Name: Solution

UW NetID: _____

CSE 332 Summer 2018: Final Exam Part 1

(closed book, closed notes, no calculators)

Instructions: Read the directions for each question carefully. We can only give partial credit based on the work you write down, so show your work.

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

Unless otherwise noted, any algorithms or code you write should be as efficient as possible, both in $O()$ terms and with respect to constant factors.

Take a deep breath.
Every tree is a forest.
You got this.

Good Luck!

Total: 75 points. Time: 60 minutes.

Question	Max Points	Score
1. Amdahl	5	
2. Parallel Code	15	
3. Parallel Patterns	12	
4. Concurrency	16	
5. Sorting	15	
6. True/False	12	
Day 1 Total	75	

1 Amdhal's Law

[5 points]

Use Amdahl's law to answer the following question. You must show your work for any credit. Your company has a program which is 1/5 sequential and 4/5 parallelized. At a minimum, how many processors do you need for a 4x speedup? For full credit, your answer must be a simplified fraction or an integer.

Amdahl's law says

$$\text{speedup} \leq \frac{1}{S + \frac{1-S}{P}}$$ where S is the sequential part and P is the parallel part.

plugging in values:

$$4 \leq \frac{1}{\frac{1}{5} + \frac{4/5}{P}}$$

Doing algebra:

$$4\left(\frac{1}{5} + \frac{4/5}{P}\right) \leq 1$$

$$\frac{4}{5} + \frac{16/5}{P} \leq 1$$

$$\frac{16/5}{P} \leq \frac{1}{5}$$

$$16 \leq P$$

So even with perfect speedup, we need at least 16 processors.

2 Parallel Code

[15 points]

Complete java code using the fork-join framework for the following problem:

Input: an array of strings
Output: the index of the leftmost String of even length
For example, if your input is `{ "a", "abc", "ab", "cd", "e", "defg" }` the output would be 2 (the index of `"ab"`). If there are no even length strings, -1 should be returned. `null` is not considered to have even length.

- Do **not** use a sequential cutoff. **Your base case must process one element.**
- Give a class definition `LeftmostEvenString`, **along with any other code or classes you need.**
- Fill in the function `findLeftmostEvenString` below.

You may not use any global data structures (except the input array) or synchronization primitives.

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


class Main{

    static ForkJoinPool POOL = new ForkJoinPool();
    //Returns index of leftmost String of even length
    // in the array input. Returns -1 if no strings have
    // even length. null does not have even length.
    public static int findLeftmostEvenString(String[] input){
        LeftmostEvenString mainThread =
            =new LeftmostEvenString(0, input.length, input);
        return POOL.invoke(mainThread);

    }
}
```

Fill in the function above, and write your class definition on the next page

Write your class definition on this page.

```
public class LeftmostEvenString extends RecursiveTask<Integer> {
    int lo;
    int hi;
    int[] arr;

    protected Integer compute() {
        if(lo - hi == 1){
            if( arr[lo] != null && arr[lo].length()%2==
                return lo;
            }
            else return -1;
        }

        int mid = (lo+hi) / 2;
        LeftmostEvenString left = new LeftmostEvenString(lo, mid, arr
        LeftmostEvenString right= new LeftmostEvenString(mid, hi, arr

        left.fork();
        Integer leftResult = right.compute();
        Integer rightResult = left.join();
        if(leftResult != -1)
            return leftResult;
        else
            return rightResult;
    }

    public leftmostEvenString(int L, int h, int[]a){
        lo=L;
        hi=h;
        arr=a;
    }
}
```

3 Parallel Patterns

[12 points]

Explain the steps you would use to perform the following tasks in parallel. Your algorithm should have the best possible $O()$ span, but you need not worry about constant factors in this problem. You may assume you have access to already allocated auxiliary arrays as needed, and may alter the input array. Use the following parallel code patterns discussed in class:

- `out = map(f, arr)` Applies `f` to every element of `arr`, storing the results in `out`.
- `out = reduce(baseFn, combineFn, arr)` Given `baseFn` which produces a value for a single element and `combineFn` which takes in the values for two subarrays and produces a value for the combined array, `reduce` stores the value for the full array in `out`.
- `out = parallelPrefixSum(arr)` Runs ParallelPrefixSum on `arr`, storing the results in `out`.
- `out = pack(condition,arr)` Given `condition`, performs a pack/filter on `arr`, storing the results in `out`.

When describing `f`, `baseFn`, `combineFn`, or `condition` you may assume you have access to whatever data you could access if you wrote out a full fork-join function for the associated function.

You can use whatever combination of sentences and pseudocode you prefer.

1. **Input** `arr`, an array of integers
   Output the sum of only the even numbers of the array.

onlyEven = map(f, array)
where $f(x) = \begin{cases} x & \text{if } x \text{ is even} \\ 0 & \text{if } x \text{ is odd} \end{cases}$

sum = reduce(identity, plus, onlyEven)
where identity is the function $f(x) = x$
plus is the function that adds its two inputs together.

our final answer is sum

2. **Input** arr, an array of integers
Output an array containing exactly the indices of arr such that arr[i] > $\sum_{j=0}^{i-1}$ arr[j].

SumsUpToi = ParallelPrefixSum(arr)

Note that the question asks for the sum upto i-1, but ParallelPrefixSum causes i to contain the sum up to index i.

Let Indices be an array of the same length as arr.

Indices = map(i, Indices)
where i returns the current index

result = Pack([i==0 && arr[i]>0] || [arr[i] > sumsUpToi[i-1]], Indices).

Here we're assuming $\sum_{j=0}^{-1}$ arr[j] = 0. We didn't deduct for choosing a different convention here.

3. **Input** arr, a **sorted** array of integers
Output an array containing exactly the elements that appear only once in arr in sorted order.

Note that in a sorted array, an element appears once if and only if the two adjacent elements are different from it.

So our final answer is just the result of:

