1) (8 pts) Algorithmic Complexity

a. Give the complexity for the function \( n^2 + 60 \) \( O(n^2) \)

b. Give the complexity for the function \( n \log(n) + n \) \( O(n \log(n)) \)

c. T \rightarrow Best case analysis is less useful than worst case analysis.

d. F \rightarrow Asymptotic analysis is relevant for small inputs (small values of \( n \)).

2. (3 pts) Calculate the asymptotic complexity of the code snippets below (using big-O notation) with respect to the problem size \( n \).

a. for (int i=n/2; i<=n/2; i++)
   f(n) = O(\( n^2 \))
   for (int k=0; k<(n*n); k++)
   ...  

b. for (int i=0; i<=n/2; i++)
   f(n) = O(\( n \log(n) \))
   for (int j=1; j<=n; j=j*2)
   ...  

c. for (int i=1; i<=100; i++)
   f(n) = O(1)
   ...

3. (1 pt) List the following big-O expressions in order of asymptotic complexity (with the lowest complexity first).

\[
O(n \log(n)) \quad O(\log(n)) \quad O(2^n) \quad O(n) \quad O(1)
\]

Answer: \( O(1) \quad O(n) \quad O(\log(n)) \quad O(n \log(n)) \quad O(2^n) \)

2) There is no question 2 for CMSC 132H
3) There is no question 3 for CMSC 132H
4) (10 pts) Sets and Maps

The **Phonebook** class maintains a database of phone numbers. In the database a name can be mapped to one or more phone numbers.

```java
public class Phonebook {
    public Phonebook() { /* You must implement this method */ }
    public void addEntry(String name, String phoneNumber) { /* You must implement this method */ }
    public Set<String> getPhones(String name) { /* You must implement this method */ }
}
```

**What You Must Implement**

1. (2 pts) Define any fields you need and any initialization that needs to be done when a Phonebook is constructed.

2. (5 pts) Implement the `addEntry` method that adds a phone number to the database. A map entry for the specified person must be created if one does not exist.

3. (2 pts) Implement the method `getPhones` which returns a set with the phones associated with a person. If no phones are associated with the person an empty set will be returned.

**One answer:**

```java
HashMap<String,Set<String>> db = new HashMap<String,Set<String>>();
public void addEntry(String name, String phoneNumber) {
    Set<String> phones = db.get(name);
    if (phones == null) {
        phones = new HashSet<String>();
        db.put(name, phones);
    }
    phones.add(phoneNumber);
}
public Set<String> getPhones(String name) {
    Set<String> phones = db.get(name);
    if (phones == null) {
        phones = new HashSet<String>();
        db.put(name, phones);
    }
    return phones;
}
```

**Another answer:**

```java
HashMap<String,Set<String>> db = new HashMap<String,Set<String>>();
public void addEntry(String name, String phoneNumber) {
    getPhones(name).add(phoneNumber);
}
public Set<String> getPhones(String name) {
    Set<String> phones = db.get(name);
    if (phones == null) {
        phones = new HashSet<String>();
        db.put(name, phones);
    }
    return phones;
}
```
5) (10 pts) Linear Data Structures

Implement the methods below based on the following Java class definitions. You may not add any instance variables or static variables to the LinkedList class. In addition, you may not use the Java API LinkedList class. Feel free to add any auxiliary non-static methods.

```java
public class LinkedList<T> {
    private class Node {
        private T data;
        private Node next;
        public Node(T data) {
            this.data = data;
            next = null;
        }
    }
    private Node head;

    public int countInstancesOf(T value) {
        Node curr = head;
        int count = 0;
        while (curr != null) {
            if (curr.data.equals(value))
                count++;
            curr = curr.next;
        }
        return count;
    }

    private Node removeInstancesOfAux(Node headAux, T value) {
        if (headAux == null)
            return null;
        else {
            if (headAux.data.equals(value))
                return removeInstancesOfAux(headAux.next, value);
            else {
                headAux.next = removeInstancesOfAux(headAux.next, value);
                return headAux;
            }
        }
    }
}
```

1. (4 pts) Implement a method `countInstancesOf` that returns the number of instances of the specified value. You can assume the appropriate equals method for T elements has been defined.

2. (6 pts) Implement a recursive auxiliary method `removeInstancesOfAux` that removes all value instances from the list. Notice this method is called by removeInstances. You can assume the appropriate equals method for T elements has been defined.
6) (6 pts) Heaps

Use the following heap to answer the questions that follow.

1. (1 pt) Draw the heap as an array.
4, 10, 28, 11, 19

2. (1 pt) **F** → Every heap is a binary search tree.

3. (2 pts) Draw the heap that would result from inserting 2 in the above heap.

4. (2 pts) Draw the heap that would result by deleting 4 from the original heap (not the heap from 3.)
7) (6 pts) Threads

1. (6 pts) Modify the following code (feel free to cross out and/or add) so the increments to the arrays are processed by two different threads. The main thread will print the contents of the arrays after they have been incremented.

