Name: _____**Sample Solution**_____

Email address (UWNetID): _____

CSE 332 Winter 2019 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.

Question	Max Points	Topic
1	10	Hashing
2	10	Graphs
3	8	More Graphs
4	10	|| Prefix
5	14	ForkJoin
6	12	Concurrency
7	16	Sorting
8	10	P/NP
Total	90	

1) [10 points total] Hash Tables

For a) and b) below, insert the following elements in this order: 7, 9, 48, 8, 37, 57. 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) [1 pt] Quadratic probing hash table

0	
1	37_2
2	8_2
3	
4	
5	
6	57_3
7	7
8	48
9	9

57_2

37_0 57_0
37_1 57_1
8_0
8_1

b) **[1 pt] Separate chaining hash table** – Use a linked list for each bucket. Order elements within buckets in any way you wish.

0	
1	
2	
3	
4	
5	
6	
7	→ 57 → 37 → 7
8	→ 8 → 48
9	→ 9

c) **[2 pts]** 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*h2(key), 2*h2(key), 3*h2(key) etc. away from the original location.

d) **[4 pts]** List 2 cons of **quadratic probing** and describe how one of those is fixed by using **double hashing**.

In quadratic probing, 1) if the table is more than half full (load factor = 0.5) then you are not guaranteed to be able to find a location to place the item, 2) suffers from secondary clustering (items that initially hash to the same location resolve the collision identically).

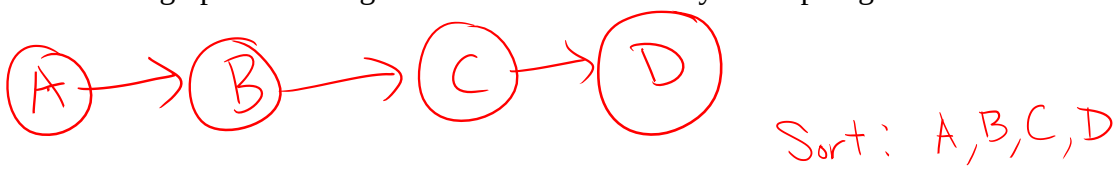
Assuming a good second hash function is used, double hashing does not suffer from 1). Assuming a good second hash function is used, double hashing avoids secondary clustering because items that initially hash to the same location resolve the collision differently, which decreases the likelihood that two elements will hash to the same index after initial collision.

e) **[2 pts]** Give a tight big-O bound for the worst case runtime of an *Insert in a **separate chaining hash table** containing N elements where each bucket points to its own binary search tree?* **For any credit explain your answer briefly.**

O(N) – A bad hash function causes all N elements to hash to the same bucket, and then those keys are inserted in either ascending or descending order – leading to the worst case running time for each insert into that one binary search tree.

2) [10 points total] Graphs!

a) **[1 pts]** Draw a directed graph containing 4 vertices that has exactly one topological sort.

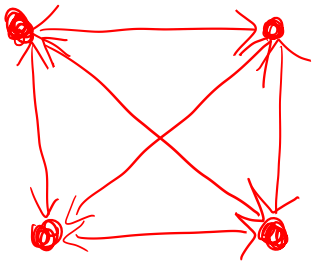


b) [3 pts] You are given Dijkstra's algorithm implemented using a priority queue as described in lecture. The priority queue implementation you must use has the following worst case running times: insert: O(N), deletemin: O(N), decreasekey: $O(N^2)$, increasekey: $O(N^2)$, buildheap: O(NlogN). Given that you must use this priority queue, what is the worst case running time of Dijkstra's? Give your answer as a tight Big-O bound in terms of V and E. **Explain how you got your answer briefly.**

```
for each node: x.cost=infinity, x.known=false          // O(V)
start.cost = 0                                         // O(1)
build-heap with all nodes                              // O(VlogV) buildheap
while (heap is not empty) {                            // V times
  b = deleteMin()                                            // O(V) deletemin
  b.known = true                                             // O(1)
  for each edge (b,a) in G                                   // d times, E times total
    if(!a.known) if(b.cost + weight((b,a)) < a.cost){              // O(1)
      decreaseKey(a,"new cost - old cost")                         // O(V^2) decreasekey
      a.path = b                                                   // O(1)
    }
}          O(V + VlogV + V*(V + d * V^2))   -> O(VlogV + V^2 + E* V^2) -> O(E* V^2)
```

c) [2 pts] What is the minimum number of edges in a complete directed graph with 4 vertices? Do NOT include self-loops in your count. **Draw a picture of your graph.**

Minimum Number of Edges:
12



d) **[2 pt]** We covered two data structures to represent graphs. Give the name of the one with a faster worst case running time for the operation of deleting a particular edge and its running time.

