1. **True or False**

For each statement below, state that its true or false. Explain.

(a) Binary search in a sorted array is O(n).

True. Binary search is O(logn), which is upperbounded by O(n)

(b) Master Theorem can always be used to find the big-Theta of any recurrence.

False. Master Theorem only works for a specific form of recurrence.

(c) Any Java object can be hashed.

True. hashCode() is a method in Java's Object class.

(d) The number of collisions in a hash table is dependent on the table capacity and the hash function.

True. We use the hash function to hash the object then mod that by table size to get the index

(e) Heaps can be implemented as an array with no gaps at each index.

True. Heaps are complete

(f) Binary Search Trees can be implemented as an array with no gaps at each index.

False. BSTs might not be complete.

(g) Binary Search Trees have a better big-O runtime comepared to AVL trees because AVL trees are bushier.

False. It is because AVL is bushier that it has a better runtime.

(h) AVL trees and BSTs have the same big-O runtime for inserting elements.

False. AVL insert is O(logn) and BST insert is O(n) worst case.

(i) 3-Heaps have better tight big-O runtime for insert than 2-heaps or 4 heaps.

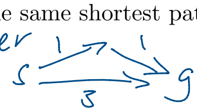
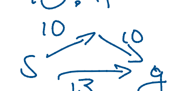
False. All are O(logn)

(j) AVL trees, BSTs, and Heaps are all DAGs.

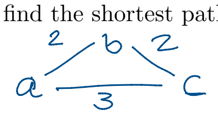
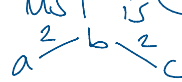
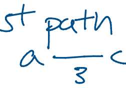
* Note: Heaps may not be DAGs depending on implementation. If it is implemented as an array it is technically not directed. Here, it is refered to as a general concept, which usually is a tree.

True. Trees are all special cases of graphs. They also are directed and have no cycles, so they are DAGs.

(k) For a graph with negative edge weights, adding a constant to make all edge weights positive and then running Dijkstra's will give the same shortest path from a node s to g as the one with negative edge weights.

False. consider  is in a graph that has an edge of −10. If we add 10 to all the edges we get  so the shortest path is different

(l) MSTs cannot be used to find the shortest path between two nodes.

True. Consider . The MST is  but the shortest path from a to c is 

(m) Graphs can be implemented with a Dictionary.

True. Dictionaries represent adjacency lists.

2. **ADTs and Implementations**

For each of the following ADTs, list the implementations that we covered in class and an advantage of each other the others. (Hint: in what situations would you use each one?)

(a) List

ArrayList → spatial and temporal locality
→ get(index) and set(index, element) is O(1)
→ add() at end is amortized O(1)
→ easy to implement

LinkedList → add and remove at front (and at back if there is a back pointer → specify on exam if there is) is O(1)

DoubleLinkedList → add and remove at both ends is O(1)

both → no empty space wasted
→ easy to rearrange elements
→ no resizing needed

(b) Dictionary/Map ← consists of 2 types of data structures. 1 for keys 1 for values

ArrayDictionary → spatial and temporal locality
→ easy to iterate through all keys
→ generally sorted order. (depends on implementation)

HashDictionary → O(1) lookup if few collisions

*BSTDictionary → sorted order
→ easier to implement

*AVLDictionary → sorted order
→ O(logn) lookup
→ no resize
→ faster than Hash tables worst case

keys do not need to be hashable but needs to be comparable

* see section 4 worksheet #7ab
* lookup = containsKey, put, remove

3. **Sorting Design Question**

For each situation below, choose an appropriate sorting algorithm from the list below. Explain your answer.

Selection Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort

(a) At class pictures the photographers usually sort the children by height before assigning places. The photographer doesn't care who was originally where in line but he/she does need to get them sorted as fast as possible and there is a lot of extra space in the room to arrange them.

not stable

not in-place

Any sort that is O(nlogn) worst case is good. Quick Sort is the fastest in practice, so this one is probably best.

(b) A company has lists of numbers that each need to be sorted. Because the lists can be long, short, reversed, in-order or a mix of all these situations, they need a sort that will always have the same tight big-O runtime regardless of the condition of the list.

A sort with a consistent runtime is best. MergeSort or HeapSort is always O(nlogn). Selection Sort is always $O(n^2)$ but since it is slow, we probably don't want to use this one

(c) When you were younger, you most likely had a box of Crayola crayons. When you first buy them however, all the colors are not sorted in order. Which sort can you use to sort the crayons by color in the box such that you only take one or two of the crayons out of the box at any given time?

in-place

Any in-place sort works.

Insertion Sort, In-place HeapSort, In-place Quick Sort.

(d) A customer requires that you use a sort that has recursion. Which one can you choose? Explain why this request is not a good idea.

MergeSort and QuickSort are the only two sorts with recursion. If the input is very large, we can run out of memory in the stack because there are too many recursive calls.

