CSE 332 : 21AU Final Solutions

Name:	**NetID:** @uw.edu

Instructions

- The allotted time is **110** minutes. Please do not turn the page until the staff says so.
- This is a closed-book and closed-notes exam. You are not permitted to access electronic devices.
- Read directions carefully, especially for problems that require you to show work or provide an explanation.
- We can only give partial credit for work that you've written down.
- Unless otherwise noted, every time we ask for an O, Ω, or Θ bound, it must be simplified and tight.
- For answers that involve bubbling ◯ or ☐, make sure to fill in the shape completely: ● or ■.
- If you run out of room on a page, indicate that the answer continues on the back of that page. Try to avoid writing on the very edges of the pages: we scan your exams and edges often get cropped off.
- Make sure you also get a copy of the formula sheet.

Advice

- If you feel like you're stuck on a problem, you may want to skip it and come back at the end if you have time.
- Look at the question titles on the cover page to see if you want to start somewhere other than problem 1.
- The TAs agree that problem 8 is the hardest. We intentionally put it at the end for that reason.
- Remember to take deep breaths.
- Every tree is a forest. You can do this.

Question	**Max points**
1. First-Half Facts	16
2. Graphs (Short Answers)	14
3. Sorting (Short Answers)	15
4. Concurrency	15
5. Finish the ForkJoin	13
6. Graph Modeling	10
7. P/NP	8
8. Better Parallel Quick Sort	9
9. Drawing	1
Total	**101**

1. First-Half Facts [16 points]

(a) Suppose you have a B-tree with parameters $M = 4$ and $L = 6$.
What is the **BEST CASE** number of disk accesses needed to perform a `find` in the B-tree if its height is h.
Give an exact formula (not a big-O description). **Solution:**

$h + 1$

(b) Given a min-heap in array form, what is the tightest **WORST CASE** runtime to transform the min-heap into a max-heap? **Solution:**

$O(n)$

(c) Consider a hashtable that uses quadratic probing as its collision resolution strategy and currently has a load factor of $99/100$. What is the **WORST CASE** runtime of an `insert` into this hashtable? **Solution:**

There is no upper bound on this time (it is not guaranteed to work). We accepted answers like ∞, $O(\infty)$.

(d) Consider a hashtable that uses separate chaining as its collision resolution strategy and has a maximum load factor of 200. What is the **AVERAGE CASE** runtime of an insertion into this hashtable? **Solution:**

$O(1)$

(e) Suppose you have an AVL tree where for every non-leaf node, the node's right subtree has a height one more than its left subtree. You `insert` a new key into this tree that is larger than all others. Give a tight, simplified big-O bound on the number of rotations required for this insertion. **Solution:**

$O(1)$, you will need to do one as you've imbalanced the node that was previously just above the leaf-level. But once you'd done this rebalancing you're done (remember you only ever need to rotate at one location).

(f) What is the **BEST CASE** runtime for an insertion into a binary tree, where the key to be inserted is new? **Solution:**

$O(1)$

(g) What is the **BEST CASE** runtime for an insertion into an AVL tree, where the key to be inserted is new? **Solution:**

$O(\log n)$. AVL trees always create new nodes below a current leaf, so you need to traverse $O(\log n)$ levels to get to the bottom.

Since we're doing big-O analysis here, you should only consider arbitrarily large trees (saying "in an empty

tree, it would take $O(1)$ time" is not a correct statement, because $O()$ lets me pick n_0 much bigger than the tree you thought of.)

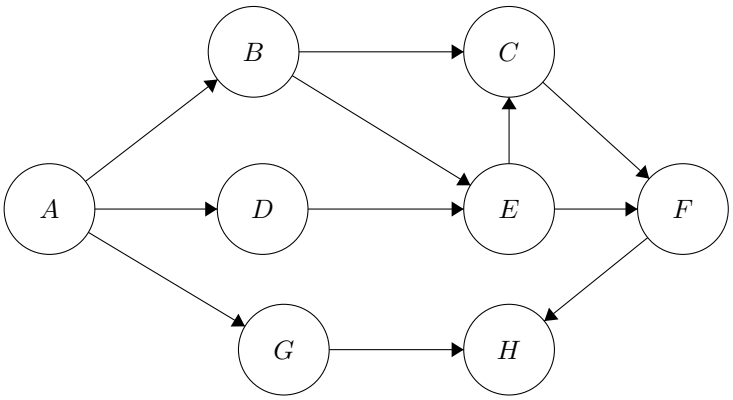
(h) You are using a hash table with n key-value pairs. Your table uses separate chaining for collision resolution, and as the load factor has just reached 2, you need to resize. An array of appropriate size has already been allocated. What is a tight big-O for the **WORST CASE** time of the remaining resizing operations? **Solution:**

