Final Exam
Summer 2024

Name Answer Key

Net ID (@uw.edu)

Academic Integrity: You may not use any resources on this exam except for your one-page (front and back) reference sheet, writing instruments, your own brain, and the exam packet itself. This exam is otherwise closed notes, closed neighbor, closed electronic devices, etc.. The last three pages of this exam provide a list of potentially helpful identities, the code referenced in Section 5, and room for scratch work (respectively). Please detach those last three pages from the exam packet. No markings on these last three pages will be graded. Your answer for each question must fit in the answer box provided.

Instructions: Before you begin, **Put your name and UW Net ID at the top of this page.** Make sure that your name and ID are LEGIBLE. Please ensure that all of your answers appear within the boxed area provided.

Section	**Max Points**
Hash Tables	16
Sorting	9
Graphs	13
Parallelism	15
Concurrency	13
Extra Credit	(+2)
Total	66

Section 1: Hash Tables

(2 pts)**Question 1: Clustering**
Using 1-2 sentences, explain how quadratic probing addresses the clustering problem of linear probing.

When there is a collision, quadratic probing searches increasingly far away, so clusters of occupied indices are less likely to grow.

(2 pts)**Question 2: Double Hashing v. Quadratic Probing**
Using 1-2 sentences, explain one improvement that double hashing makes compared to quadratic probing.

In quadratic probing, when two keys has to the same index, they will follow the same probing pattern and therefore collide at all the same places. In double hashing we use a second hash function to make it unlikely for this to occur.

(2 pts)**Question 3: Separate Chaining v. Open Addressing**
In 1-2 sentences, explain why, in general, a separate chaining hash table permits a larger load factor before resizing compared to an open addressing hash table.

It's impossible for an open addressing hash table to have a load factor greater than 1.

(5 pts)**Question 4: Quadratic Probing**
Insert 27, 39, 18, 17, 29, 37, 16 (in that order) into the open addressing hash table below. You should use the primary hash function $h(k) = k\%10$. In the case of collisions, use quadratic probing for collision resolution. If an item cannot be inserted into the table, indicate this and continue inserting the remaining values. Do not resize the hash table.

Items that could not be inserted:

0	29
1	17
2	
3	
4	
5	16
6	37
7	27
8	18
9	39

(5 pts)**Question 5: Double Hashing**

Insert 18, 28, 38, 14, 22 (in that order) into the open addressing hash table below. You should use the primary hash function $h(k) = k\%10$. In the case of collisions, use double hashing for collision resolution where the secondary hash function is $g(k) = 1 + (k\%7)$. If an item cannot be inserted into the table, indicate this and continue inserting the remaining values. Do not resize the hash table.

Items that could not be inserted:	

0	
1	
2	38
3	
4	14
5	
6	22
7	
8	18
9	28

Section 2: Sorting

(4 pts)**Question 6: Sorting Olympics**
For each scenario below, select the sorting algorithm property the situation most needs, then select the *fastest* sorting algorithm with that property. Select only from the options provided.

1. A total of 206 different nations have participated across 30 summer Olympics. The most gold medals ever one by a country in a single Olympics was the U.S., which won 83 gold medals in 1984. Which algorithm should we use to sort all 6,000 country-year pairs by the number of gold medals won by that country in that year?

   Property Options: In Place, Stable, Adaptive, Online, Non-Comparison-Based.

   Property Needed: non-comparison based

   Algorithm Options: Quick Sort, Insertion Sort, Heap Sort, Radix Sort.

   Algorithm Suggestion: Radix Sort

2. Suppose we had a list of all runners already sorted by their finish time in the 100 meter dash. A small number of runners incurred time penalties, so we need to sort the list again to adjust for those penalties. What algorithm should we use for this second sort?

   Property Options: In Place, Stable, Adaptive, Online, Non-Comparison-Based.

   Property Needed: Adaptive

   Algorithm Options: Quick Sort, Insertion Sort, Merge Sort.