4.

```java
public class Friend implements Comparable<Friend> {
    private String name;

    public Friend(String name) {
        this.name = "brian" + name;
    }

    // assume this method runs in O(name.length()) runtime
    public boolean equals(Object o) {
        if (!(o instanceof Friend)) {
            return false;
        }

        Friend other = (Friend) o;
        return this.name.equals(other.name);
    }

    public int hashCode() {
        boolean hasLowercase = false;
        for (int i = 0; i < name.length(); i++) {
            if (name.charAt(i) >= 'a') {
                return 0;
            }
        }
        return 1;

    }

    public int compareTo(Friend other) {
        int sizeThis = this.name.length();
        int sizeOther = other.name.length();
        int index = 0;
        while (index < sizeThis && index < sizeOther) {
            if (this.name.charAt(index) < other.name.charAt(index)) {
                return -1;
            } else if (this.name.charAt(index) > other.name.charAt(index)) {
                return 1;
            } else {
                index++;
            }
        }
        return sizeThis - sizeOther;

    }

    public String toString() {
        return this.name;
    }
}
```

Warm-up:

(a) Give the best case runtime and the worst case runtime for the hashCode method for the Friend class as a simplified Tight O bound, where m is the length of the Friend's name. The .length() and .charAt method run in constant time. Worst case: O(m) Best case O(1)

(b) Give the best case runtime and the worst case runtime for the compareTo method for the Friend class as a simplified Tight O bound, where m1 is the length of this Friend's name and m2 is the length of the other Friend's name. The .length() and .charAt method run in constant time.

Worst case: O(min(m1, m2)), best case O(1)

```java
public static IDictionary<Friend, IList<Friend>> simulateFriendshipGraph(DoubleLinkedList<String> names,
     int numFriends) {
    IDictionary<Friend, IList<Friend>> graph = new ChainedHashDictionary<>();
    IDictionary<String, Friend> stringToFriend = new ChainedHashDictionary<>();

    for (int i = 0; i < names.size(); i++) {
        String name = names.get(i);
        Friend friend = new Friend(name);
        stringToFriend.put(name, friend);
    }

    Iterator<String> itr = names.iterator();
    for (int i = 0; i < names.size(); i++) {
        Friend friend = stringToFriend.get(itr.next());
        IList<Friend> neighbors = new DoubleLinkedList<>();
        graph.put(friend, neighbors);
    }

    for (KVPair<String, Friend> pair : stringToFriend) {
        String originalName = pair.getKey();
        Friend friend = pair.getValue();
        names.delete(names.indexOf(originalName));
        IList<Friend> neighbors = graph.get(friend);
        IList<String> removed = new DoubleLinkedList<>();
        for (int j = 0; j < numFriends; j++) {
            // generates a random number from 0 to names.size() - 1 inclusive on both ends in constant time
            int randomIndex = new Random(10).nextInt(names.size());

            String name = names.get(randomIndex);
            neighbors.add(stringToFriend.get(name));
            removed.add(name);
            names.delete(randomIndex);
        }

        while (!removed.isEmpty()) {
            names.add(removed.remove());
        }
        names.add(originalName);


    }
    return graph;
}
```

You can assume that line 19, 20, and 26 all run in constant time. Assume that each String in the names parameter is of length m. You can assume that numFriends will be less than names.size() so the code works. All the runtimes should be in terms of constants, m, numFriends, and n (the .size() of names).

(a) For the for loop from lines 5 to 9

i. list any method calls that are n runtime or a bigger complexity class get(i)

ii. what is the worst case runtime (consider CHD to use separate chaining and appropriate resizing strategies) for the loop from lines 5 to 9? Give your answer as a simplified, tight big Oh bound. $O(n^2 + nm)$

(b) For the for loop from lines 12 to 16

i. What is maximum number of —V—, the number of vertices in the graph? n

ii. list any method calls that are n runtime or a bigger complexity class graph.put(friend, neighbors)

iii. what is the worst case runtime (consider CHD to use separate chaining and appropriate resizing strategies) for the loop from lines 12 to 16? Give your answer as a simplified, tight big Oh bound. $O(n^2 + nm)$

(c) For the loop from lines 18 to 32

i. list any method calls that are n runtime or a bigger complexity class (highlighted in code)

ii. how many times (maximum) does the loop from lines 18 to 32 run? n

iii. how many times (maximum) is the line 31 executed? n * numFriends

.

(d) What is the overall worst case runtime of this method? Give your answer as a simplified, tight O bound.

$O(\text{numFriends} * n * (n+m))$

(e) About the behavior of the method:

i. What is this code doing? (in 1 sentence) Pointing a friend to numFriends different random people