```
Pack( [i==0 && arr[i] != arr[i+1]] ||
      [i== arr.length -1 && arr[i-1] != arr[i]] ||
      [arr[i] != arr[i-1] && arr[i] != arr[i+1]]
      arr )
```

4 Concurrency

[16 points]

Dumbledore needs your help again! Having heard the wonders of concurrency, he has altered the discipline review system you helped with on the midterm. Now he has one priority heap, and intends that both Professors McGonagall and Snape insert their disciplinary actions into the same heap concurrently.

He shows you his current code.

```
1
2  public class ConcurrentHeap{
3      private Discipline[] arr;
4      private int size; //current number of elements
5      private int capacity; //size of arr
6
7      /**
8       * inserts new disciplinary action d into the heap
9       */
10     public void insert(Discipline d){
11         if(size == capacity)
12             resize();
13         arr[size++] = d;
14         percolateUp(size-1);
15     }
16
17     //details omitted. Methods work as discussed in class
18     //and on earlier programming projects.
19     private void percolateUp(int ind){  /*...*/}
20     private void percolateDown(int ind) { /*...*/}
21     public Discipline removeMin() { /*...*/}
22     private void resize(/*doubles size of arr and copies over elements*/)
23     public Discipline peek() { /*...*/}
24     public int getSize(){/*...*/}
25
26 }
```

1. Show Dumbledore a bad interleaving of the code above.

Remember that "lines" of code can be multiple steps.

Thread 1 (both in insert)

(1) arr[size] = d;

(3) size ++;

Thread 2

arr [size] = d; (2)

size ++; (4)

Thread one's insert will be last.

"Ahh, I think I see the problem," Dumbledore replies. "I've heard of these things called re-entrant locks. Let me add some of them. One per professor should do the trick."
He alters the code to look like this.