   Algorithm Suggestion: Insertion Sort

(2 pts)**Question 7: In-Place Sort**
Which of the following best matches the definition of an in-place sort? Write the letter of your choice in the box.

A. Items are re-ordered by swapping within the given list data structure

B. There is no randomness used in the sorting algorithm

C. The worst case running time matches the best case running time

D. Items that are already at the correct index will not be moved

E. If you re-run the algorithm on an already-sorted list, no items will change order

A

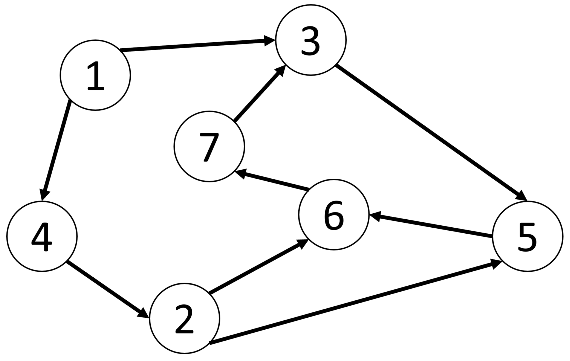
(3 pts)**Question 8: Quick Sort Runtime**

Using 1-2 sentences, explain why the method used to select the pivot for Quick Sort should not take more than linear time.

If the pivot takes more than linear time to select, then the solution to the recurrence relation for its running time will be worse than $n \log n$.

Section 3: Graphs

(2 pts)**Question 9: BFS**
For the graph below, list the nodes in an order that they might be removed from the queue in a BFS starting from node 1 (note that this is the same as the previous graph, but now undirected).



BFS Order: 1, [3 and 4], [2 and 5], 6, 7
(e.g. 1,3,4,5,2,6,7)

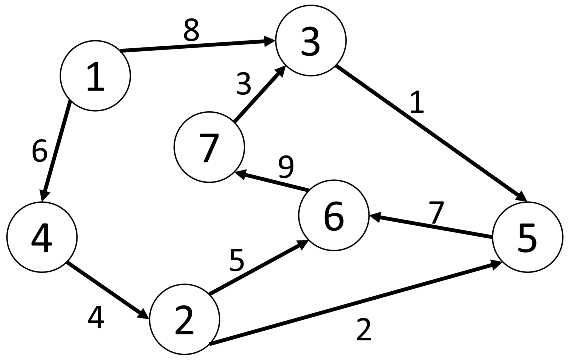
(2 pts)**Question 10: DFS**
Using the same graph as the previous problem, list the nodes in a depth-first-search order starting from node 1. If there are multiple choices for the next node, you should always select the node with the smallest value first.

BFS Order: 1,3,5,6,7,4,2

(1 pt)**Question 11: Back Edge**
Identify the first back edge found by the DFS done in the previous problem.

(7,3)

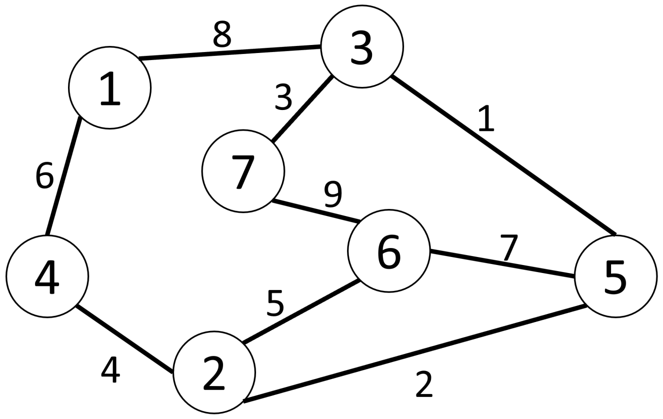
(2 pts)**Question 12: Dijkstras**
For the graph below, list the nodes in an order that they might be removed from the priority queue when running Dijkstra's algorithm starting from node 1 (note that this is the same as the previous graph, but now with weights).



