CSE 332: Data Abstractions — Autumn 2015

Final Exam Solutions

Name:	**Sample Solutions**		
ID #:	**1234567**		
TA:	**The Best**	**Section:**	**A9**

INSTRUCTIONS:

- You have **110 minutes** to complete the exam.
- The exam is closed book. You may not use cell phones or calculators.
- All answers you want graded should be written on the exam paper.
- If you need extra space, use the back of a page. Make sure to mention that you did though.
- The problems are of varying difficulty.
- If you get stuck on a problem, move on and come back to it later.
- It is to your advantage to read all the problems before beginning the exam.

Problem	Points	Score	Problem	Points	Score
1	13		5	10	
2	10		6	12	
3	10		7	20	
4	10		8	15	
			Σ	100	

One Liners.

This section has questions that require very short answers. To get full credit, you should answer in no more than one sentence per question.

1. ASN [13 points]
For each of the following, answer **ALWAYS**, **SOMETIMES**, or **NEVER**. Give a brief (one sentence) explanation of your answer for each.

(a) (2 points) In an implementation of the `UnionFind` ADT, `find` is $\mathcal{O}(1)$.

Solution: **Sometimes.** `find` could be $\mathcal{O}(1)$ (e.g., using the first implementation we discussed), but it could also be much worse (e.g., if we do a linear search for the smallest one).

(b) (2 points) In an implementation of the `UnionFind` ADT, `union` is amortized $\mathcal{O}(1)$.

Solution: **Sometimes.** The reasoning is identical to the previous question.

(c) (2 points) An algorithm that solves the `SORT` problem must have $\Omega(n \lg n)$ swaps.

Solution: **Never.** We proved in lecture that the generalized `SORT` problem must have $\Omega(n \lg n)$ **comparisons**. Consider *selection sort* which only does $\mathcal{O}(n)$ swaps.

(d) (2 points) `SORT` $\in$ P

Solution: **Never.** `SORT` is not a decision problem.

(e) (2 points) `CIRCUITSAT` $\in$ NP

Solution: **Always.** `CIRCUITSAT` is a decision problem, the witness is a satisfying assignment, and the verifier checks that the assignment actually works.

(f) (2 points) `BST-FIND` $\in$ NP (Given a BST T and a number n, is $n \in T$?)

Solution: **Always.** `BST-FIND` is a decision problem, and, since `BST-FIND` $\in$ P, we know `BST-FIND` $\in$ NP.

(g) (1 point) Radix Sort can be used to sort a list of numerical data.

Solution: **Always.** Radix sort works on any kind of data which can be represented as a "sequence" of comparable values. We can represent each number as a sequence of digits, so radix sort works. You could argue that this is "sometimes" if you consider the case where the numerical data is made up of real numbers.

Basic Techniques.

This part will test your ability to apply techniques that have been explicitly identified in lecture and reinforced through sections and homeworks. Remember to show your work and justify your claims.

2. RCTOA NDECOIIN [10 points]

Imagine we run `main`. Is there a race condition? If there is one, explain why by showing a bad interleaving and explaining what the race is. If not, explain why not.

```
1  public static Lock lock = new ReentrantLock();
2  public static Stack<Integer> stack;

3  public static void main(String[] args) {
4     stack = <initialize stack with elements>;
5
6     Task t1 = <run task>;
7     Task t2 = <run task>;
8     t1.fork();
9     t2.fork();
10    int sum = t1.join() + t2.join();
11    System.out.println("sum = " + sum);
12 }
```

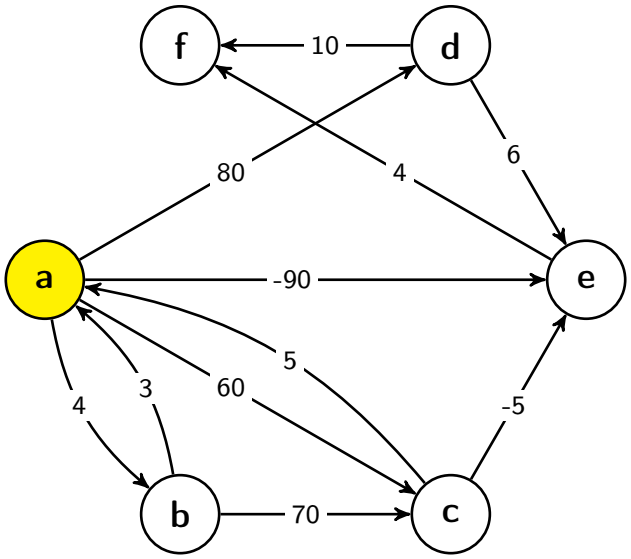
```
13 public int task() {
14    int count = 0;
15    while (!stack.isEmpty()) {
16       lock.lock();
17       count += stack.pop();
18       lock.unlock();
19    }
20    return count;
21 }
```

Solution:

`int count = 0`	
	`int count = 0`
`while (!stack.isEmpty())`	
	`while (!stack.isEmpty())`
`lock.lock()`	
`count += stack.pop()`	
`// stack is now empty!`	
`lock.unlock()`	
	`lock.lock()`
	`count += stack.pop()`
	`// stack is now empty!`
	`lock.unlock()`
`return count`	
	`return count`

Consider a stack containing exactly one element. When both stacks start, they both call `stack.isEmpty()`, both obtain `false`, and so both enter the while loop. Then, both tasks will attempt to pop something from the stack. One will succeed, but then the other will fail and throw an exception. An exception is the incorrect behavior, as we don't expect the program to crash.

