first hash function determines the original location where we should try to place the item. If there is a collision, then the second hash function is used to determine the probing step distance as  $1 * h_2(\text{key})$ ,  $2 * h_2(\text{key})$ ,  $3 * h_2(\text{key})$  etc.

e) What is one advantage of **double hashing** over **quadratic probing**, be specific.

Double hashing avoids secondary clustering that quadratic probing can have. Double hashing does not suffer from the case where quadratic probing may not find a viable bucket when the table is more than half full.

f) What is the big-O worst case runtime of a *find operation* on a table like table b)?  $O(N)$

g) What is the big-O worst case runtime of an *Insert* in a *separate chaining hash table* containing  $N$  elements where each bucket points to an AVL tree?  $O(\log N)$

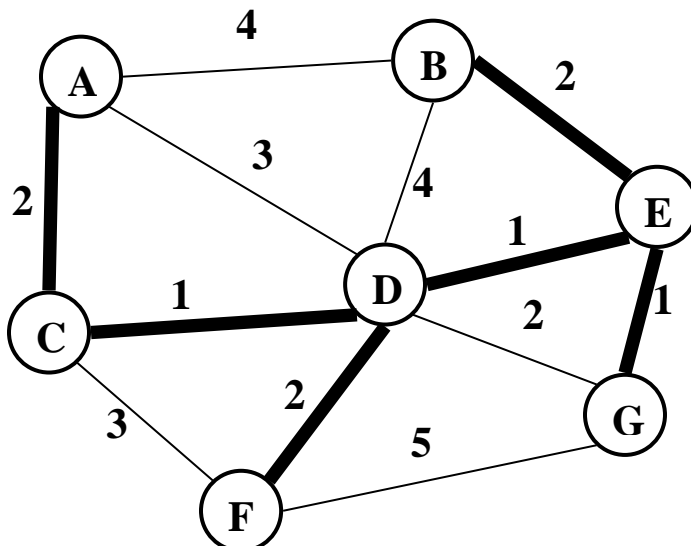
2) [12 points total] Graphs!

- a) [2 points] What is the big-O running time of Dijkstra's algorithm (assuming an adjacency list representation) if:
- (i) A priority queue is used?  **$O(E \log V + V \log V)$  or just  $O(E \log V)$**
  - (ii) An unsorted list is used?  **$O(EV + V^2)$**

- b) [2 points] Which implementation of Dijkstra's (priority queue vs. unsorted list) is likely to be faster if the graph is known to be dense? **Explain your answer in ~one sentence for any credit.**

**Priority Queue is better than unsorted list regardless of whether the graph is dense or sparse.  $O(\log V)$  is always better than  $O(V)$  whether  $E$  is close to  $V$  or  $V^2$ .**

- c) [2 points] Give a Minimum Spanning Tree (MST) of the graph below by highlighting the edges that would be part of the MST.



- d) [6 points] For (ii) and (iv) below, *you are given a perfect binary tree of height  $h$  containing  $n$  nodes*. Your answer should be an **exact formula**, such as  $3/2 \log h$ , or  $5 \times 3^n$ , not big-O notation.

- (i) Depth First Search: What is the name of the data structure used in DFS? **stack**

- (ii) What is the maximum size of that data structure during a DFS ?

**$h + 1$**

- (iii) Breadth First Search: What is the name of the data structure used in BFS? **queue**

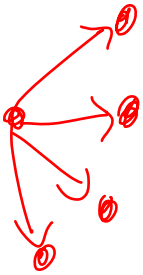
- (iv) What is the maximum size of that data structure during a BFS ?

**$2^h$**

3) [10 points total] More Graphs!

a) [4 points] **Draw a picture** of a connected directed graph with 5 nodes that has the largest possible number of topological sorts. How many different topological sorts does it have?

**Either 4 nodes all pointing to one or one node pointing to 4 nodes**



OR



How Many topo sorts? **4! Or 24 sorts**

b) [6 points total] Given a weighted, undirected graph with  $|V|$  nodes, answer the following. Assume all weights are non-negative.