Dijkstra's Order: 1,4,3,5,2,6,7

(6 pts)**Question 13: MSTs**

The next 2 questions will relate to running minimum spanning tree algorithms (Kruskal's and Prim's) on the graph below (which is the same as the previous graph, but now undirected):



1. What are the weights of the first three edges added to the minimum spanning tree when running Kruskal's algorithm?

   First edge's weight: 1

   Second edge's weight: 2

   Third edge's weight: 3

2. What are the weights of the first three edges added to the minimum spanning tree when running Prims's algorithm starting with node 1?

   First edge's weight: 6

   Second edge's weight: 4

   Third edge's weight: 2

Section 4: Parallelism

(8 pts)**Question 14: ForkJoin**
For this question you will complete a parallel implementation of the following sequential method using the Java ForkJoin Framework.

```
static boolean lastEmpty(String[] arr){
    int last = -1;
    for(int i = 0; i < arr.length; i++){
        if (arr[i].length() == 0){
            last = i;
        }
    }
    return last;
}
```

This method returns the last index which contains the empty string from a given array of strings, or -1 if the empty string does not appear in the list. For example, given the array `["0", "1", "" , "3", "", "5"]` the method would return 4.

On the next page we have an incomplete parallel implementation of `lastEmpty` using ForkJoin. In particular, we have provided:

– A Client class that has the `lastEmpty` method.

– Fields for the `LastEmptyTask` class

– The constructor for the `LastEmptyTask` class

– The signature of the `compute` method as well as the sequential code that will run when the problem size is within the sequential cutoff.

And the code is missing:

– The body of the `lastEmpty` method, which should create an instance of `LastEmptyTask` and then call the `invoke` method of `ForkJoinPool`. (Starts on line 6)

– The class that `LastEmptyTask` should extend. (Line 10)

– The parallelized portion of the `compute` method. (Starts on line 31)

Complete our implementation by providing the missing code in the boxes following the code.

```
import java.util.concurrent.*;

public class Client {
    public static final ForkJoinPool POOL = new ForkJoinPool();
    public static int lastEmpty (String[] input) {
        // Part 1 answer will go here
    }
}

public class LastEmptyTask extends ??? { // Part 2 will replace the ???
    String[] arr;
    int hi;
    int lo;

    public LastEmptyTask(String[] arr, int lo, int hi){
        this.arr = arr;
        this.hi = hi;
        this.lo = lo;
    }

    public Integer compute(){
        if(hi-lo < 100){
            int last = -1;
            for(int i = lo; i < hi; i++){
                if (arr[i].length() == 0){
                    last = i;
                }
            }
            return last;
        }
        // Your implementation of compute from Part 3 will go here.
    }
}
```

Finish the code in the boxes below:

1. Implement the body of `lastEmpty` in the box provided. The code you provide will be placed starting at line 6 of the code above

```
return POOL.invoke(new LastEmptyTask(input, 0, input.length);
```