3. GOTO Considered Harmful; while loops considered okay [10 points]



(a) (6 points) Use Dijkstra's Algorithm to find the **lengths** of the shortest paths from **a** to each of the other vertices. For full credit, you must show the worklist at every step, but how you show it is up to you.

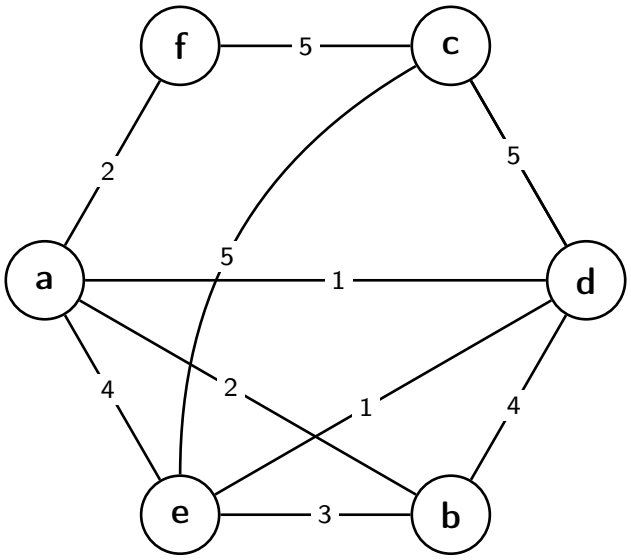
Solution:

Vertex	Init	a	b	c	d	e	f
a	0	✓					
b	∞	4			✓		
c	∞	60				✓	
d	∞	80					✓
e	∞	-90	✓				
f	∞	∞	-86	✓			

(b) (4 points) Are any of the lengths you computed using Dijkstra's Algorithm in part (a) incorrect? For each length that is incorrect, explain what the correct answer is and why the answer from part (a) was incorrect.

Solution: None of the lengths are incorrect.

4. Spring Time! [10 points]



(a) (5 points) Use Kruskal's Algorithm to find *two* minimum spanning trees of the above graph.

Solution: Three possible answers are:

- e-d, a-d, a-f, a-b, f-c
- e-d, a-d, a-f, a-b, c-d
- e-d, a-d, a-f, a-b, e-c

On an exam, we would expect you to show the forests, when each edge is chosen, etc. Just showing the answer would not get any credit.

(b) (5 points) Imagine that the above graph had some negative edges in it. Would Prim's Algorithm necessarily return a correct result? Explain your answer in 1-2 sentences.

Solution: Prim's algorithm **will** return the correct result, even with negative edges. Prim's algorithm works only by doing a relative comparison between the different edges currently under comparision and will pick the cheapest. This is in contrast to Dijkstra's, which both compares AND sums up the edges and assumes that each new edge increases the cost of the path.

5. Definitely A Graph! [10 points]

(a) (5 points) Suppose you are given a graph G. Explain how you would figure out if it has a cycle.

Solution: To find if a *directed* graph has a cycle, run topological sort. If the number of vertices at the end is not the number of total vertices, there is a cycle.

To find if an *undirected* graph has a cycle, run DFS. If you ever hit a node that you've already visited, then there is a cycle.

(b) (5 points) Suppose you are given a DAG G representing the work graph of a bunch of `ForkJoin` tasks. Explain how you would determine the longest dependency path of G. What does the longest dependency path in G represent with respect to the algorithm G?

Solution: Run DFS on every node with an in-degree of zero. Keeping track of the longest path found along the way.

Since this DAG represents ForkJoin operations, the longest dependency path represents the span of whatever algorithm is doing the forking and joining.

6. X-Tra [12 points]
Consider the `expand` problem. `expand` takes in an `int[]` and outputs a new `int[]` where each element `a[x]` is copied `a[x]` times. For example,

```
expand([1, 2, 1, 4, 5, 3]) = [1, 2, 2, 1, 4, 4, 4, 4, 5, 5, 5, 5, 5, 3, 3, 3]
```

Give a high-level algorithm to solve the `expand` problem. DO NOT WRITE CODE! Your solution should have $\mathcal{O}(n)$ work and $\mathcal{O}(\lg n)$ span where n is the sum of the elements in the array. Be sure to explain why your solution meets this bound.