ii. Is the graph directed or undirected? Directed → think HW7

iii. Is the graph weighted or unweighted? Unweighted

iv. Is the names parameter mutated or not mutated after the method is over? mutated not in same order

```java
public static void printRandomReachingFriendship(IDictionary<Friend, IList<Friend>> graph) {
    IList<Friend> list = new DoubleLinkedList<>();
    IPriorityQueue<Friend> heap = new ArrayHeap<>();
    for (KVPair<Friend, IList<Friend>> pair : graph) {
        heap.add(pair.getKey());
    }
    /*while (!heap.isEmpty()) {
        list.add(heap.removeMin());
    }*/

    Friend start = heap.removeMin();
    ISet<Friend> included = new ChainedHashSet<>();
    DoubleLinkedList<Friend> toProcess = new DoubleLinkedList<>();
    toProcess.add(start);
    while (!toProcess.isEmpty()) {
        Friend next = toProcess.get(0);
        toProcess.delete(0);
        System.out.println("next person: " + next);
        for (Friend neighbor : graph.get(next)) {
            if (!included.contains(neighbor)) {
                included.add(neighbor);
                toProcess.add(neighbor);
            }
            mysterySort(toProcess, 0, toProcess.size() - 1);
        }
    }
}

public static void mysterySort(DoubleLinkedList<Friend> list, int low, int high) {
    if (low < high) {
       /* pi is partitioning index, list[pi] is
         now at right place */
        int pi = partition(list, low, high);
        // Recursively sort elements before
        // partition and after partition
        mysterySort(list, low, pi-1);
        mysterySort(list, pi+1, high);
    }
}

public static int partition(DoubleLinkedList<Friend> list, int low, int high) {
    Friend pivot = list.get(low);
    int i = low - 1;
    int j = high + 1;
    while (true) {
        do {
            i++;
        } while (list.get(i).compareTo(pivot) < 0);

        do {
            j--;
        } while (list.get(j).compareTo(pivot) > 0);

        if (i >= j) {
            return j;
        }

        // swap list.get(i) and list.get(j)
        Friend temp = list.get(i);
        list.set(i, list.get(j));
        list.set(j, temp);
    }
}
```

You can assume that System.out.println runs in constant time. Also assume this method is passed a graph that was constructed by the method on the previous page. For simplicity you can assume that compareTo and hashCode take constant time, but if you want to challenge yourself try to compute the runtimes considering what you gave earlier for the worst/base case runtimes. All your answers for this part should be defined in terms of $|V|$, $|E|$, where they represent the total number of vertices and total number of edges in the graph parameter, and numFriends.

(a) For the loop on lines 5-7 give a simplified tight O bound for:

   i. the best case runtime V

   ii. the worst case runtime V * log(V)

(b) For the loop from lines 16 to 27:

   Worst: V
   Best: Numfriends

   i. how many times are lines 17-19 executed in the worst case? Bonus question: best case?

   ii. how many times max will mysterySort get called in the worst case? E times

   iii. how many times are lines 22 and 23 executed in the worst case? V times

   iv. list any method calls that are V or E runtime or bigger complexity class (highlighted)

   v. What is the worst case runtime for the loop from 16 to 28? Give your answer as a simplified tight Oh. For now, say the runtime of mysterySort is S runtime. $O(V^2 + EVS)$

(c) for mysterySort:

   i. What sort does mysterySort look the most like? Quicksort

   ii. Imagine the first time partition is called, and the difference between low and high is the size of the list parameter, called p. What is the worst case runtime for partition in terms of p? p^2 runtime

   iii. What is the worst case (describe it, not the runtime) for mysterySort? Hint: model the code as a recurrence / consider what sort you think this is and the worst case for this sort? pivot is largest or smallest

   iv. What is the best case (describe it, not the runtime) for mysterySort? an even split for Quicksort

   v. How many times will we recurse in the worst case of mysterySort (aka how many times do we call partition)? Assume that the size of the list parameter is called p. p times

   vi. Putting that together, what's the worst case runtime for mysterySort? Assume that the size of the list parameter is called p. p^3

(d) Is the way mysterySort used above more like the best case or the worst case runtime situation? worst case

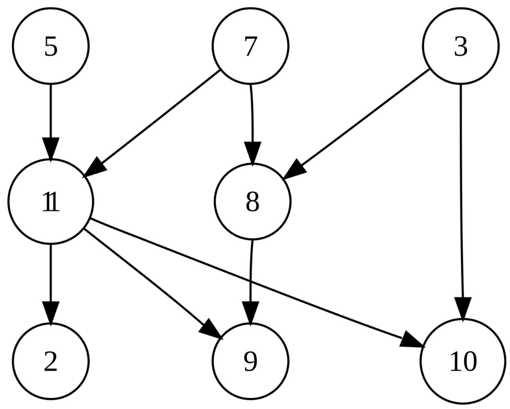
(e) What is the complexity class of the max size of toProcess? V

(f) Combine your answers above to substitute in for the S runtime you used for mysterySort earlier, and provide a new simplified, tight O bound for the runtime of printRandomReachingFriendship.

$$O(V^2 + EV^4) \longrightarrow O(EV^4)$$

5. **Topo Sort**

Consider the graph below:



(a) What are the characteristics of this graph?

Directed
Acyclic

(b) Give two possible orderings produced using Topological sort.

5, 7, 3, 11, 8, 2, 9, 10

3, 7, 5, 8, 11, 10, 2, 9

(c) Which graph algorithm can you modify to run Topo Sort?

DFS. The nodes where the DFS enters the base case are the last nodes in the topological sort (i.e. they have no children / out degree = 0). As you return up the recursive levels, the nodes are in topological order. Repeat DFS until all nodes are processed.

* See Wikipedia Topological Sort for psuedocode