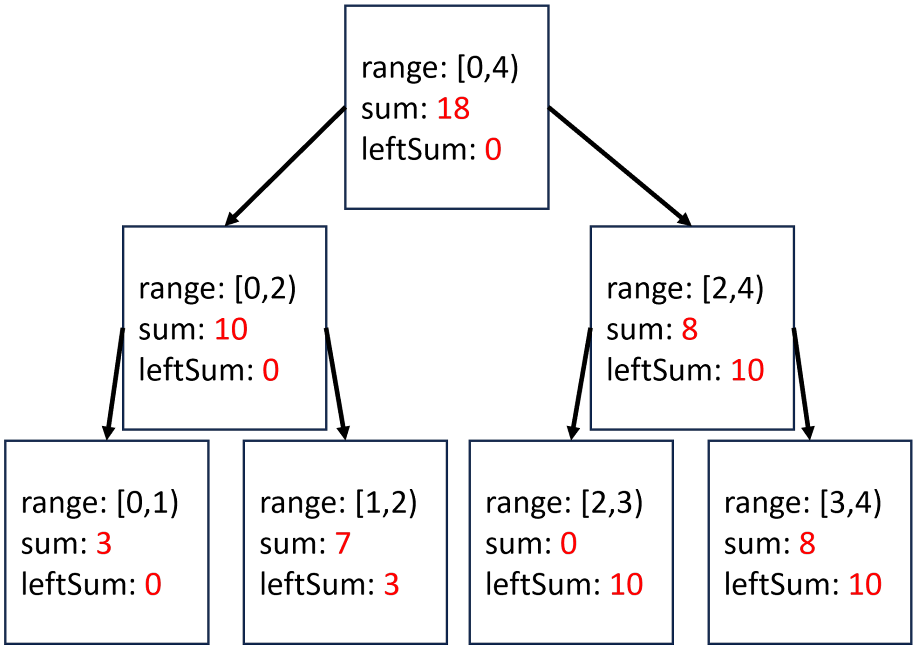
2. Finish line 10 from the code above by filling the in the ??? with the class that `LastEmptyTask` should extend.

```
RecursiveTask<Integer>
```

3. Finally, finish the `compute` method. Your code will begin on line 31 above.

```
int mid = lo+(hi-lo)/2;
LastEmptyTask left = new LastEmptyTask(arr, lo, mid);
left.fork();
LastEmptyTask right = new LastEmptyTask(arr, mid, hi);
int rightans = right.compute();
if(rightans >= 0)
        return rightans;
return left.join();
```

(4 pts)**Question 15: Parallel Prefix**
Given the array `[3, 7, 0, 8]` as input, fill in the tree below as it would be completed by the parallel prefix sum algorithm.



(3 pts)**Question 16: Amdahl's Law**
Suppose we have a program in which a $\frac{3}{4}$ proportion can be parallelized with perfect linear speedup. Using $T_1 = 1$, answer the following using Amdahl's Law:

1. What is T_2?

$\frac{5}{8}$

2. What is T_4?

$\frac{7}{16}$

3. What is T_{10}?

$\frac{13}{40}$

Section 5: Concurrency

The remaining questions in this section use the classes below in a parallel implementation. Both relate to finding socks in a sock drawer. Two socks will match whenever they have the same color. To find a match we select a first sock at random, then repeatedly select a second sock at random until a match is found.

```java
public static Random r = new Random();
public class Sock{
    public String color;
    public boolean isDirty = false;
    public Sock(String color){
        this.color = color;
    }
    public synchronized void wear(){
        if(this.isDirty)
            System.out.println("GROSS");
        this.isDirty = true;
    }
}
public class SockDrawer{
    public List<Sock> drawer;
    public SockDrawer(){
        this.drawer = new ArrayList<>();
    }
    public Sock pickRandom(){
        synchronized(drawer){
            int choice = r.nextInt(0, drawer.size());
            return drawer.get(choice);
        }
    }
    public synchronized void putOn(Sock sock){
        sock.wear();
        drawer.remove(sock);
    }
    public void wearPair(){
        Sock sock1 = pickRandom();
        synchronized(sock1){
            findMatch(sock1);
        }
    }
    public Sock findMatch(Sock sock1){
        Sock sock2 = pickRandom();
        while(!sock2.color.equals(sock1.color) || sock1==sock2){
            sock2 = pickRandom();
        }
        putOn(sock1);
        putOn(sock2);
    }
}
```

Note: The code is also provided on the last page of the exam, which you may detach for convenient reference.

(3 pts)**Question 17: Race Condition**

If `wearPair` is run sequentially, it is impossible for the code to print GROSS, regardless of the contents of the drawer. If two threads are both executing the `wearPair`, however, this is possible. Describe the drawer's contents and an interleaving that causes the code to print GROSS.

if two threads pick the same sock1 but different matching sock2's (for example).

Is the error in this code a Data Race or a Bad Interleaving error? Write either DR or BI in the box to indicate your answer.