Name of structure: _____**Adjacency Matrix**_____

Worst Case tight Big-O bound on Running Time: _____**O(1)** _____

e) **[1 pt]** Give one way in which Breadth First Search is better than Depth First Search.

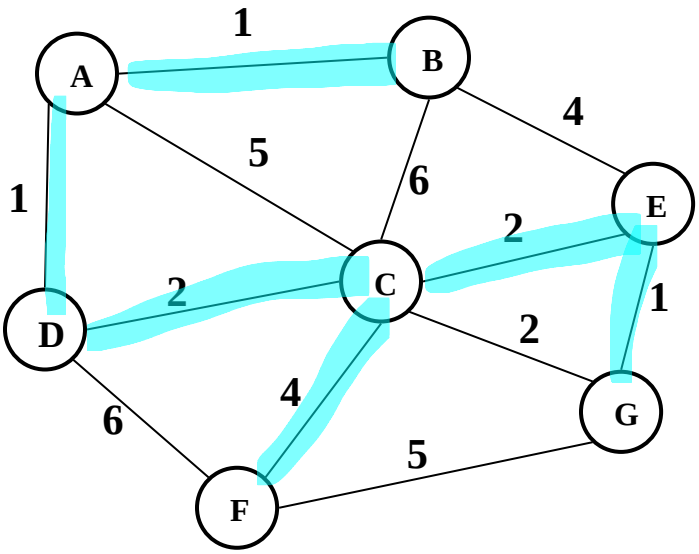
Breadth First Search will find the shortest (unweighted) path to a vertex, Depth First Search does not necessarily find that.

f) **[1 pt]** Give one way in which Depth First Search is better than Breadth First Search.

Depth First Search often uses less memory than Breadth First Search.

3) [8 points total] More Graphs!

a) [6 pts] Find a minimum spanning tree with Prim's algorithm **using vertex A as the starting node. ****Mark, circle, or highlight edges in the graph that are in your minimum spanning tree. ** Show your steps in the table by crossing through values that are replaced by a new value. Break ties by choosing the letter that comes first alphabetically; ex. if Y and Z were tied, you should pick Y. *Note that b) asks you to recall what order vertices were declared known.*



Vertex	Known (F/T)	Cost	Prev
A	~~F~~ T	0	—
B	~~F~~ T	1	A
C	~~F~~ T	~~5~~, 2	~~A~~, D
D	~~F~~ T	1	A
E	~~F~~ T	~~4~~, 2	~~B~~, C
F	~~F~~ T	~~6~~, 4	~~D~~, C
G	~~F~~ T	~~2~~, 1	~~C~~, E

b) [1 pt] In what order would Prim's algorithm mark each node as *known*?

