The exam is closed book and closed notes. Do not begin until instructed.

You have **90 minutes**. Good luck!

Write your name and Cornell **NetID** at the top of EACH page! There are 6 questions on 8 numbered pages, front and back. Check that you have all the pages. When you hand in your exam, make sure your booklet is still stapled together. If not, please use our stapler to reattach all your pages!

We have scrap paper available. If you do a lot of crossing out and rewriting, you might want to write code on scrap paper first and then copy it to the exam, so that we can make sense of what you handed in.

Write your answers in the space provided. Ambiguous answers will be considered incorrect. You should be able to fit your answers easily into the space provided.

In some places, we have abbreviated or condensed code to reduce the number of pages that must be printed for the exam. In others, code has been obfuscated to make the problem more difficult. This does not mean that its good style.
1. **True / False** (20 points)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>a)</strong></td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td><strong>b)</strong></td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td><strong>c)</strong></td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td><strong>d)</strong></td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td><strong>e)</strong></td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td><strong>f)</strong></td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td><strong>g)</strong></td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td><strong>h)</strong></td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td><strong>i)</strong></td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td><strong>j)</strong></td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td><strong>k)</strong></td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td><strong>l)</strong></td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td><strong>m)</strong></td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td><strong>n)</strong></td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td><strong>o)</strong></td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td><strong>p)</strong></td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td><strong>q)</strong></td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td><strong>r)</strong></td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td><strong>s)</strong></td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td><strong>t)</strong></td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>
2. Short Answer (15 points)

2.a Hashing (9 points)

Consider implementing a set using hashing with linear probing. Assume an array of 6 elements of class \texttt{Integer}, as shown below. We define the hash function \( f(i) = (2 * i) \mod 6 \), where 6 is the table size. For example, hashing 4 gives 8 mod 6, which is 2.

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
null & 6 & 4 & 1 & null & null \\
\end{array}
\]

(i) 5 points Insert the values 4, 1, 0, 1, and 6 into the set, in that order —write the values in the appropriate element in the table above, crossing off the value currently in that element. Do not re-size the array, even though this is the standard way to implement a hash table.

(ii) 2 points Now remove the value 1 from the set. Do you simply set the array element to null? Explain why or why not.

No. We set the value of \texttt{isInSet} to false. If we simply set the array element to null, then the linear probing algorithm might stop searching for an element before it should.

(iii) 2 points Consider the insertion of values in point (i) above. Which insertion (if any) caused the load factor to surpass 0.25?

The second insertion (the first “1” in the list of values above) into the set changed the load factor to 2/6, or .333.

2.b Exception Handling 6 points

(i) 2 points Consider the statement below, appearing in a method \texttt{m}, where \texttt{b} is an int array. Does its execution result in a \texttt{RuntimeException} being thrown out to the call of \texttt{m}? Write your answer and an explanation for it to the right of the statement.

```java
try {
    int x = b[-5];
} catch (Exception e) {
    throw new RuntimeException();
}
```

Yes. The line \texttt{throw new RuntimeException();} in the catch block above throws a \texttt{RuntimeException} that is not caught in \texttt{m}.
(ii) **4 points** To the right below, write down what is printed by the `println` statements during execution of the call `mm(1)`, where method `mm()` is defined as follows:

```java
public static void mm(int x) {
    try {
        System.out.println("11");
        int b= 5/(x-1);
        System.out.println("12");
        return;
    } catch (RuntimeException e) {
        System.out.println("13");
        int c= 5/(x-1);
        System.out.println("14");
    }
    System.out.println("15");
    int d= 5/x;
    System.out.println("16");
    return;
}
```

11
13 The division in the catch block throws an exception, which is not caught.

3. **Complexity** (13 points)

(a) **4 points** For each of the functions `f` below, state the function `g(n)` such that `f(n)` is \(O(g(n))\). `g(n)` should be as small as possible. (e.g. one could say that \(f(n) = 2n^2\) is \(O(n^3)\), but the best answer, the one we want, is \(O(n^2)\).)

(i) \(f(n) = \log(n) + n + n^3\). \(g(n) = n^3\)

(ii) \(f(n) = 2n + \frac{3000}{n} + 42\). \(g(n) = n\)

(iii) \(f(n) = 2^{n+12} + 400n^4\). \(g(n) = 2^n\)

(iv) \(f(n) = 55\) \(g(n) = 1\)

(b) **3 points** State the tightest (smallest) asymptotic time complexity (in terms of \(n\)) of the following statement sequence:

```java
int s= 0;
for (int k= 0; k < n; k= k+1) {
    for (int j= k+5; j > k; j= j-1) {
        s= s + j*k;
    }
}
```

\(O(n)\). The inner loop iterates no more than 5 times, so it is \(O(1)\).
(c) **6 points**  Give a formal proof that \( f(n) = 3000 + 2n \) is \( O(n) \).

\[
\begin{align*}
f(n) &= \text{<definition of } f> \\ &= 3000 + 2n \\ &\leq \text{<for } n \geq 1> \\ &= 3000n + 2n \\ &= \text{<arithmetic> } \\ &= 3002n
\end{align*}
\]
So, select \( c = 3002, N = 1 \)

4. **Searching, Sorting, and Invariants** (14 points)

(a) **6 points**  Assume that this procedure has already been written:

```java
/** Merge sorted segments b[h..j] and b[j+1..k] so that b[h..k] is sorted. */
public static void merge(int[] b, int h, int j, int k) {...}
```

Below, write the body of procedure \( MS \), completely in Java.

