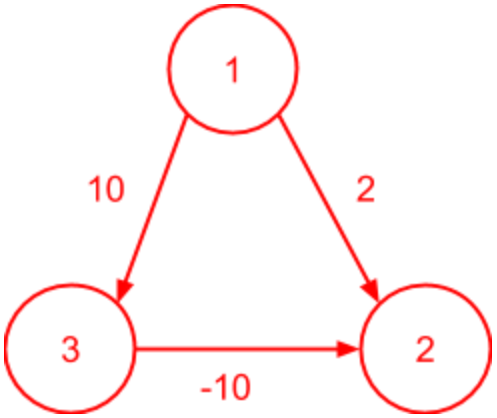
Practice Final- Solutions

This test has 13 questions worth a total of 200 points, and is to be completed in 180 minutes. The exam is closed book, except that you are allowed to use one double sided written cheat sheet (front and back). No calculators or other electronic devices are permitted. Give your answers and show your work in the space provided. Write the statement out below in the blank provided and sign. You may do this before the exam begins.

#	Points	#	Points
1	16	8	0
2	12	9	10
3	9	10	15
4	25	11	20
5	8	12	17
6	25	13	30
7	9		
		Total	200

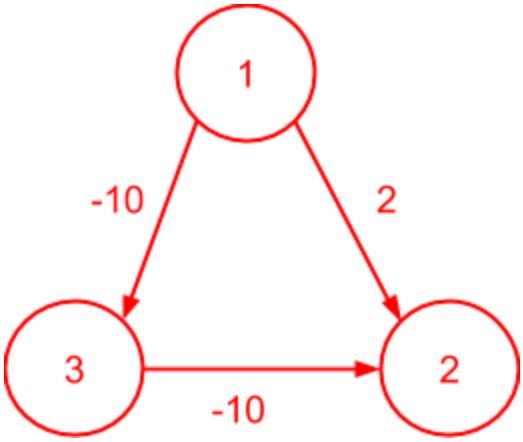
1. Starting this Test on a Positive Note (15 pts).

a) (4 pts) Give an example where Dijkstra's would fail because of a negative edge weight



In this example, we would go from node 1 to node 2. Then we go from node 1 to node 3. But we see that we have shorter path to 2 than we initially had- a contradiction.

b) (4 pts) Give an example where Dijkstra's **does not** fail with n negative edge weights. State assumptions that you must make in order for this to work. If you have some method to make it work, explain the method.



Take n = 2. If we start from the node 1, we will get the shortest path because of the fact that the negative edge weights can be reached from the start vertex by solely traversing negative edges. If there was any point where there is a positive edge weight between a negative edge weight and the start vertex, it is possible that Dijkstra's would mess up. This only works if the graph has no negative cycle formed, otherwise we would get an in infinite loop.

c) (4 pts) What is the least amount of negative edges (>0) that a graph can have and still have Dijkstra's work? What is the most amount of negative edges that we can have? State assumptions that must be made

1 edge is the least amount of negative edges for which Dijkstra's algorithm works. This will only work for directed graphs and if the shortest path to the vertex the edge goes to would be the shortest path if the negative edge was weighted 0.

One acceptable answer is that if the graph is a tree (|V| -1 edges and the graph is connected) and all the edges are negative, the shortest path will always be found. The maximum amount of negative edges goes back to one of our critical ideas, negative cycles cannot exist or you will get stuck in them. Another observation that we can make is that we cannot have more than 1 way to get a vertex, otherwise we cannot guarantee that there is not a shorter way to get there.

Another acceptable answer is $\frac{V(V-1)}{2}$, or a complete graph, with a generalized version from part b. Essentially what this means is that before the ith iteration, the Shortest Path Tree contains i-1 vertices. When the actual ith iteration is run, the node added to the Shortest Path Tree would have to be the neighbor of the vertex added in the ith iteration. For this to work, connect the first vertex with |V|-1 vertices, the second vertex with |V|-2 vertices and so forth up until the nth vertex is connected to no vertices. Then the total amount of edges would be $\frac{V(V-1)}{2}$. This works because there would be no way for a vertex to get back to itself (no cycles) which means we cannot go to negative infinity. Additionally, since we are only going through 1 path (that is the vertex added would be a direct neighbor, in the shortest path tree, of the vertex added in the iteration 1 before) there will never be a path to the vertex we added that is better.