A, B, D, C, E, G, F

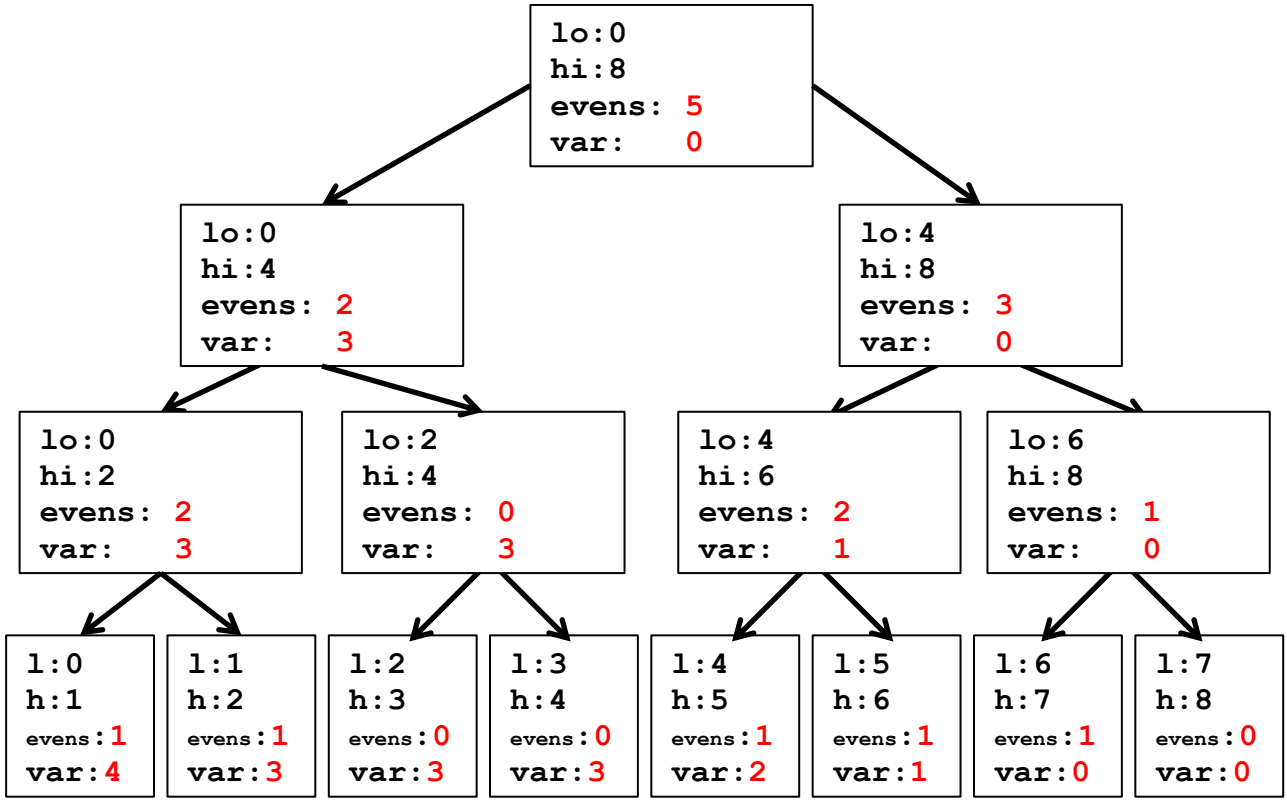
c) **[1 pt]** Will Prim's starting at vertex A find a correct minimum spanning tree if the weight of edge (E, G) is set to be -8? (circle one)

YES **NO**

Did you highlight your original MST in the graph above?

4) [10 points] Parallel "Suffix Count Evens" (Like Prefix, but <u>from the Right</u> instead):

a) Given the following array as input, calculate the "suffix count of evens" using an algorithm similar to the parallel prefix algorithm discussed in lecture. In other words, `output[i]` should contain the count of even numbers from `input[i]` to `input[input.length - 1]` inclusive. The first pass of the algorithm is similar to the first pass of the parallel prefix code you have seen before. Fill in the values for **`evens`** and **`var`** in the tree below. The **`output`** array has been filled in for you. Do not use a sequential cutoff.



Index	0	1	2	3	4	5	6	7
`input`	0	-6	7	25	14	14	4	5
`output`	5	4	3	3	3	2	1	0

b) Give formulas for the following values where **`p`** is a reference to a non-leaf tree node and **`leaves[i]`** refers to the leaf node in the tree visible just above the corresponding location in the **`input`** and **`output`** arrays in the picture above.

```
p.evens = p.left.evens + p.right.evens

p.left.var = p.right.evens + p.var

p.right. var = p.var

output[i] = leaves[i].evens + leaves[i].var
```

c) Describe how you assigned a value to **`leaves[i].evens`** .

```
leaves[i].evens = 1 if input[i] % 2 == 0 else 0
```

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

• **Input**: An array of `ints` containing values in the range 0-15 (inclusive). There can be duplicates and assume that the length of the array is an odd number.
• **Output**: Return the median value found in the array.

If the input array is {0}, the program would return 0.
If the input array is {15, 3, 6}, the program would return 6.
If the input array is {13, 12, 0, 5, 3, 12, 15, 2, 7}, the program would return 7.

- 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 `int`.)
- Give a class definition, `FindMedianTask`, **along with any other code or classes needed**.
- Fill in the function `findMedian` **below.**

***You may **NOT** use any **global data structures** or **synchronization primitives (locks).**
***Make sure your code has O(log n) span and O(n) work.**

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

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

      // Returns the median value in input array with values 0-15.
      // Assumes an odd number of values in input array.
      public static int findMedian (int[] input) {

         // Your code here:

         int[] res = fj.invoke(new FindMedianTask(0, input.length, input));

         int mid = input.length/2 + 1;
         int count = 0;

         for(int i = 0; i < 16; i++) {
            count += res[i];
            if (count >= mid) {
               return i;
            }
         }
         return -1; // This should never happen









      }
}
```

Please fill in the function above and write your class(es) on the next page.

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