$O(n)$, you have to re-hash every element.

2. Graphs (Short Answers) [14 points]

Graph Basics [10 of 14 points]

(a) Give a valid topological sort ordering of the graph below. [2 points]

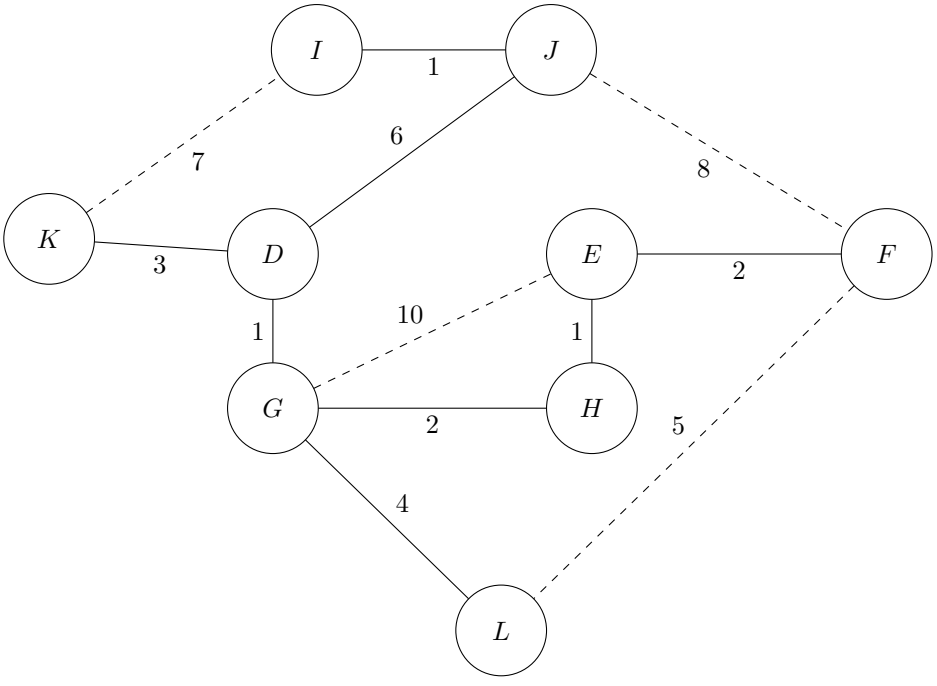


Solution:

There are many, for example: A, B, D, E, C, F, G, H
A, D, B, E, C, F, G, H
G can go anywhere in the ordering after A and before H, but all of the other nodes have to be in the ordering above

(b) **Color/trace through** the edges of an MST of the graph below. You do not need to show your work. [3 points]

Solution:



(c) Which kind of graph search (BFS or DFS) can be used to (efficiently) find strongly connected components of a graph? [1 point] **Solution:**

DFS

(d) Which kind of graph search (BFS or DFS) can be used to (efficiently) find shortest paths in an unweighted graph? [1 point] **Solution:**

BFS

(e) Which kind of graph search (BFS or DFS) can also be implemented recursively (hint: remember that recursion uses the call *stack*)? [1 point] **Solution:**

DFS

(f) Suppose vertex x has k neighbors in a graph with n vertices and m edges. If you use an *adjacency list* representation of your graph (where the lists are doubly-linked lists), what is the big-O running time of listing all neighbors of x? [1 point] **Solution:**

$O(k)$.