One answer:

```java
public class AddIncrementToArray implements Runnable {
    private int[] data;
    private int increment;

    public AddIncrementToArray (int[] data, int increment) {
        this.data = data;
        this.increment = increment;
    }

    public void addIncrement() {
        for (int i=0; i < data.length; i++)
            data[i] += increment;
    }

    public void run() {
        addIncrement();
    }

    public static void main(String[] args) throws Exception {
        int[] a = {5, 10, 15};
        int[] b = {1000, 2000, 3000};

        Thread t1 = new Thread(new AddIncrementToArray(a,100));
        Thread t2 = new Thread(new AddIncrementToArray(b,200));
        t1.start();
        t2.start();

        t1.join();
        t2.join();
        for (Integer i : a)
            System.out.println(i);

        for (Integer i : b)
            System.out.println(i);
    }
}
```
8) (10 pts) Binary Trees

Use the following classes to answer the questions below. You may not add any instance variables or static variables to either class, and you may not add any methods to the Node class. Feel free to add any auxiliary non-static methods to the BinarySearchTree class. **Non-recursive solutions will receive zero credit.**

```java
public class BinarySearchTree <K extends Comparable<K>, V> {
    private class Node {
        private K key;
        private V data;
        private Node left, right;
        public Node(K key, V data) {
            this.key = key;
            this.data = data;
        }
    }
    private Node root;
    public int size() { /* YOU MUST IMPLEMENT THIS METHOD */ }
    public void add(K key, V data) { /* YOU MUST IMPLEMENT THIS METHOD */ }
}
```

1. (3 pts) Implement the method `size` which returns the number of elements in the tree.
2. (7 pts) Implement the method `add` that adds the key, data pair to the binary search tree. The data value will be updated if the key already exists in the tree.
public int size() {
    return sizeAux(root);
}

private int sizeAux(Node rootAux) {
    return rootAux==null ? 0 : sizeAux(rootAux.left) + sizeAux(rootAux.right) + 1;
}

public void add(K key, V data) {
    if (root==null)
        root = new Node(key, data);
    else
        addAux(key, data, root);
}

private void addAux(K key, V data, Node rootAux) {
    int comparison = key.compareTo(rootAux.key);
    if (comparison == 0)
        rootAux.data = data;
    else if (comparison < 0) {
        if (rootAux.left == null)
            rootAux.left = new Node(key, data);
        else
            addAuxBooolean(key, data, rootAux.left);
    } else {
        if (rootAux.right == null)
            rootAux.right = new Node(key, data);
        else
            addAuxBooolean(key, data, rootAux.right);
    }
}
10) (10 pts) Graphs

1. (4 pts) Graph traversals

   ![Graph Diagram]

   a. Give *two* different possible orders in which the nodes of this graph could be visited in performing a *Breadth First Search (BFS)* starting at vertex A.

   **Answer** Any two of: ACFED, ACFDE, AFCED, AFCDE

   **Grading** 1 pt each correct solution.

   b. Give *two* different possible orders in which the nodes of this graph could be visited in performing a *Depth First Search (DFS)* starting at vertex A.

   **Answer** ACEFD, AFDCE

   **Grading** 1 pt each correct solution

2. (6 pts) Single source shortest paths

   ![Graph Diagram]

   Apply Dijkstra’s algorithm using A as the starting (source) node. Indicate the cost and predecessor for each node in the graph after processing 2 nodes (A and another node). Remember to update the appropriate table entries after processing the 2nd node (after it has been added to the set of processed nodes). An empty table entry implies an infinite cost or no predecessor. **Note: points will be deducted if you simply fill in the entire table with the final costs and predecessors instead of showing the table at the first two steps.**
Answer/Grading
-1 for each incorrect table entry (do not exceed -6)

After processing 1 node:

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost</td>
<td>0</td>
<td>∞</td>
<td>1</td>
<td>7</td>
<td>∞</td>
</tr>
<tr>
<td>Predecessor</td>
<td>-</td>
<td>-</td>
<td>A</td>
<td>A</td>
<td>-</td>
</tr>
</tbody>
</table>

After processing 2 nodes:

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>Predecessor</td>
<td>-</td>
<td>C</td>
<td>A</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

11) There is no question 11 for CMSC 132H
12) There is no question 12 for CMSC 132H
13) (13 pts) Honors Sorting

1. T \rightarrow An external sorting algorithm does not keep all the keys in memory.
2. T \rightarrow The asymptotic complexity of heapsort is $O(n \log(n))$.
3. T \rightarrow In a stable sorting algorithm the relative order of equal keys is unchanged.

4. What is the proven lower bound for a comparison sort?
   a. $O(\log(n))$
   b. $O(n \log(n))$
   c. $O(n)$
   d. None of the above

5. To implement a parallel sort, which of the basic sorting algorithms that we discussed would be most appropriate as a basis for the parallel sort?
   