```
public static class FindMedianTask extends RecursiveTask<int[]> {
        int lo, hi;
        int[] input;

        public FindMedianTask(int lo, int hi, int[] input) {
            this.lo = lo;
            this.hi = hi;
            this.input = input;
        }

        public int[] compute() {

            if (hi - lo <= 1) {
                int[] counts = new int[16];
                counts[input[lo]] = 1;
                return counts;
            }

            int mid = lo + (hi - lo)/2;

            FindMedianTask left = new FindMedianTask(lo, mid, input);
            FindMedianTask right = new FindMedianTask(mid, hi, input);

            right.fork();
            int[] leftResult = left.compute();
            int[] rightResult = right.join();

            for (int i = 0; i < 16; i++) {
                leftResult[i] += rightResult[i];
            }
            return leftResult;
        }
    }
}
```

6) [12 points total] Concurrency: The `PhoneMonitor` class tries to help manage how much you use your cell phone each day. Multiple threads can access the same `PhoneMonitor` object. Remember that synchronized gives you reentrancy.

```
1
2  public class PhoneMonitor {
3     private int numMinutes = 0;
4     private int numAccesses = 0;
5     private int maxMinutes = 200;
6     private int maxAccesses = 10;
7     private boolean phoneOn = true;
8     private Object accessesLock = new Object();
9     private Object minutesLock = new Object();
10
11    public void accessPhone(int minutes) {
12
13       if (phoneOn) {                    // for part c REMOVE this line
14
15          synchronized (accessesLock) {
16
17             synchronized (minutesLock) {
18
19                if (phoneOn) {           // for part c ADD this line
20
21                   numAccesses++;
22                   numMinutes += minutes;
23                   checkLimits();
24                }
25             }
26          }
27    }
28
29    private void checkLimits() {
30
31       synchronized (minutesLock) {     // for part c swap line 31 & 33
32
33          synchronized (accessesLock) {
34
35             if ( (numAccesses >= maxAccesses) ||
36                  (numMinutes >= maxMinutes) ) {
37                      phoneOn = false;
38                }
39             }
40          }
41    }
42 }
```

a) **[4 pts]** Does the `PhoneMonitor` class as shown above have (circle **all** that apply):

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

Justify your answer. Refer to line numbers in your explanation. Be specific!

There is a data race on `phoneOn`. Thread 1 (not needing to hold any locks) could be at line 13 reading `phoneOn`, while Thread 2 is at line 35 (holding both of the locks) writing `phoneOn`. A data race is by definition a type of race condition.

6) (**Continued**)

b) **[4 pts]** Suppose we made the `checkLimits` method **`public`**, and changed nothing else in the code. Does this modified `PhoneMonitor` class have (circle **all** that apply):

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

If there are any FIXED problems, describe why they are FIXED. If there are any NEW problems, give an example of when those problems could occur. Refer to line numbers in your explanation. Be specific!

The same data race still exists, and thus so does the race condition.

By making `checkLimits` method `public`, it is possible for Thread 1 to call `accessPhone` and be at line 17 holding the `accessesLock` lock and trying to get the `minutesLock` lock. Thread 2 could now call `checkLimits` and be at line 31, holding the `minutesLock` lock and trying to get the `accessesLock` lock

c) **[4 pts]** Assuming the `checkLimits` method is now **`public`,** add these two methods to the class:

```
public int increaseMaxMinutes(int minutes) {

  synchronized(minutesLock) {  // for part c add this
    maxMinutes += minutes;

    return maxMinutes;
  }
}

public int increaseMaxAccesses(int accesses) {

  synchronized(accessesLock) {  // for part c add this
    maxAccesses += accesses;

