1. (20%) Runtime analysis. Given the following piece of code:

```java
public static void mystery(int n)
{
    if (n==1) {
        System.out.println("Something");
    }
    else {
        System.out.println("Something");
        mystery(n/2);
    }
}
```

Let $T(n)$ be the running time of mystery in terms of its argument $n$.

(a) (5%) write the $T(n)$ formula as shown in class and explain briefly.

$$T(n) = T(n/2) + C$$ where $C$ is a constant. The program makes a recursive call of size $n/2$ and a constant number of operations (if statement + boundary condition).

(b) (5%) What is the runtime of this function based on the $T(n)$ formula you derived? The runtime is $O(\log n)$, similar to binary search we saw in class - every call cuts the input size by half and, so $O(\log n)$ calls at most are needed, each one does a constant operation.

(c) (5%) Exactly how many times will “Something” be printed for $n = 7$?

Three times. One for $n = 7$, then for $n = 3$ and then for $n = 1$ (remember that integer division rounds down).

(d) (5%) This part is unrelated to (a-c) above. A given algorithm runs as $O(2^n)$ where $n$ is the input size. If the algorithm takes 10 seconds to run on an input of size 5, what is the input size that makes the algorithm run for approximately 40 seconds? Explain.

The answer is 7. If $O(2^5) \approx 10$ seconds, then multiplying both sides by 4 gives us $4 \times O(2^5) = O(2^7) \approx 40$. Another way to look at it: If the runtime is $O(2^n)$, then adding 1 to the input size doubles the runtime, and adding 2 increases the runtime 4-fold, hence 7.

2. (25%) Java implementation and collections: For each of the following statements say whether it is possible or not using only the API collections we learned in class. Explain each answer briefly.
(a) (5%) A program has set up a Map from String to Integer, i.e. Map<String, Integer>. Now it wants to make two different keys, “x” and “y” map to the same Integer 6.

Yes, this is ok. Values can be duplicated.

(b) (5%) A program has set up a Map from String to Integer. Now it wants to set up the inverse map from Integer to String. Can it just allocate a new Map<Integer,String> get all the HashEntries from the original map, reverse the role of key and value and insert it to the new map, such that all the original data is retained?

No, because some values may be duplicated.

(c) (5%) A program has set up a List of integers, List<Integer>, and added several elements to it. Now it wants to determine the smallest number in less than O(n) time.

No, lists are not sorted by value.

(d) (5%) Same as (c) above only using a HashSet<Integer>.

No, HashSets are not sorted.

(e) (5%) Same as (c) above only using a TreeSet<Integer>.

Yes, TreeSets are sorted.

3. Hash tables (20%): In each one of the following questions provide a brief explanation (no more than a sentence or two).

(a) (7%) Let H be a hash table where collisions are handled by separate chaining. Re-hashing is used whenever the load factor (ratio of items in the table and the size of the table) exceeds $\frac{1}{2}$. Assume that the initial size of H is 2 and re-hashing doubles the size of the table. After inserting 10 items with different keys, what is the size of H? The size is 32. Let’s trace the inserts: After the first two inserts we have to re-hash to 4 (two and not one because the load factor has to strictly exceed $\frac{1}{2}$). After one more insert we have three items and re-hash again to 8. After two more inserts we have five items and rehash to 16. After four more inserts we have 9 items and rehash to 32. Insert item 10 and we’re done.

(b) (5%) Consider an initially empty hash table H. In the worst case scenario, what is the time complexity (big-Oh) to insert n keys into the table if separate chaining is used to resolve collisions? Suppose that each entry stores a linked list, and when adding a new element to an unordered linked list, such an element is inserted in the beginning of the list. Every insert takes exactly O(1) if we insert in the beginning of the list, because we need constant time to find the entry and constant time to insert in the beginning of a list. A total of n insert gives us O(n) worst case run time.

(c) (5%) In the hash table in (b) above, what would be the worst case run time if we maintained every entry as a sorted data structure, say a TreeSet (the sorting is by key)?

In the worst case scenario all the items go to the same slot, so we have to create and maintain a sorted set of n keys, overall O(n log n) inserts.

4. Graphs (45%):

(a) (13%) There are two types of professional wrestlers: "babyfaces" ("good guys") and "heels" ("bad guys"). Between any pair of professional wrestlers, there may or may not be a rivalry. Suppose we have n professional wrestlers and we have a list of r pairs of wrestlers for which there are rivalries. Give an O(n + r) time algorithm that determines whether it is possible to designate some of the wrestlers as babyfaces and the remainder as heels such that each rivalry is between a babyface and a heel. If it is possible to perform such a designation,
your algorithm should produce it. You may re-use any algorithm we learned in class as-is (without re-writing all the stages).

Every wrestler is a node in a graph and every rivalry is an edge. In this case, just see whether the graph is bipartite.

(b) (12%) Trace the run of Depth-First search (DFS) algorithm starting from $v_1$ in the graph below. For tracing, use the same notation as in the class notes and HW3 solutions. In case of multiple options follow numerical order. Mark an * near the edges that participate in the final tree.

(c) (10%) Show how to use a graph traversal algorithm to detect all the nodes that are on a given level $i$ in a tree. A level is defined as the number of edges separating a node from the root, so that the root is at level 0.

Simply run BFS from the root and extract all nodes of distance $i$.

(d) (10%) DAGs, strong connectivity: Show how to find the set of strongly connected components in a DAG. (Hint: You can use the algorithm shown in class, but if you think about it – it’s much simpler).

This is a semi-trick question. In a DAG every vertex is its own SCC (why?), so you don’t really need to do anything.