- (i) [2 points] If each edge has weight  $\leq w$ , what can you say about the cost of an MST? Your answer should give a lower bound, or an upper bound on the cost of the MST, e.g. “the cost of the MST is  $\geq 2^{w+n}$ ”, or “the cost of the MST is  $\leq \log(w \log n)$ ”.

**The cost of the MST is  $\leq (|V| - 1)w$  (since there are  $|V| - 1$  edges in an MST, and each can have weight  $\leq w$ )**

- (ii) [2 points] If the cost of an MST is  $c$ , what can you say about the shortest distances returned by Dijkstra’s algorithm when run with an arbitrary vertex  $s$  as the source? You should give a lower bound or an upper bound for the distance between arbitrary vertices  $u$  and  $v$ .

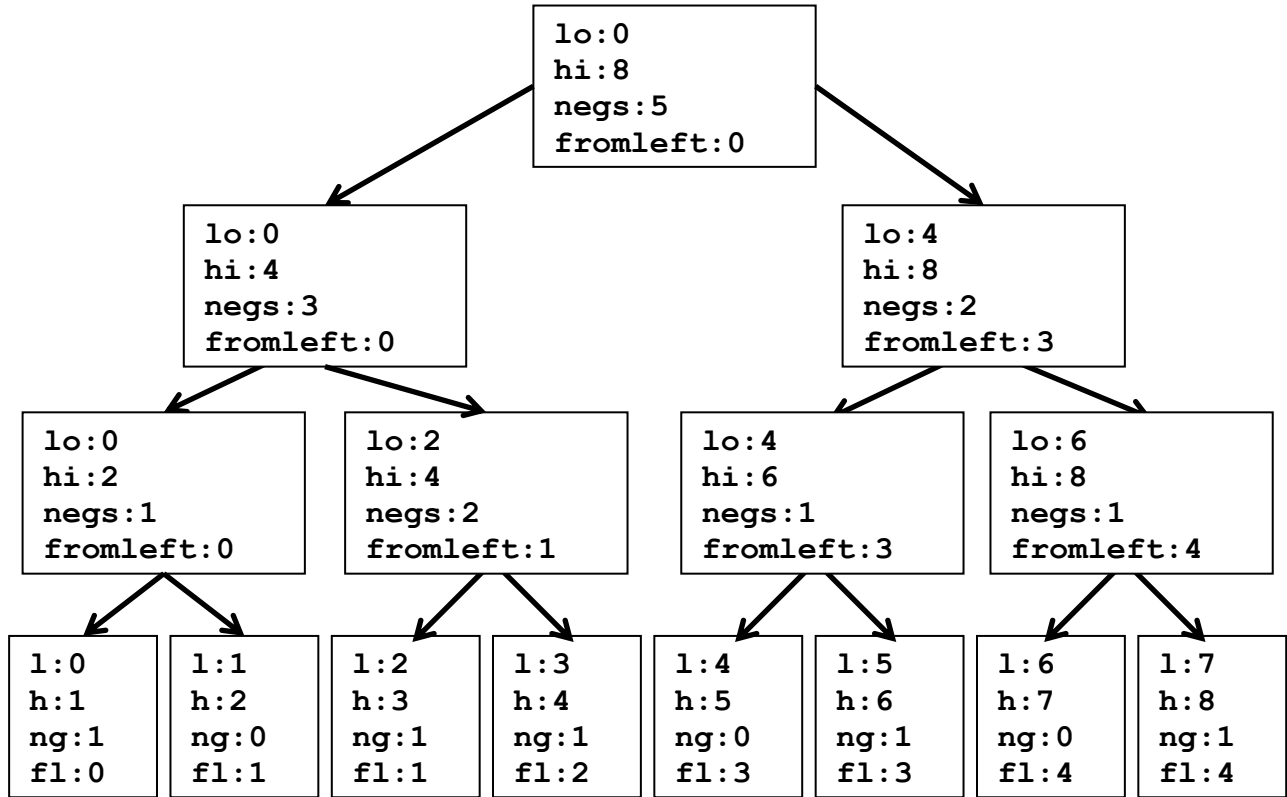
**The shortest distance between any two points  $\leq c$ , since any two points are connected in the MST (which is undirected)**

- (iii) [2 points] If there exists a Hamiltonian circuit of cost  $c$ , then what can you say about the cost of the minimum spanning tree ?