(g) Answer the same question, but for an *adjacency matrix*. [1 point] **Solution:**

$O(n)$

A Modified Graph Algorithm [4 of 14 points]

Imagine you have a graph where you are guaranteed that all edges have the weight of either 2, 4, or 6.

You design a modified version of Prim's Algorithm, where instead of using a heap to get the vertex that is reached by the next smallest edge to add to the MST, you create four sets of vertices: a set for vertices not seen yet, a set for vertices reachable by an edge of weight 2, a set for vertices reachable by an edge of weight 4, and a set for vertices reachable by an edge of weight 6. As you find edges that reach a vertex in a smaller weight, you move the vertex between the sets.

You will implement your sets using hash tables. Since you control the keys (the names of the vertices), do your analysis assuming there are no collisions in any of your tables.

(a) Suppose you are moving a vertex from a set with p vertices to a set with q vertices. What is the **WORST CASE** running time to move the vertex from one set to the other? [2 points] **Solution:**

O(1). We are heavily relying on the no collisions assumption in this part.

(b) Let your graph have m edges and n vertices. What is the **WORST CASE** overall running time of this version of Prim's? [2 points] **Solution:**

$O(n + m)$. We'll have to process each vertex and edge, but every step is now constant time thanks to the hash tables.

3. Sorting (Short Answers) [15 points]

Which Sort? [9 of 15 points]

For each of the following scenarios choose exactly one of the following sorts as the best option, and give a 1-2 sentence explanation of why it is the best sort for the given scenario.

The sorts available to you are:

- Heapsort
- Mergesort
- Quicksort (using median-of-3 for pivot selection)

You will use each sort once.

(a) Every night you get a dataset consisting of many (separate) lists. You want a sorting algorithm which will sort each of the lists (sequentially, one after the other). There is no pattern to the lists (you can think of them as randomly ordered). Your goal is to finish sorting all the lists as soon as possible, so you care about constant factors. **Solution:**

> Quicksort has the best constant factors. Another way to get this answer is to notice that if we have a large number of randomly ordered lists, then we care about the average case, for which we said quicksort is the best.

(b) You are writing code for an IoT device. Because its memory is so limited, it gets a list that takes up more than half of its memory. To ensure the device has good uptime, you care about sorting the list with the best-possible worst-case time. **Solution:**

> Since we're already using half of our memory, we need an in-place sort. That eliminates mergesort. Between quicksort and heapsort, heapsort has the better worst-case behavior.

(c) You are finishing writing backend code for a new gradebook to be used by CSE instructors. You know that instructors often need to sort across multiple columns (e.g. to alphabetize, they'll sort by last name then first name). **Solution:**

> Mergesort – we need a stable sort and neither heapsort nor (standard) quicksort are stable.

Sorting Lower Bounds [6 of 15 points]

Recall that we showed a sorting lower-bound in class (a limit on how fast a sorting algorithm could be). For each of the following, you should state either

- "impossible" if the claim contradicts the lower-bound from class.
- "possible" if the claim does not contradict the lower-bound from class (whether you know how to write the described algorithm or not!).

Also include 1-3 sentences explaining your answer. Your explanation should relate to the sorting lower-bound ("I've never heard of that algorithm" or "Oh, that's radix sort" aren't good explanations, because they don't relate to the sorting lower-bound).

(a) Sashu tells you she has designed a comparison-based algorithm that finds the median of a list of n elements in $O(n)$ time.

Solution:

Possible (indeed, you'll learn such an algorithm in 421, and we briefly mentioned in class that this exists). One way of finding a median is to sort the list and take the middle element. If you did it that way, it would certainly take $\Omega(n \log n)$ time, but we didn't say how we were doing it! Other algorithms that don't fully sort the list exist.

(b) Allen shows you his new comparison-based sorting algorithm. The **BEST CASE** running time is $\Theta(n \log \log n)$
Solution:

Possible! (we even saw one with $\Theta(n)$ best-case time in class). The lower-bound applies in the worst case.

(c) Aashna shows you a comparison-based sorting algorithm that has a **WORST CASE** running time of $O(n \log^*(n!))$.
Solution:

Impossible! $\log^*(n!) = 1 + \log^*(n \log n) = 2 + \log^*(\log(n) + \log \log n) < 2 + \log^*(2 \log n) << \log(n)$.