The first option requires assumptions about how the graph is constructed wheras the second option needs assumptions about the actual Shortest Path.

d) (4 pts) What is the smallest weight we can have and guarantee that Dijkstra's will always work, regardless of the graph's construction?
0. Dijkstra's is guaranteed to work on edges with no negative edges.

2. Sorting and Fun (12 pts). In this class we have worked with various sorts. Below are a few sorts that we did not go over. In the box left of each sort, describe its runtime in Ω/O or Θ and a brief description of howhow it sorts. Each questions is worth 4 points

Ω(N) O(N^2):
This algorithm swaps 2 elements if they are in the wrong order.
In the best case, the list is almost entirely sorted. In this case, you go through the list once, don't find any (or very few) out of order pairs.
In the worst case, the list is completely inverted (eg {3,2,1}) the algorithm would then have a runtime of **Θ(N^2).** This is because the counter decrements if an element swaps, so each of the N items will be swapped N times, which makes the runtime **O(N^2)** in the worst case.

```
public static void dwarfSort(int[] ar) {
  int i = 1;
  int N = ar.length;
  while (i < N) {
    if (i == 0 || ar[i - 1] <= ar[i]) {
      i++;
    } else {
      int tmp = ar[i];
      ar[i] = ar[i - 1];
      ar[i-1] = tmp;
      i--;
    }
  }
}
```

Θ(N): This is actually one of the most pointless sort algorithms because all it does is check if the ith element is less than the i-1st element, if so just brute force it to be bigger (adding one to the i-1th element). You go through the array once and you are done "sorting" after 1 iteration.

```
private static int[] waffle(int[] arr){
   int[] syrup = arr;
   for(int i = 1; i < arr.length; i++){
      if(arr[i] < arr[i-1]){
        arr[i] = arr[i-1] + 1;
     }
   }
   return syrup;
}
```

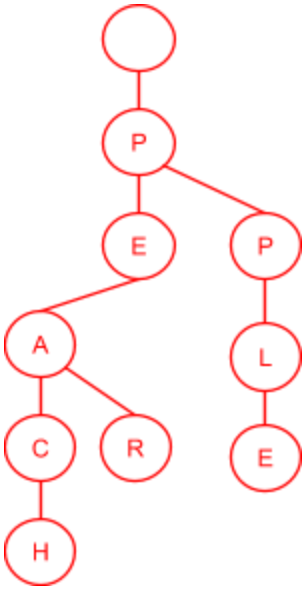
Θ(N^2): In this sort, we are keeping track of some pointer(curr_size) which is initially set to the size of the array. We find the maximum element and then we reverse the array from 0 to the index of the maximum element (so the maximum
element becomes the 0th index). Then we reverse the array from

```
//Find the runtime of flapJackSort. Note that the flip
function is defined as follows flip(array, index) and reverses
the entire array up until that index.

public static int[] flapJackSort(int arr[]) {
  for (int curr_size = arr.length; curr_size > 1; curr_size--) {
    int syrup = 0;
    for (int choco = 0, chip = 0; chip < curr_size; chip++) {
      if (arr[chip] > arr[choco]) {
        choco = chip;
        syrup = choco;
```

the 0th index to curr_size. We then decrement curr_size so we look at everything before that element. This sort does a total of N “flips”/reversals, so we have a running time of **Θ(** N^2 **).** This sort was actually made by Bill Gates, its real name is Pancake Sort.	``` } } if (syrup != curr_size - 1) { flip(arr, syrup); flip(arr, curr_size - 1); } } return arr; } ```

3. The Fruits of Your Labor (15 pts).

a) (3 pts) You own a trie that can hold many fruits; however, you want to save space so that your neighbor doesn't cut off your branches. Find how you make one modification (change, delete, or add a letter) to one of the following 3 words "Apple", "Peach, "Pear". What modification can you do to save the most space? Draw the resulting trie after your modification.



The modification that was done was removing the "a" from apple. This makes it so that all the items share a prefix of "p". We deleted one node and made one shared among all 3 words. No other change would have made us use less space since we removed 2 nodes with 1 deletion.