**The cost of the MST  $\leq c$ , since we can remove one edge (which might have weight 0) from the Hamiltonian circuit to get a (not necessarily minimum) spanning tree.**

4) [10 points] **Parallel Prefix CountNegatives:**

- a) Given the following array as input, perform the parallel prefix algorithm to fill the **output** array with the **number of negative values contained in all of the cells to the left** (including the value contained in that cell) in the input array. Fill in the values for **negs**, and **fromLeft** in the tree below. Do not use a sequential cutoff. **Note:** This is NOT sum!



Index	0	1	2	3	4	5	6	7
<b>Input</b>	<b>-3</b>	<b>5</b>	<b>-4</b>	<b>-7</b>	<b>2</b>	<b>-8</b>	<b>9</b>	<b>-5</b>
<b>Output</b>	<b>1</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>3</b>	<b>4</b>	<b>4</b>	<b>5</b>

- b) How is the **fromLeft** value computed for a node in the tree? Specifically, if you have a node with **negs** & **fromLeft** computed, how do you compute **fromLeft** for its left & right children (both of which have **negs** already computed).

Left child's fromLeft:

**Parent's fromLeft (or fromLeft, if viewed from parent)**

Right child's fromLeft:

**Parent's fromLeft + sibling's neg (or fromLeft + left.negs, if viewed from parent)**

5) [14 points] In Java using the ForkJoin Framework, write code to solve the following problem:

- **Input:** An int k, and an array of ints of size n containing values in the range 0...k-1
- **Output:** the value (between 0 and k-1) occurring most often in the Input array. If there is a tie for the most frequently occurring value, return the *smallest* number.

For example, if k=7 and input array of size 9 is {0, 2, 5, 2, 6, 5, 4, 5, 2}, the output would be 2.

- Do **not** employ a sequential cut-off: **the base case should process one element.** (You can assume the input array will contain at least one element.)
- Assume k is small (e.g. less than 50)
- Fill in the function `findMostCommon` **below.**
- Give a class definition, `FindComTask`, **on the next page.**

**You may not use any global data structures or synchronization primitives (locks).**

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

class Main{
    static final ForkJoinPool fjPool = new ForkJoinPool();

    // Returns the most common value in the array input.
    // Values in input are in the range 0 to k.
    int static findMostCommon (int k, int[] input) {

        int[] counts =
            fjPool.invoke(new FindComTask(k, input, 0,
            input.length));

        int maxCount = 0;
        int maxCountIndex = 0;
        for (int i = 0; i < k; i++){
            if (counts[i] > maxCount) {
                maxCount = counts[i];
                maxCountIndex = i;
            }
        }
        return maxCountIndex;
    }

}
```

**Please fill in the function above and write your class on the next page.**

5) (Continued) Write your class on this page.

```
public static class FindComTask extends RecursiveTask<int[]> {
    int k;
    int[] input;
    int lo;
    int hi;
    public FindComTask(int k, int[] input, int lo, int hi) {
        this.k = k;
        this.input = input;
        this.lo = lo;
        this.hi = hi;
    }

    public int[] compute() {
        if (hi - lo <= 1) {
            int[] counts = new int[k];
            counts[input[lo]] = 1;
            return counts;
        } else {
            int mid = lo + (hi - lo)/2;

            FindComTask left = new FindComTask(k, input, lo, mid);
            FindComTask right = new FindComTask(k, input, mid, hi);
            right.fork();
            int[] leftResult = left.compute();
            int[] rightResult = right.join();

            for (int i = 0; i < k; i++) {
                leftResult[i] += rightResult[i];
            }
            return leftResult;
        }
    }
}
```

6) [10 points] **Concurrency:** Once again, we are helping out with the Aviation Management System. This time they have a data structure that keeps track of whether there is a flight from one airport to another. If there is a flight from airport x to airport y, then we should be able to assume that there is also a flight from airport y to airport x. This information is being kept in an array (indexed by unique airport number), where each location in the array points to a list of the airport numbers that can be reached from this airport by a single flight (an **AirportList**). We would like to allow as much concurrent access to this data structure as possible, while assuring that each thread always sees a consistent state of the data structure. We attempt this by having a different lock on each of the **AirportLists**.

a) Our first attempt at the **removeRoute** method is below. Assume **hasFlightTo** and **removeFlight** are methods on an **AirportList**. **removeFlight** will throw an exception if the specified flight is not present.