Or, more briefly, $\log^*$ makes things incredibly small. Small enough that we more than account for the $n!$, and this running time is better than $n \log(n)$, contradicting the lower-bound.

4. Concurrency [15 points]

The Allen school is hoping to prevent students from having to wait in a physical line for the career fairs and has implemented its own better virtual queue that students can use to join online. Assume the `ConcurrentLinkedQueues` are THREAD-SAFE (that is, multiple threads may operate on them simultaneously without any concurrency issues in the objects themselves.

You should also assume the queues have enough space and operations on them will not throw an exception.

```
1  class CareerFairQueue {
2      private int maxQueueSize = 20;
3      private Queue<String> names = new ConcurrentLinkedQueue<>();
4      private Queue<String> numbers = new ConcurrentLinkedQueue<>();
5
6      public String joinQueue(String name, String number) {
7
8          if (names.size() < this.maxQueueSize) {
9              names.add(name);
10             numbers.add(number);
11             return "joined queue";
12         } else {
13             return "unable to join queue, queue is full";
14         }
15     }
16
17     public String getNextStudent() {
18         if (names.isEmpty()) {
19             return null;
20         } else {
21             String nextName = names.remove();
22             String nextNumber = numbers.remove();
23             return nextName + " " + nextNumber;
24         }
25     }
26 }
```

(a) Does the `CareerFairQueue` class as shown above have (bubble all that apply):

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

If there are any problems, give an example of when they could occur. Be specific, and use the line numbers above.

Solution:

There are race conditions in this code.

Two threads may both try to join the queue, but there can be a bad interleaving where thread1 adds their name then thread2 adds their name and their number and then finally thread1 adds their number. This will cause the name and number to be mismatched. A similar race condition behavior can occur with getNextStudent.

There is no data race since the Queue is assumed to be thread-safe and maxQueueSize is only ever read.

(b) The Allen School decides to add one more method to the class to increase the maxQueueSize.

```
1  public void increaseMaxQueueSize(int amount) {
2      maxQueueSize += amount;
3  }
```

Does adding this method **cause any new** (bubble all that apply):

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

If there are any NEW problems, give an example of when they could occur. Be specific, and use the line numbers on this page and the prior page. Remember this code goes inside the class on the previous page (even though the line numbers for this snippet start over at 1). **Solution:**

There is a new data race and race condition that has been introduced!

Two threads could both call increaseMaxQueueSize which means that two threads will be writing and reading the same value at the same time. By definition, a data race is a type of race condition.

(c) On the next page, we will have another copy of the code for the `CareerFairQueue` including the `increaseMaxQueueSize` method we added. Insert (and use) locks on the next page to allow the most concurrent access while avoiding all of the potential problems listed above. For full credit you must allow the most concurrent access possible without introducing any errors. Create locks as needed, but do not create extraneous locks.

DO NOT use `synchronized`. You should create re-entrant lock objects and can call the locking methods as follows:

```
ReentrantLock lock = new ReentrantLock();
lock.acquire();
lock.release();
```

Solution:

```
class CareerFairQueue {
    private int maxQueueSize = 20;
    private Queue<String> names = new ConcurrentLinkedQueue<>();
    private Queue<String> numbers = new ConcurrentLinkedQueue<>();

    private ReentrantLock queueLock = new ReentrantLock();
    private ReentrantLock sizeLock = new ReentrantLock();

    public String joinQueue(String name, String number) {
        sizeLock.acquire();
        queueLock.acquire();
        if (names.size() < this.maxQueueSize) {
            names.add(name);
            numbers.add(number);
            sizeLock.release();
            queueLock.release();
            return ”joined queue”;
        } else {
            queueLock.release();
            sizeLock.release();
            return ”unable to join queue, queue is full”;
        }
    }

    public String getNextStudent() {
        queueLock.acquire();
        if (names.isEmpty()) {
            queueLock.release();
            return null;
        } else {
            String nextName = names.remove();
            String nextNumber = numbers.remove();
            queueLock.release();
            return nextName + ” ” + nextNumber;
        }
    }

    public increaseMaxQueueSize(int amount) {
        sizeLock.acquire();
        maxQueueSize += amount;
        sizeLock.release();
    }
}
```

5. Finish the ForkJoin [13 points]

Call a number **very odd** if it is both (1) odd itself and (2) stored at an odd-index of the array. You wish to write fork-join code that will return the **product** of all very odd numbers in an array. If there are no very odd numbers, you should return 1. You may assume the final answer is small enough to be stored in an `int`.