    return maxAccesses;
  }
}
```

Now modify the **code above and on the previous page** to ***allow the most concurrent access*** and to avoid all of the potential concurrency problems listed above. Use only `synchronized` statements or methods. Create other objects or fields as needed. **For full credit you must allow the most concurrent access possible without introducing any of the synchronization problems listed above.**

7) [16 points total] Sorting

a) **[3 pts]** Give the ***recurrence*** for Quicksort (parallel sort & sequential partition) – best case span. (Note: We are NOT asking for the closed form.) **For any credit, explain all parts of your answer briefly**.

$T(N) = T\left(\frac{N}{2}\right) + O(N)$

- In the best case Quicksort partitions the input into two equal sized pieces.
- The two calls to Quicksort happen in parallel, so the coefficient is 1.

The partition step requires that all N elements be examined.

b) **[3 pts]** Give the ***recurrence*** for SEQUENTIAL Mergesort – worst case running time. (Note: We are NOT asking for the closed form.) **For any credit, explain all parts of your answer briefly**.

$T(N) = 2 \cdot T\left(\frac{N}{2}\right) + O(N)$

- Mergesort always divides the input into two equal sized pieces.
- Mergesort is called twice, once on each of the two pieces

After returning from the two calls to mergesort, the two sorted pieces must be merged. This requires examining O(N) elements.

c) **[2 pts]** Give a tight Big-O bound for the running time of **selection sort** if it *happened to be given an array that was already sorted from smallest to largest.* For any credit explain your answer.

Running time:
$O(N^2)$

Selection sort will have N iterations of its outer loop. For iteration k, its inner loop examines all the remaining not-yet-sorted elements and finds the smallest one, putting it in position k. For sorted data, even though for iteration k, the kth-smallest element is already in the correct position, the sort does not know this. Thus it must still must *examine* all N-k not-yet-sorted elements even though it does not have to move anything. The number of elements examined is still: N, N-1, N-2, N-3, …. 1 or (N*(N+1))/2 or $O(N^2)$ whether the data is pre-sorted or not.

7) (Continued)

d) **[2 pts]** Give a tight Big-O bound for the running time of **mergesort** if it *happened to be given an array that was already sorted from smallest to largest.* For any credit explain your answer.

Running time:

O(NlogN)

Sequential Mergesort as described in lecture:
1) Divides the given array in half with math (size/2) in constant time.
2) Makes 2 recursive calls to Mergesort on those two equal sized pieces.
3) Merges those two sorted pieces together and copies them into a new array.

Steps 1 and 2 are not affected by the elements being sorted already. Step 3 is the only place it could potentially make a difference. However even if the two pieces could just be concatenated together without any copying, the merge step would still examine the entire first piece (of size N/2) to determine it was sorted. Thus our recurrence would still be T(N) = 2T(N/2) + O(N) -> O(NlogN)

e) **[3 pts]** Is bucket sort in-place? YES **NO**

Explain your answer. Be specific – **refer to the bucket sort algorithm in particular**, do not just give a definition of in-place.

Bucket sort uses a set of "buckets" of size k as an auxiliary data structure. Bucket sort reads through the input and either increments a count of each element or adds the item to a linked list (see Movie Ratings example in slides) attached to each bucket. In the final step it reads through the set of k buckets in order, outputting the N elements in sorted order. Since bucket sort uses a non-trivial auxiliary data structure we consider it NOT to be in-place.

f) **[3 pts]** Is radix sort stable? **YES** NO

Explain your answer. Be specific – **refer to the radix sort algorithm in particular**, do not just give a definition of stable.

In the case of ties, a stable sort preserves the original ordering of values.

Radix sort does multiple passes of bucket sort. The first pass is done on the least-significant digit, sorting by a digit per pass until it does the last pass on the most-significant digit. In order to preserve the results of the previous pass, each pass MUST be stable.
E.g. if the first pass sorted one the ones place, the second pass must preserve that ordering while also sorting on the 10s place, thus after pass k, the data has been sorted if you only look at the least-significant k digits. Each pass can be made stable by having each bucket maintain the original order of the values put into it (e.g. each bucket is a FIFO queue).

8) **[10 points total] P, NP, NP-Complete**

a) **[1 pt]** "NP" stands for ____**Non-deterministic Polynomial**_____

b) **[2 pts]** What does it mean for a problem to be in NP?

Given a candidate solution, we can verify whether the solution is correct in polynomial time.

c) **[2 pts]** What should you do if you **suspect** (but are not sure) a problem you are given is NP-complete, and you need an answer to the problem in a reasonable amount of time?

First try to establish that it is NP-complete. Part of this includes doing a reduction from a known NP-complete problem to your problem in polynomial time. (Another part includes showing that your problem is in NP, although that alone is not sufficient.)

AFTER you have shown that your problem is NP-complete, you can quit trying to find a polynomial time algorithm and instead use any of the techniques we use for NP-complete problems (e.g. approximation algorithms, solutions for a restricted version of the problem, heuristics).

d) **[5 pts]** For the following problems, circle **ALL** the sets each problem belongs to:

Finding a cycle that visits each vertex in a graph exactly once	**NP-complete**	P	**NP**	None of these
Determining if an array is sorted.	NP-complete	**P**	**NP**	None of these
Finding the shortest path from one vertex to all other vertices in the graph. The graph is directed and weighted.	NP-complete	**P**	**NP**	None of these
Finding a path in a weighted graph that begins and ends at the same vertex, and visits every vertex exactly once. The path must have a total cost < k.	**NP-complete**	P	**NP**	None of these
Determining if a chess move is the best move on an N x N board	NP-complete	P	NP	**None of these**