```
void removeRoute1(int x, int y, AirportList[] airports) {
    synchronized(airports[x]) {
        synchronized(airports[y]) {
            if(airports[x].hasFlightTo(y)) {
                airports[x].removeFlight(y);
                airports[y].removeFlight(x);
            }
        }
    }
}
```

i. Does the code above have (circle all that apply):

potential for deadlock, a data race, a race condition, none of these

ii. If possible, show (as done in class) an interleaving of two or more threads calling **removeRoute1** that demonstrates a concurrency error. If not possible, explain why not.

**Thread 1:**  
**removeRoute1(a, b, airports) {**  
     **synchronized(airports[a]) {**  
         **(waiting on airports[b])**  
     **}**  
**}**

**Thread 2:**  
**removeRoute1(b, a, airports) {**  
     **synchronized(airports[b]) {**  
         **(waiting on airports[a])**  
     **}**  
**}**



## 6) (Continued)

b) Our second attempt at the `removeRoute` method is below.

```
void removeRoute2(int x, int y, AirportList[] airports) {
    synchronized(airports[x]) {
        if (!(airports[x].hasFlightTo(y))) {
            return;
        }
    }
    synchronized(airports[x]) {
        airports[x].removeFlight(y);
    }
    synchronized(airports[y]) {
        airports[y].removeFlight(x);
    }
}
```

i. Does the code above have (circle all that apply):

potential for deadlock, a data race, **a race condition**, none of these

ii. If possible, show (as done in class) an interleaving of two or more threads calling `removeRoute2` that demonstrates a concurrency error. If not possible, explain why not.

```
Thread 1:                                Thread 2:
removeRoute2(a, b, airports) {
    synchronized(airports[a]) {
        if (!(airports[a].hasFlightTo(b))) {
            return;
        }
    }

    synchronized(airports[a]) {
        airports[a].removeFlight(b);
        // THROWS AN EXCEPTION
    }
}
```

