1. **General: True or False**[1]
   For each statement below, state that its true or false. Explain.

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

   True - the worst case runtime for binary search in a sorted array takes log(n) time, which is a function in O(n).

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

   False - if your recurrence is not of the form $aT(n/b)+$ some n^c function, master theorem does not apply.

   (c) To implement a dictionary, you should always use a hash table over an AVL tree if your main priority is speed.

   False - if nebulous "speed" is your priority, AVL can be faster in the worst case or if you want your keys to be sorted.

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

   False - the keys and collision resolution strategies also affect how many collisions there are.

   (e) The specific order that values are inserted into a heap will affect both the internal ordering and runtime for future operations.

   True - 1,2,3,4,5,6 can be inserted in various different orderings that could result in different heap internals, that would affect future operation runtimes.

   (f) AVL trees and BSTs never have the same tight big-O runtime for inserting elements.

   False - BSTs can become balanced by chance and exhibit at worst log(n) behavior given certain orders of insertions.

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

   False - logs of different bases are actually all the same complexity class.

   (h) For a graph with negative edge weights, adding the same 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 - You can come up with an example where adding the constant to every edge actually changes the original shortest path to not be the shortest path in the new graph.

   (i) MSTs can be used to find the minimum cost path between any two nodes.

   False - see exercise 5.

   (j) Graphs can be implemented with a Dictionary.

   True - adjacency lists are one such implementation.

   (k) To find the shortest path from one vertex to another in an unweighted graph, you should use Dijkstra's algorithm as it is the most efficient solution.
   False - you should use BFS - it is more efficient and Dijkstra's cannot run on an unweighted graph.

[1]We ran out of space on this page to put in a title, but this document is **Week 9: Final Review**.

2. **ADTs and Implementations**

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

(a) List

   i. doubly-linked-list:
      A. no need to resize (encountering theta(n) time in some situations)
      B. "better" memory usage no extra wasted space with empty array spots
      C. efficient add and removal near the front
   ii. array-list
      A. get by index is efficient theta(1) in all cases

(b) Dictionary/Map

   i. chained-hash-dictionary (or in general probing or chaining hash tables):
      A. in practice case for get/put/containsKey is constant runtime
   ii. array-dictionary
      A. easier to implement
      B. memory compact
      C. iterator should be very fast constant factors
   iii. AVL-dictionary
      A. getting keys in sorted order takes order n time (vs nlogn)
      B. has the best worst case runtimes among all dictionaries here - theta(logn) time for get/put/containsKey worst case
   iv. BST-dictionary
      A. better constant factors than AVL for adding bc no self balancing checks/rotations
      B. easier to implement

(c) Union-Find

   i. array-disjoint-set:
      A. union extremely efficient (worst case logn, in practice constant)
   ii. dictionary of representative/set id $\rightarrow$ collection of members in set
      A. can implement functionality to grab all members in a specific set efficiently
      B. less memory usage than array-disjoint-set
   iii. dictionary of member $\rightarrow$ representative/set id
      A. findSet is extremely fast
      B. less memory usage than array-disjoint-set

3. **Design Decisions: Sorting**

For each of the following scenarios, choose the most appropriate sorting algorithm from the list, then briefly explain (1-2 sentences) why your choice is the best.

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 if necessary.

Quick sort has the best constant factors and is in-practice nlogn running time. You could also argue heap/merge sort are good here because "as fast as possible" is pretty ambiguous.

(b) A company has lists of numbers that each need to be sorted, and they want their algorithm to be reliably and consistently fast so they can anticipate when the job will be done.

Any sort that always runs in the same runtime complexity class in all 3 cases is fine here. Between the two nlogn-always sorts, your options are either merge sort or heap sort.

(c) When you were younger, you may have 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 gradient color in the box such that you only take one or two of the crayons out of the box at any given time?

Any in place sort will work here, and if runtime matters and there's a large amount of crayons, you probably want to choose an nlogn sort. You could also argue, however, that children cannot comprehend the mechanics of in place quicksort or in place heap sort, so that selection sort is a viable option, especially if there aren't that many crayons.