**Solution**

```java
/** Sort b[h..k], using the mergesort algorithm. */
public static void MS(int[] b, int h, int k) {
    if (h >= k) return; // Segments of size \( \leq 1 \) are in sorted order
    int m = (h + k) / 2;
    MS(b, h, m);
    MS(b, m + 1, k);
    merge(b, h, m, k);
}
```

(b) **8 points**  Below is the precondition and postcondition of an algorithm to swap the negative values in \( b[h..k] \) into the beginning of \( b[h..k] \). Note that \( h \) is not necessarily 0 and \( k \) is not necessarily \( b.length-1 \). Do not change \( h \) and \( k \). You may assume the following procedure has already been written:

```java
/** Swap b[i] and b[j]. */
public static void swap(int[] b, int i, int j) {...}
```

Complete procedure \( p \) below using a loop that follows the given loop invariant to accomplish this task.

<table>
<thead>
<tr>
<th>Precondition:</th>
<th>Postcondition:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b )</td>
<td>( b )</td>
</tr>
<tr>
<td>( h )</td>
<td>( k )</td>
</tr>
<tr>
<td>( h )</td>
<td>( k )</td>
</tr>
<tr>
<td>( h )</td>
<td>&lt; 0</td>
</tr>
<tr>
<td>( h )</td>
<td>( \geq 0 )</td>
</tr>
</tbody>
</table>

5 of 8
Solution

/** Modify b[h..k] as defined above. */
public static void p(int[] b, int h, int k) {
    int e = k + 1; OR int f = k;
    int f = k; OR int e = f + 1;
    while (e != h) { // 2 pts for init making inv true
        if (b[e - 1] < 0) { // 2 pts for inv && !cond implies result
            e = e - 1; // 2 pts for body making progress
        } else { // 2 pts for maintaining inv
            swap(b, e - 1, f); // pts could be deducted for doing
            e = e - 1; // what should not be done
            f = f - 1; // e.g. changing h and k, starting a 0
        }
    }
}

5. Trees (17 points)

5.a Binary Search Trees (9 points)

Assume that each node of a binary search tree (BST), of class Node, has these fields:
- Node left: the left subtree (null if empty)
- Node right: the right subtree (null if empty)
- int data: the data of the node

Write the body of the following method:
Solution

/** Return true if v is in tree t and false otherwise.
 * Takes O(d) time, where d is the maximum depth of t
 * Precondition t is a BST -- t being null means an empty tree */
public static boolean contains(Node t, int v) {
    if (t == null) return false;
    if (v == t.data) return true;
    if (v < t.data) return t.left.contains(v);
    return t.right.contains(v);
}

5.b Heaps (8 points)

Consider writing heapsort to sort an int array b. Complete the implementation of step (2) in
the method below. You do not have to concern yourself with the implementation of step (1).
In implementing step (2):

Should be contains(t.left, v)
Should be contains(t.right, v)
• Remember that $b[0]$ is the root of the heap and is the largest value (it is a max-heap).
• Assume that function int poll(int[] b, int k) has already been written and can be used. It assumes that $b[0..k-1]$ is a heap, removes the root, does what is necessary to make $b[0..k-2]$ back into a heap, and returns the removed value.

Solution

```java
/** Sort array b using heapsort */
public static void heapsort(int[] b) {
    // (1) Make b[0..b.length-1] into a max-heap (largest value ends up in b[0])
    heapify(b);

    // (2) Poll values from heap and put them into their sorted position in b
    for (int i = b.length - 1; i >= 0; i--) {
        b[i] = poll(b, i + 1);
    }
}
```

6. **Graphs** (21 points)

(a) 3 points  There are two basic ways to implement a graph: (1) an adjacency matrix and (2) an adjacency list. Let a graph have $n$ nodes and $e$ edges. Below, for each point, state to the right which representation of a graph has that property:

(i) Uses space $O(n + e)$: **Adjacency list**
(ii) Takes time $O(n)$ to iterate over the edges that leave given node $n$: **Adjacency matrix**
(iii) Takes time $O(1)$ to determine whether there is an edge from node $n_1$ to node $n_2$: **Adjacency matrix**

(b) 5 points  For the graph below, give a list of the nodes that are visited by the breadth-first search algorithm starting at node A, in the order visited. Whenever there is a choice of processing nodes in any way, process them in alphabetical order by their names. Example: to do something with nodes G, A, D, first do A, then D, and then G.

```
A --- B --- C
    |       |
    |       |
    F --- D
```

A, B, C, E, D, G, F

(c) 3 points  Is the following graph a planar graph? Write yes or no to its right.
Yes. Bend each of the two long edges that cross another edge out around the node so it does not cross the other edge.

(d) 3 points State the difference between Kruskal’s algorithm and Prim’s algorithm for constructing a spanning tree of an undirected graph.
Kruskal’s algorithm selects the smallest edge that does not introduce a cycle and adds it to the tree at each iteration. Prim’s algorithm starts with a specific node and selects the smallest edge leaving the connected component being built at each iteration.

(e) 7 points Complete recursive algorithm dfs, given below. Do not be concerned about how marking occurs. You may simply say “mark n” and “if n is unmarked”.
Solution

/** Mark all nodes that are reachable along unmarked paths from n. *
 * Precondition: n is unmarked. */
public static void dfs(Node n) {
    mark n;
    for (Node v : n.neighbors) {// You could other ways to describe neighbors.
        if (v is unmarked) { // We were not particular.
            dfs(v);
        }
    }
}