```
removeRoute2(b, a, airports) {
    // Entire method runs, removing
    // flight from b->a and a->b
}
```

c) Finally, the developers' consider scrapping the one-lock-per `AirportList` strategy in favor of a single lock, locking the entire array of `AirportLists`.

One benefit of a single-lock locking all airports:

**Less risk of programmer creating a concurrency error, easier to implement.**

One drawback of a single-lock locking all airports:

**Less concurrency (no concurrency!)**

## 7) [10 points] Sorting

You are given a list of AVL trees. The keys in the AVL trees are ages of people. Each AVL tree represents the ages for people in a different community. Your task is to sort the AVL trees such that tree X comes before tree Y if and only if:

- The minimal age in tree X is less than the minimal age in tree Y, **or**
- The minimal ages are the same, but the maximal age in tree X is less than the maximal age in tree Y

Otherwise, ties are broken arbitrarily. You may assume that:

- There are  $k$  trees
- Each tree has  $n$  keys in it
- The range of ages is fixed (0-127)

a) [5 points] Describe in a few sentences or numbered steps how you could use Mergesort to sort these trees efficiently in the worst case. What is the running time in terms of  $k$  and  $n$ ?

1. **Find the minimum and maximum age for each tree. Time  $O(k \log n)$ , since each tree is balanced.**
2. **Perform a mergesort where the comparison is based on using the min element for each tree, with the max element as a tie breaker. Time is  $O(k \log k)$ .**

**Total worst case time  $O(k \log n + k \log k)$**

**Note: Another way of doing this was to just find the min and max values when doing the comparison in mergesort which would be  $O(k \lg k * \lg n)$ .**

Running Time: <b><math>O(k \log n + k \log k)</math></b>
--

b) [5 points] Describe in a few sentences or numbered steps how you could use ideas from Radixsort to sort these trees efficiently in the worst case. What is the running time in terms of  $k$  and  $n$ ?

1. **Find the minimum and maximum as above. Worst case  $O(k \log n)$ .**
2. **Perform a BinSort using the maximum age of each tree (It is critical that you sort on the least significant – the part I care about least, first!). Time  $O(k+128) = O(k)$**
3. **BinSort on the minimum age of each tree. Time  $O(k)$ .**

**Total time  $O(k + k \log n) = O(k \log n)$ .**

Running Time: <b><math>O(k \log n)</math></b>
---

8) [11 points] More Sorting

- a) [3 points] Give the recurrence for Mergesort (parallel sort & sequential merge) – best case span: (Note: We are NOT asking for the closed form.)

**Answer:  $T(N) = T(n/2) + O(n)$**

- b) [3 points] In the \_\_\_\_\_ spaces below, order these sorts from slowest to fastest in terms of big-O runtimes. For parallel sorts, use the span. **Draw a circle around any sorts whose big-O runtimes are the same.** You do not have to give the runtimes, just list the sort letters.

A) Mergesort (sequential) – worst case

B) Quicksort (parallel sort & parallel partition) – best case span  $O(\log^2 n)$

C) Quicksort (sequential) – best case

D) Quicksort (sequential) – worst case  $O(n^2)$

E) Quicksort (parallel sort & parallel partition) – worst case span

\_\_\_\_  $D(n^2)$  \_\_\_\_  $ACE$  are ties at  $(n \log n)$  \_\_\_\_  $B(\log^2 n)$  \_\_\_\_  
Slowest Fastest

- c) [2 points] Suppose we choose the median of five items as the pivot in quicksort. If we have an  $N$  element array, then we find the median of the elements located at the following positions: left (= 0), right (=  $N - 1$ ), center (the average of left and right, rounded down), leftOfCenter (the average of left and center, rounded down), and rightOfCenter (the average of right and center, rounded down). The median of these elements is the pivot.

What is the worst case running time of *this version* of quicksort?

$O(N^2)$

- d) [1 point] Any algorithm for sorting must take  $\Omega(N \log N)$  time in the worst-case.

TRUE FALSE

- e) [2 points] What does it mean for a sort to be stable?

**In case of ties during the sort, the original ordering of values is preserved.**

9) [11 points] P, NP, NP-Complete

a) [2 points] “NP” stands for \_\_\_\_\_ Non-deterministic polynomial \_\_\_\_\_

b) [2 points] What does it mean for a problem to be in NP-complete?

**These problems are in NP (Given a candidate solution, we can verify whether the solution is correct in polynomial time.) and they are the “hardest” problems in NP. NP-complete problems can be reduced to other NP-complete problems in poly-time. We are pretty sure these problems do not have polynomial time solutions. If any one NP-complete problem could be solved in polynomial time, then all NP-complete problems could be solved in polynomial time.**

c) [5 points] For the following problems, circle ALL the sets they belong to:

Determining if a chess move is the best move on an N x N board	NP	P	NP-complete	<u>None of these</u>
Finding the maximum value in an array	<u>NP</u>	<u>P</u>	NP-complete	None of these
Finding a cycle that visits each vertex in a graph exactly once	<u>NP</u>	P	<u>NP-complete</u>	None of these
Finding a cycle that visits each edge in a graph exactly once	<u>NP</u>	<u>P</u>	NP-complete	None of these
Determining if a program will ever stop running	NP	P	NP-complete	<u>None of these</u>

d) [1 point] If there exists a polynomial time algorithm to solve **Euler Circuit**, then there exists a polynomial time algorithm to solve **SAT**.

TRUE      FALSE

e) [1 point] If there exists a polynomial time algorithm to solve **Hamiltonian Circuit** then any problem in **NP** can be solved by some polynomial time algorithm.

TRUE      FALSE