   Answer: mergesort, since it divides the problem into two equal size subproblems

6. Is quicksort a stable sort?
   
   Answer: No

7. Is mergesort a stable sort?
   
   Answer: yes

8. Describe, at a high level, how quicksort works. What recursive call(s) are made, and what computations are performed before and/or after the recursive call(s)?
   
   Answer:
   1) Pick a partition element from the array (e.g., first element, random element, median of 3 elements)
   2) Partition the elements into three sections: elements less than or equal to the partition element, the partition element, and elements greater than the partition element.
   3) Recursively sort the first and last sections

9. Describe, at a high level, how mergesort works. What recursive call(s) are made, and what computations are performed before and/or after the recursive call(s)?
   
   Answer:
   1) Sort the first half and the second half of the array
   2) Merge the two sorted halves of the array into a temporary array (iteratively copy the smaller of the elements at the end of the unmerged halves)
   3) Copy back from the temporary array into the array to be sorted
14) (12 pts) Honors Recursion

Use recursion to solve the partition problem: Given a vector of n ints, it is possible to partition them into two groups/multisets such that the sum of the ints in each group are equal (this isn’t the same partition problem we discussed earlier in class in another context). For example, \{3, 3, 5, 9, 2\} can be partitioned into \{3, 3, 5\} and \{9, 2\} (both sum to 11), but \{1, 5, 16\} cannot be partitioned. Note that the groups do not have to contain the same number of integers, and both the original vector and the two groups can contain duplicates.

1. What is the asymptotic running time of your implementation below?

    \(O(2^n)\)

2. Provide your implementation of boolean `partition(int [] values)`, and any auxiliary recursive functions, below.

```java
boolean partition(int [] values) {
    return partition(values, 0, 0);
}

boolean partition(int[] values, int pos, int sum) {
    if (pos >= values.length)
        return sum == 0;
    if ( partition(values, pos+1, sum+values[pos]))
        return true;
    return partition(values, pos+1, sum-values[pos]);
}
```
15) **(15 pts) Honors Topics**

1. **F →** DNS is used to determine which port to connect to

2. Provide a regular expression that matches strings of a’s and b’s that ends in an a (e.g., matches “aba”, “a”, “baa” but not “ab” or “b”)

   **Answer:** (a|b)*a

3. Provide a regular expression that matches strings of a’s and b’s such that any sequence of a’s is at least two characters long (e.g., matches “aab”, “baaab”, “aa”, “baabaaa”, but not “a”, “aba” or “baba”).

   **Answer:** (aa|b)*

4. What is a strongly connected component (SCC) of a graph?

   **Answer:** a set of vertices in a directed graph such that from any vertex in the SCC, you can get to any other vertex in the SCC, but there are no other vertices in the graph that can also be part of the SCC (e.g., the set is maximal).

5. Briefly describe what a union-find data structure is, what operations it supports, and an example of an application or algorithm in which it is used.

   **Answer:** Union-find starts with a set of elements, each in its own group, and supports two operations: determining if two elements are in the same group, and merging the groups containing two elements. It is used in Kruskal’s minimum spanning tree algorithm to determine whether an edge connects two subtrees that are already connected.
6. In building a Huffman tree, in each step, two trees are combined. How do you decide which two trees to combine?

**Answer:** Choose the two trees that have the smallest frequencies (where the total number of all occurrences of elements in the tree is smallest).

7. Pick one of the following concurrent abstractions: CountDownLatch, Semaphore, or BlockingQueue: Briefly describe it, give an example of a situation that would cause a call to the abstraction to block, and what would need to happen for that call to become unblocked.

**Answers** (only one expected):

CountDownLatch: an nonnegative integer counter that be decremented (never incremented) and that you can wait for it to get to zero. If you make a call to wait for it to be zero, and it isn’t zero, the call will block until some other thread(s) decrement it all the way down to zero.

Semaphore: An integer counter that represents a number of permits. You can either ask for or return a number of permits. Asking for permits decrements the permit count, but never allows the number of permits stored in the semaphore to go negative. Returning permits increments the permit count. If you ask for more permits than the semaphore has, you will block until enough permits are added by other threads that you can get the permits you asked for.

BlockingQueue: A queue for elements, with operations such as adding and removing elements. There are blocking operations that ask for an element and if none are available, block until an element is available. If such a call blocks, you can be unblocked when another thread adds an element to the queue, allowing your call to remove one.

8. A coworker has asked for help with his code. He had a class with only one field, a HashMap. He wanted to make his class work with multiple threads, so he just changed the HashMap to a ConcurrentHashMap. He says it now works most of the time with multiple threads, but he occasionally gets weird errors. Without looking at his code, (briefly) describe the most likely problem in his code and what he will need to do to fix it.

**Answer:** The likely problem is a lack of atomicity: One of more of his methods is performing multiple operations on the concurrent hashmap (such as a get followed by a put), and expecting that those will be performed atomically (i.e., without any intervening operations by another thread). One possible solution is to change his code to use methods such as putIfAbsent that combine several operations into one atomic operation. Another possibility is to just use a normal hashmap, and synchronize the methods.