4.

```java
public static void printReachingFriendships(ChainedHashDictionary<String, IList<String>> graph) {
    IPriorityQueue<String> heap = new ArrayHeapPriorityQueue<>();
    for (KVPair<String, IList<String>> pair : graph) {
        heap.add(pair.getKey());
    }


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

public static void mysterySort(DoubleLinkedList<String> list, int low, int high) {
    if (low < high) {
       /* pi is partitioning index, list.get(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<String> list, int low, int high) {
    String pivot = list.get(low);
    int i = low - 1;
    int j = high + 1;
    while (true) {
        i++;
        while (list.get(i).compareTo(pivot) < 0) {
            i++;
        }

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

        if (i >= j) { // if i has become big enough and j has become small enough that they cross
            return j;
        }

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

The above method printReachingFriendships takes in a graph in the form of an adjacency list. The vertices represent people and edges between two people represent that those people are friends. This means when you call graph.get("Robbie") a list of Robbie's friends is returned. The graph stored inside the dictionary parameter represents an undirected and unweighted graph.

Answer the following questions about the above method to model the runtime of the above code. You can assume that System.out.println and compareTo run in constant time. Assume that all ChainedHashDictionary and ChainedHashSet operations will run in their in-practice runtimes (even for worst/best case analysis here). All your answers for this part should be defined in terms of n and m, where they represent the total number of nodes (vertices) and total number of edges in the graph parameter.

(a) For the loop on lines 3-5 give a simplified theta bound for:

  i. the best case runtime $\Theta(n)$

  ii. the worst case runtime $\Theta(nlogn)$

(b) For the loop from lines 12 to 23:

  i. how many times are lines 13-15 executed in the worst case? n

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

  iii. how many times are lines 18 and 19 executed in the worst case? n

  iv. What is the worst case runtime for the loop from 12 to 23? Give your answer as a simplified theta bound. For now, say the runtime of mysterySort is S runtime. $\Theta(n + ms)$

(c) for mysterySort:

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

  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? $\Theta(p^2)$

  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? uneven splits / partitions $\rightarrow$ worse recurrence

  iv. What is the best case (describe it, not the runtime) for mysterySort? even splits / partitions $\rightarrow$ better recurrence

  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

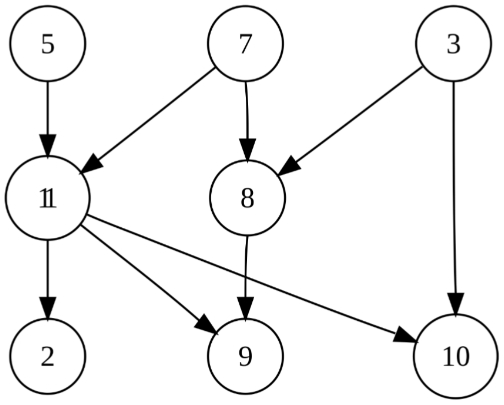
  vi. Putting that together, what's the worst case runtime for mysterySort? Assume that the size of the list parameter is called p. $\Theta(p^3)$

(d) Consider what inputs are being passed to mysterySort on line 21 in printReachingFriendships. Is this triggering situations more like the best case or the worst case runtime? worst case because continually sorting an almost sorted list

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

5. **Topo Sort**

Consider the graph below:



(a) What are the characteristics of this graph? (Hint: what is graph terminology is required for it to have a valid topological ordering?) directed, acyclic (is a DAG, so we can run topological sort on it)

(b) Give two possible orderings produced using Topological sort.
5, 7, 3, 11, 8, 2, 9, 10
3, 7, 5, 11, 8, 2, 9, 10

6. **Graph Modeling**

As a pilot for PrimAir, you have been assigned a charter flight that will take a rich customer from Shanghai, China to Jackson Hole, Wyoming. Unfortunately, the trip is about 7,000 miles direct, while your second-hand Cessna 152 only has a range of 500 miles. This means that you must pick airports along the way to refuel. However, it is a bad idea to land at every single every airport that you see along the way, because every landing adds about 50 miles of wasted flying (in addition to the ground distance that you are covering).

After taking CSE 373, you wrote your own flight computer and you decided to use your own software to model this flight. Your flight computer contains information about all the airports in the world. Assume each airport has enough jet fuel for your plane. **You want to perform the least amount of flying while not running out of fuel at any time during your trip.**

(a) Explain how you would model this scenario as a graph. Answer the following questions in bullet points or short sentences.

i. What do your vertices represent? If you store extra information in each vertex, what is it? Vertices represent airports. They store information like the name of the airport and the coordinates.

ii. What do your edges represent (your answer may be a real-world object, or an abstract description of what edges will exist)? If you store extra information in each edge, what is it? Edges represent the routes between airports that airplanes will potentially follow by flying through them. We will construct our graph to only have edges between two airports if the distance between them is less than 450 miles, so we know for sure that we can travel there safely. They don't really need to store any extra information besides the weights.

iii. Is your graph directed? Briefly explain (1 sentence). You could argue either: yes because wind makes a difference in how much fuel/distance you'll actually cover, or no because you could fly in either direction and it'd have the same properties.

iv. Is your graph weighted? If so what are the weights? Briefly explain (1 sentence). Yes the graph is weighted because we want to associate each edge with the distance between the two airports it connects. This will be useful later when we run existing graph algorithms on our graph.

v. Do you permit self-loops (i.e. edges from a vertex to itself)? Parallel edges (i.e. more than one copy of an edge between the same location)? Why? You could argue either: maybe yes for both because you have routes back to yourself planned and want to account for multiple air routes from one airport to another. Or maybe you could argue no for both, as you could state that the self looping routes aren't going to affect your goal of finding the shortest path and that you could just select the smallest weight the parallel edge to be included in your final graph going forward, so that you can simplify your setup.

(b) Consider the following information: Everett/Paine Field is 50 miles north of Seattle-Tacoma, Vancouver is 100 miles north of Everett/Paine Field, Anchorage is 1500 miles northwest of Vancouver. Using your graph model, draw out a portion of your graph using these four airports.

(c) Explain how you would run an algorithm taught in this course to find the best route from Shanghai to Jackson Hole. If applicable, say how you would modify the standard algorithm to take landing/takeoff overhead into consideration. Run Dijkstra's algorithm from Shanghai and the destination being Jackson Hole. Use the output from Dijkstra's algorithm to back trace through the predecessors and produce the path (list of airports and routes to actually visit that is the shortest distance and meets the criteria of the original problem).

(d) Using the answer above, write a simplified tight Big-O for the worst case runtime of your chosen algorithm in terms of n (number of nodes/vertices) and m (number of edges).

$$\Theta(mlogn + nlogn)$$