For example, if your array is $[5, \mathbf{3}, 2, \mathbf{3}, 4, 6, 7, \mathbf{5}]$ you should return $3 \cdot 3 \cdot 5 = 45$. Notice that the 6 was not part of the final answer because the number itself is not odd.

Finish the code snippets below to give fork-join code with work $O(n)$ and span $O(\log n)$.

Solution:

```
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 sum of very odd numbers in input.
  public static int SumVeryOdd(int[] input){


  return fj.invoke(new VeryOddTask(input, 0, input.length   ) );
  }
}

public static class VeryOddTask extends RecursiveTask<Integer>{
  int[] input;
  int lo;
  int hi;

  public VeryOddTask(int[] input, int lo, int hi){
    this.input = input;
    this.lo = lo;
    this.hi = hi;
  }
  public Integer compute() {
    if (hi - lo <= 1) {
      if(lo % 2 == 1 && input[lo] == 1)
        return new Integer(input[lo]);
      else return new Integer(0);
    }

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

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

      right.fork();
      Integer leftResult = left.compute();
      Integer rightResult = right.join();

      Integer result = leftResult + rightResult;
```

```
      return result;

    } //end of else branch
  } //end of compute method
} // end of VeryOddTask class
```

6. Graph Modeling [10 points]

With the new light rail station, the Northgate area is being converted to an outdoor mall, but renovations aren't complete yet. The planning commission has a goal that pedestrians must be able to get from any shop to any other using sidewalks. Currently none of the sidewalks have been completed. You have a map of all the shops, along with the potential locations where sidewalks can be added. Every potential sidewalk also contains the length (in feet) of sidewalk required to connect those two shops.

You know that it takes $\ell(\ell - 5) + 10$ hours to make a sidewalk of length ℓ feet. Sadly, you can afford only one sidewalk construction team, and with the station already opened, you need to make a plan which will meet the commission's goal of connecting all the shops as soon as possible.

(a) What will the vertices of your graph be? **Solution:**

We will have a vertex for each possible shop.

(b) What will the edges be? You should at least say whether your edges are directed or not and whether they're weighted or not. **Solution:**

We will have **undirected** edges, one in every location where we can put a section of sidewalk. If the sidewalk would have length ℓ, then we assign weight $\ell(\ell - 5) + 10$.

(c) What algorithm will you run on your graph? **Solution:**

Either Prim's or Kruskal's algorithm is acceptable.

(d) How will you interpret the output of your algorithm? (i.e., what sidewalks are you building "in the real world" instead of just in graph terms). (1-3 sentences) **Solution:**

We should build a sidewalk for every edge that the MST selects (i.e., if the edge (u, v) is in the MST, then we build the sidewalk that goes directly from shop u to shop v.

(e) Briefly (2-4 sentences) explain why your model works. You should at least address why you ran the algorithm you did (e.g., why are you looking for a shortest path/MST/topological ordering/etc.) and how you are incorporating the formula relating sidewalk length to time. **Solution:**

Our goal is to ensure pedestrians can get from every location to every other. So we're looking for a connected and spanning subgraph. We want to get this done as soon as possible; since we set our edges to represent time, the weight of the edges we choose should be minimized. Thus a minimum spanning tree will give us the correct answer. The formula for sidewalk length is used directly in the edge weights, so the time to completion (rather than length of sidewalk) is represented by the weight of the MST.