BI

(2 pts)**Question 18: Deadlock**

This code additionally contains a potential for deadlock. Using about 3 sentences, explain how a deadlock could occur.

Consider there are exactly 2 black socks. Thread 1 selects a black sock as sock1 then Thread 2 selects the other black sock as sock1. Then when inside of findMatch, they must wait on each other to release their locks.

(8 pts)**Question 19: Deadlock and/or Race Condition**
Below we provide alternative implementations of the `wearPair` method.

For each implementation, consider two threads invoking the `wearPair` method. Indicate whether it is still possible for the code to print "GROSS" and whether there is still a potential for deadlock by writing "yes" or "no" in the corresponding box. Assume all code except for `wearPair` remains unchanged.

```
public void wearPair(){
    Sock sock1 = drawer.get(0);
    synchronized(sock1){
        findMatch(sock1);
    }
}
```

1. Deadlock? No
2. Can Print GROSS? Yes

```
public void wearPair(){
    Sock sock1 = drawer.get(drawer.size()-1);
    synchronized(sock1){
        findMatch(sock1);
    }
}
```

3. Deadlock? Yes
4. Can Print GROSS? Yes

```
public void wearPair(){
    synchronized(drawer){
        Sock sock1 = pickRandom();
        findMatch(sock1);
    }
}
```

5. Deadlock? No
6. Can Print GROSS? No

```
public void wearPair(){
    Sock sock1 = pickRandom();
    synchronized(sock1){
        drawer.remove(sock1)
        findMatch(sock1);
    }
}
```

7. Deadlock? No
8. Can Print GROSS? Yes

Extra Credit

(2 pts)**Question : Well Done!**
Overall, Nathan has felt that this has been an exceptional group of students, and is very proud of you all! More importantly, though, you should be proud of yourselves! Name one thing you accomplished this summer that you're proud of.

...

Scratch Work

Nothing written on this page will be graded.

Identities

Nothing written on this page will be graded.

Summations

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x} \text{ for } |x|< 1$$
$$\sum_{i=0}^{n-1} = \sum_{n}^{i=1} = n$$
$$\sum_{i=0}^{n} i = 0 + \sum_{n}^{i=1} i = \frac{n(n+1)}{2}$$
$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6}$$
$$\sum_{i=0}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2 = \frac{n^4}{4} + \frac{n^3}{2} + \frac{n^2}{4}$$
$$\sum_{i=0}^{n-1} x^i = \frac{1-x^n}{1-x}$$
$$\sum_{i=0}^{n-1} \frac{1}{2^i} = 2 - \frac{1}{2^{n-1}}$$

Logs

$$x^{\log_x(n)} = n$$
$$\log_a(b^c) = c\log_a(b)$$
$$a^{\log_b(c)} = c^{log_b(a)}$$
$$\log_b(a) = \frac{\log_d(a)}{\log_d(b)}$$

Section 5 Code

```java
public static Random r = new Random();
public class Sock{
    public String color;
    public boolean isDirty = false;
    public Sock(String color){
        this.color = color;
    }
    public synchronized void wear(){
        if(this.isDirty)
            System.out.println("GROSS");
        this.isDirty = true;
    }
}
public class SockDrawer{
    public List<Sock> drawer;
    public SockDrawer(){
        this.drawer = new ArrayList<>();
    }
    public Sock pickRandom(){
        synchronized(drawer){
            int choice = r.nextInt(0, drawer.size());
            return drawer.get(choice);
        }
    }
    public synchronized void putOn(Sock sock){
        sock.wear();
        drawer.remove(sock);
    }
    public void wearPair(){
        Sock sock1 = pickRandom();
        synchronized(sock1){
            findMatch(sock1);
        }
    }
    public Sock findMatch(Sock sock1){
        Sock sock2 = pickRandom();
        while(!sock2.color.equals(sock1.color) || sock1==sock2){
            sock2 = pickRandom();
        }
        putOn(sock1);
        putOn(sock2);
    }
}
```