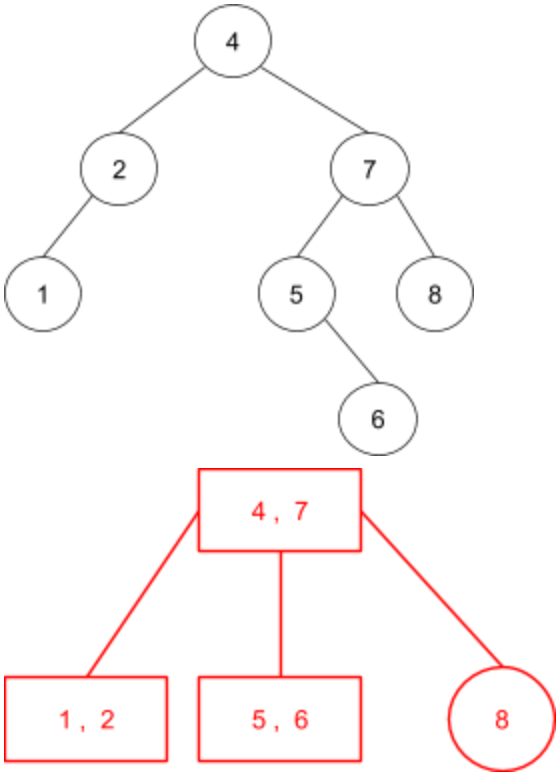
b) (3 pts) A wealthy CEO wants to encrypt all his secrets; however, he decided that he wants to store all of his secrets within a flashy hashmap. But he doesn't want a plain old hashmap, after all, that could get hacked. To be different, he decides that, for any given day, a secret will hash to different location than it would have the previous day. Is his idea valid? Provide your reasoning.

This IS valid. There is a subtilty in this problem, where we say items hash to a different location. This does not mean that it has a different hashcode. If a secret hashes to a different location then it would have a day ago, that could just mean that the hashmap has been resized. There is no problem with this unless we know that the hashmap retains its size or we know that the hashcode is different for two equivalent secrets.

c) (3 pts) 2-3 Trees are known to be fast because they are self balancing and they have a nice low level? Why don't we just use 2-3-4 trees or 2-3-4-5-6 trees?

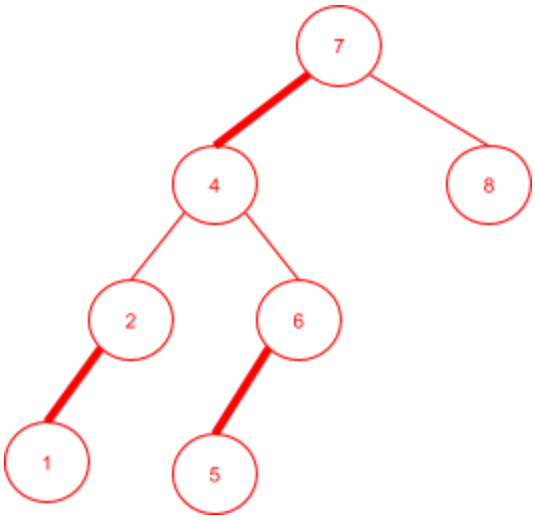
The main reason we don't have trees like this is because they are fairly difficult to implement. Additionally, there really isn't anything to gain, except for possibly a change in a constant factor. However, if we have larger nodes, we will have to use more memory for each type of node, which leads to more memory overhead to worry about.

d) (3 pts) Create the 2-3 tree that corresponds to the below Binary Search Tree.



Something important to note in this problem was the order that the items were inserted in the initial Binary Search Tree. You couldn't just get all the keys and make an arbitrary 2-3 tree. After deriving a set of insertions that work for the initial Binary Search Tree, simply perform the insertions on a 2-3 tree model.

e) (3 pts) Create a Red-Black Tree that corresponds to the 2-3 tree/Binary Search Tree in part a.