```
1
2  public class ConcurrentHeap{
3      private Discipline[] arr;
4      private int size; //current number of elements
5      private int capacity; //size of arr
6      private Lock SnapeLock;
7      private Lock McGonaLock;
8      /**
9       * inserts new disciplinary action d into the heap
10      */
11     public void insert(Discipline d){
12         SnapeLock.acquire();
13         McGonaLock.acquire();
14         if(size == capacity)
15             resize();
16         arr[size++] = d;
17         percolateUp(size-1);
18         McGonaLock.release();
19         SnapeLock.release();
20     }
21
22     //details omitted.
23     //All other methods acquire SnapeLock then McGonaLock at the top of the method
24     //And release both at the bottom, in the same way insert does.
25     private void percolateUp(int ind){  /*...*/}
26     private void percolateDown(int ind) { /*...*/}
27     public Discipline removeMin() { /*...*/}
28     public Discipline peek() { /*...*/}
29     private void resize(/*doubles size of arr and copies over elements*/)
30     public int getSize(){ /*...*/ }
31 }
```

2. Does this code have a bad interleaving to produce a race condition? If so describe one. If not briefly describe why it doesn't.

No, every method that changes or reads any of the fields of the heap must acquire McGonaLock (and SnapeLock, but either is enough) before reading or writing.

3. Does this code have potential for deadlock? If so, describe an interleaving to cause it. If not briefly describe why it can't happen.

No, despite the fact that there are two locks, they are always acquired in the same order.

4. Tell Dumbledore a way to improve his code. If you found errors in the previous parts, your alterations should fix them. If you did not find errors, you should still find a way to improve his use of synchronization.

While there are no errors in the code, the use of locks can be improved. The locks are protecting the same critical sections, so it helps nothing to have both. And it adds significant complexity to maintaining the code, as every new use of the locks will need to acquire both, in the correct order.

Dumbledore should have just one lock here, protecting alterations to arr and size.

5 Sorting

[15 points; 3 points each]

For each of the following scenarios, choose one of the sorting algorithms discussed in class, and describe why you believe it is best-suited for the task.

1. You're writing code to sort numbers on the next NASA rover. Since it's going to space, you can't afford much memory, and because of limited communication windows with Earth, you want the code to run consistently quickly.

   Heapsort.

   We want worst case $O(n \lg n)$ because of the communication windows, and we want in-place because of the lack of memory.

2. You just finished sorting a large array, when another thread changes the values of a few entries in the array. You acquire a lock on the entire array and decide to sort again.

   Insertion sort

   it has $O(n)$ running time on mostly sorted data.

3. At your new job working for Nanofluff, you're asked to write a sorting algorithm to be used by their spreadsheet software Cellective. Users of Cellective enter data spanning multiple columns, and wish to sort their data in arbitrary combinations of those columns.

   Merge sort or Quick sort

   The most important thing is to choose a stable sort, as the user could want the results of one sorting operation to be incorporated into the next sort.

   The other stable sorts (radix sort & bucket sort) need extra information about the data, and spreadsheets can store data that doesn't behave well for those sorts (like doubles/floats).

4. Briefly explain what it means for a sort to be stable.

If there are two elements where neither is less than the other, the final sorted list keeps them in the same relative order as they were in the initial array.

5. Caitlin tells you she's invented a new comparison-based sorting algorithm that takes $\Theta(n)$ time on lists that have many repeated elements. Should you believe her, or is such an algorithm impossible? Briefly justify your answer (1-2 sentences).

This question was ambiguous. We threw it out and gave everyone 3/3.

If you interpreted "many repeated elements" as something like the array has $n/5$ distinct elements, each appearing 5 times, then the answer is no. That would contradict the lower bound from class.

But if you interpreted "many repeated elements" as something like $(n-\sqrt{n})$ elements are all identical, then an $\Theta(n)$ algorithm is possible (and the lower bound isn't violated because we're making a very strong assuption about the data set).

6 True/False

[12 points] For each of the following statements, say whether the statement is true or false. If it is false, explain how to make the statement correct. There may be more than one error in a false statement.

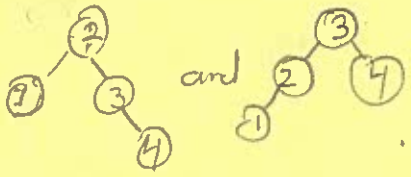
1. Regardless of algorithm, creating an AVL tree with n elements requires $\Omega(n \log n)$ time in the worst case, because most of the AVL tree nodes are in the bottom few levels, and therefore take $\Omega(\log n)$ time to put them in the tree.

The second half of the statement is not a valid proof of the first statement. For example an algorithm to build an AVL tree might sort the array then do a traversal for the insertion

To correct the statement we need to use the reduction of Exercise 7. Since we can do a traversal in $\mathcal{O}(n)$ time, building the tree must take $\Omega(n \log n)$ time

2. Building a heap can be done faster than building an AVL tree because there is only one possible AVL tree for a set of elements, but there are many possible heaps.

False: There is more than one possible AVL tree for the same elements

e.g.  and

The proper end of that sentence is that heaps don't fully sort a data set. thus avoiding the reduction of Exercise 7.

3. Parallel Quicksort (where we parallelize both the recursion and partitioning) has span $\Theta(\log^2 n)$ in the best case.

True!

4. Each step of radix sort must be in-place for the sort to produce the correct output.

False: "in-place" needs to be "stable".

Blank page for scratch work

Some Useful Facts

When we're using the tree method to solve a recurrence, we usually use the following steps:

0. Draw a few levels of the tree.
1. Let the root node be at level 0. Give a formula for the size of the input at level i.
2. What is the number of nodes at level i?
3. What is the work done at the i^{th} recursive level?
4. What is the last level of the tree?
5. What is the work done at the base case?
6. Write an expression for the total work done.
7. Simplify until you have a "closed form" (i.e. no summations or recursion).

Geometric series identities:

$$\sum_{i=0}^{k} c^i = \frac{c^{k+1}-1}{c-1} \qquad \sum_{i=0}^{\infty} c^i = \frac{1}{1-c} \text{ if } |c| < 1$$

Common Summations:

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \qquad \sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \qquad \sum_{i=0}^{n} i^3 = \frac{n^2(n+1)^2}{4}$$

Log identities:

$$a^{\log_b(c)} = c^{\log_b(a)} \qquad \log_b(a) = \frac{\log_d(a)}{\log_d(b)}$$

Master Theorem:
Given a recurrence of the following form:

$$T(n) = \begin{cases} d & \text{if } n \leq \text{ some constant} \\ aT(n/b) + n^c & \text{otherwise} \end{cases}$$

with a, b, c are constants.
If $\log_b(a) < c$ then $T(n)$ is $\Theta(n^c)$
If $\log_b(a) = c$ then $T(n)$ is $\Theta(n^c \log n)$
If $\log_b(a) > c$ then $T(n)$ is $\Theta\left(n^{\log_b(a)}\right)$