Solution: Expand is basically the opposite of pack. We do `expand` using the following steps:

- `scan` the input array using $+$ as the operation. Call the output array `sumA`.
- Allocate a new array with the size of the last element of `sumA`.
- `map` over `input` by filling indices `sumA[i - 1]` (or `0` if `i = 0`) to `sumA[i]` with `input[i]`

Note that `map` and `scan` both have $\mathcal{O}(n)$ work and $\mathcal{O}(\lg n)$ span; so, the algorithm overall has these bounds as well.

A Moment's Thought!

This section tests your ability to think a little bit more insightfully. The approaches necessary to solve these problems may not be immediately obvious. Remember to show your work and justify your claims.

7. Peek-a-boo [20 points]

(a) (15 points) Write a `ForkJoin` algorithm to solve the following problem:

> **Input(s)**: An int k, An array of ints with values between 0 and k
> **Output**: The largest missing number in the array, or -1 if none of them is missing

Your solution must have $\mathcal{O}(n)$ work and $\mathcal{O}(\lg n)$ span where n is the size of the input array. You may assume that k is much smaller than n. You may not use any global data structures or synchronization primitives (locks).

```
public class LargestMissingNumber {
   private static final ForkJoinPool POOL = new ForkJoinPool();

   public int largestMissingNumber(int k, int[] input) {
```

Solution:

```
      boolean[] result = POOL.invoke(new LargestMissingTask(input, 0, input.length);
      for (int i = k - 1; i > 0; i--) {
         if (result[i]) {
            return i;
         }
      }
      return -1;

   }
```

Solution:

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

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

         LargestMissingTask left = new LargestMissingTask(input, lo, mid);
```

```
            LargestMissingTask right = new LargestMissingTask(input, mid, hi);

            right.fork();

            boolean[] left = left.compute();
            boolean[] right = right.join();

            for (int i = 0; i < k; i++) {
               left[i] |= right[i];
            }
            return left;
         }

   }
```

(b) (5 points) Write recurrences for the work and the span for your solution in terms of n and k.

Solution:

- $\mathsf{work}(n, k) = 2 \cdot \mathsf{work}(n/2, k) + k$ ($\mathsf{work}(1) = k$)
- $\mathsf{span}(n, k) = \mathsf{span}(n/2, k) + k$ ($\mathsf{span}(1) = k$)

8. FootPen Needs Your Help! [15 points]
A new social networking company called FootPen has arrived on the start-up scene. Because UW CSE students are known to be the best in industry, FootPen has asked you to help them implement several features. For each feature request,

- explain how to represent the problem as a graph,
- give a (high-level) idea for an algorithm to solve the problem, and
- give a runtime analysis of your algorithm.

(a) (5 points) How can FootPen determine the number of people that have at least one friend in common with a particular user.

Solution:
Representation: Each person is a vertex; each friendship is an edge between two friends. The graph is undirected and unweighted.

Algorithm: Run BFS, but stop two vertices away from the particular user. Ignore the set of people found one vertex away (friends), and return the set of people found two vertices away (friend of a friend). If somebody is found in both sets, return them (since they do have a friend in common).

Runtime: We're running a BFS. Even though we're only going two levels out, it's possible that we have to traverse everything. So, it's graph linear.

(b) (5 points) FootPen would like to add a "make a new friend" feature. To facilitate this feature, FootPen has an algorithm that generates an "familiarity score" for every pair of friends. The familiarity score is a real number between 0.0 and 1.0 which represents how well two users know each other. The closer the score is to zero, the more friendly the two users are.

The "make a new friend" feature is intended to help users find *a single user* who *they do not know* but are likely to get along with. We can estimate a familiarity score between two users who *are not friends* by summing the scores of the shortest weight chain of friends that joins the two non-friends.

FootPen has run experiments and found that an estimated familiarity score of between 0.5 and 1 is likely to indicate that two users do not already know each other but would make good friends.

How can FootPen write the "make a friend feature"?

Solution:
Representation: Each person is a vertex; Each friendship an edge between two friends. Make the weight of an edge be the "familiarity score" between those two people. The graph is undirected.

Algorithm: If person A would like a friend, run Dijkstra's algorithm starting at their vertex. Terminate the exploration process once the total cost exceeds 1.0 (since the friendliness score is too high to be worth exploring more and there are no negative weights). Return all encountered vertices that are not directly adjacent to the starting vertex.

Runtime: In the worst case, we have to explore the entire graph. So, the runtime is the same as standard Dijkstra's, which is $\mathcal{O}((|E| + |V|) \lg(|V|))$.

(c) (5 points) Now that FootPen has become popular, the government has sent a request asking for the group of people who correspond with each other the most. FootPen has records of the number of conversations each pair of users has in its database. How can FootPen identify the k users with the largest number of inter-communications?

Solution:

Representation: Each person is a vertex; each correspondence an edge between two people. Make the weight of an edge be the number of messages sent between those two people. The graph is undirected.

Algorithm: We need to find the set of k people with the largest number of inter-communications. The easiest way to do this is to brute force through all sets of k people. Note that TopKSort doesn't work here, because we want the *group of k people with the most communications* which is not the same as *the k people with the most communicatons*.

Runtime: The runtime is $\mathcal{O}(|V|^k)$.