7. P vs. NP [10 points]

(a) What two words does "NP" stand for? [2 points] **Solution:**

Nondeterministic Polynomial

(b) For each of the following problems, bubble all the categories which the problem is **known** to be in. [2 points each]

Given a directed graph, decide whether there is a tour (a walk that visits every vertex exactly once) of length at most k.	☐ P	☐ NP	☐ NP-complete	☐ None of these
Given an undirected graph, decide whether there is a spanning tree of weight at most k.	☐ P	☐ NP	☐ NP-complete	☐ None of these
Given an array of `ints`, sort the array.	☐ P	☐ NP	☐ NP-complete	☐ None of these

(c) Briefly (2-3 sentences) describe why designing a polynomial-time algorithm for 3-coloring would mean you also have a polynomial time algorithm for 3-SAT. Your answer should include both the mention of a complexity class and a description of what the 3-SAT algorithm would look like. [2 points] **Solution:**

Since 3-coloring is NP-complete (and 3-SAT is in NP) we know that 3-SAT reduces to 3-coloring. But then if we have a polynomial time algorithm for 3-coloring we can just "run the reduction" (that is plug in the library function into the reduction) to get a polynomial time algorithm for 3-SAT.

8. Parallel QuickSort [9 points]

The TAs thinks this is the trickiest problem on the exam, which is why it's at the end. We've intentionally made it worth fewer points than expected for its difficulty, so don't be afraid to jump around and come back if you have time.

In class, we designed a parallel version of quick sort that used a heuristic to find a (usually) good pivot. In this problem, we'll modify the algorithm to find the true median to use as the pivot. For simplicity, throughout the problem assume all elements of the array are distinct, and that we have any needed auxiliary arrays already allocated.

To find the median, we'll run n reduces, where the i^{th} reduce will calculate "how many elements of `A` are smaller than $A[i]$"?

(a) What is the **span** of running all n of these reduces (include both the individual reduces and the process of forking/joining enough threads to run them)? Give your answer in *simplified* big-O. [1 point] **Solution:**

$O(\log(n))$ – each of the reduces can be done in parallel relative to each other! So it's $\log(n) + \log(n)$. Another way to think of it: we need n^2 threads total, but $\log(n^2)$ is $2\log(n)$.

(b) What is the **work** of running all n of these reduces (include both the individual reduces and the process of forking/joining enough threads to run them)? Give your answer in *simplified* big-O. [1 point] **Solution:**

$O(n^2)$. We're doing n iterations through the (length n) array.

(c) Briefly (3-5 bullet-points or sentences) describe how, given `B` where `B[i]` contains the output of the i^{th} reduce, you can output `m` the index of the median of `A` (the original array). Assume all threads from the last step have already joined together into a single thread. You may not use concurrency primitives (e.g. no locks). Your process must have the best possible big-O span, but don't worry about constant factors. You may assume you already have extra auxiliary arrays if needed. [3 points] **Solution:**

We run a reduce, where each thread returns: i if $B[i]$ contains $\lfloor n/2 \rfloor$ (since the median has $\lfloor n/2 \rfloor$ elements smaller than it – we accepted anything that was within 1 in either direction, including answers that did not contain ceilings/floors) and -1 otherwise. When joining, we return the positive number returned by a thread (if one exists), or -1 otherwise (i.e., if both threads we're joining returned -1.

(d) What is the work and span of your process? [1 point]

Solution:

Since we're reducing an array of size n, the work is $O(n)$ and the span is $O(\log n)$.

(e) Recall that the other non-recursive work of quicksort is to pack elements smaller than the pivot into one array and the elements greater-than-or-equal to the pivot into another. What is the span of **just that part** of the process (give a simplified, big-O) [2 points] **Solution:**

$O(\log n)$.

(f) Give a recurrence that describes the worst-case **span** of our new quicksort (using the exact median finding). You may ignore ceilings and floors in your recurrence, and use O-notation in the non-recursive work descriptions. [1 point] **Solution:**

$$T(n) = \begin{cases} T(n/2) + O(\log(n)) & \text{if } n > \text{cutoff} \\ O(1) & \text{if } n \leq \text{cutoff} \end{cases}$$

9. Drawing [1 point]

Draw what you would do in a world where you had just found an extremely efficient algorithm for 3-SAT. These help TA morale while grading! **Any non-blank page will receive the point**.

Gumball wishes you a happy Winter Break!