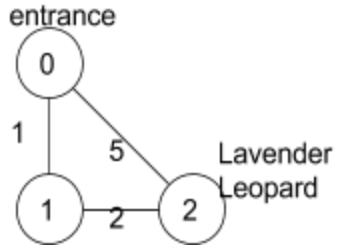
Simply convert the above 2-3 tree into a Red-Black Tree using our conversion rules. Another way to do this is look at the order of insertions for the Binary Search Tree and directly insert into the Red-Black Tree.

*The bolded links are red links

4. The Lavender Leopard (15 points).

You are a jewel thief starting at the entrance of a museum where all rooms are connected to each other- that is a room has a hall to every other room besides itself; however, hall lengths may differ. You are in search of the very expensive gem, the "Lavender Leopard", the only problem is that you have no idea what room it is in.
Your idea is to go to the room closest to the one you are currently in (excluding ones that you have already visited) and then as soon as you find the gem, you will immediately exit the building. When going towards the exit, you want to pick a path that will have you travel, at most, the distance that you travelled to get to the gem. However, if you know for a fact that a hall will take you to the exit faster you will take it.

a) (20 pts) Fill in the Museum class and following code to find the gem and then escape as quickly as possible. You are guaranteed that the jewel will be in the building.



In the above example, you would travel to 1, then 2, get the gem, and go back to room 1 then the entrance.

```
public class Hall {
  public final Room room1; public final Room room2;
  public final double length;
  public Hall(Room r1, Room r2, double length){
     this.room1 = r1; this.room2 = r2; this.length = length;
  }
  public Room r1(){return room1;}
  public Room r2(){return room2;}
  public double howFar(){return this.length;}
```

```
public class Room {
  public LinkedList<Hall> adjacentHall;
  boolean beenTo = false;
  public boolean hasGem;
  public int order; //provides a definite ordering for the rooms, does not tell you the location, but can be used as some reference.
  public Room(int order, boolean gem, int rooms) {
     this.order = order;
     this.hasGem = gem;
     adjacentHall = new LinkedList<Hall>();
    }}
```

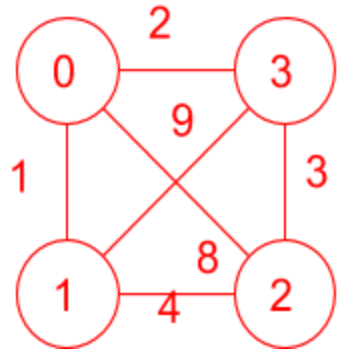
```
import java.util.ArrayList; //feel free to use these imports, if not it's okay as well
import java.util.LinkedList;

public class Museum {
public int totalRooms;
public int totalHalls;
public LinkedList<LinkedList<Hall>> adj;
for (int room = 0; room < rooms.length; room++) {
  adj.add(room, rooms[room].adjacentHall);
}
public Room entrance;

//Constructor
public Museum(Room[] rooms) {
  this.totalRooms = rooms.length;
  this.totalHalls = 0;
  adj = new LinkedList();
  for (int room = 0; room < rooms.length; room++) {
    adj.add(room, rooms[room].adjacentHall);
  }
  entrance = rooms[0];
}

 public void steal() {
    ArrayList<Room> visited = new ArrayList<>();
    Room r = goIn(visited);
    goOut(r, visited);
  }

  public Room goIn(ArrayList<Room> visited) {
    Room r = entrance;
    boolean stolegem = false;
    while (!stolegem) {
      visited.add(r);
      r.beenTo = true;
      if (r.hasGem) {
        stolegem = true;
        break;
      }
      Hall minHall = new Hall(new Room(0, false, 0), new Room(0, false, 0),
Double.MAX_VALUE);
      for (int i = 0; i < adj.get(r.order).size(); i++) {
        Hall curr = adj.get(r.order).get(i);
        if (curr.length < minHall.length && !((curr.from().beenTo) && curr.to().beenTo)) {
          minHall = curr;
```

```
            }
        }
        if (r.order == minHall.from().order) {
            r = minHall.to();
        } else {
            r = minHall.from();
        }
    }
    return r;
}

public void goOut(Room r, ArrayList<Room> visited) {
    int bound = visited.size();
    while (r != entrance) {
        Room closestroom = null;
        double dist = Double.MAX_VALUE;
        int currentlimit = bound;
        for (Hall neighborhall : adj.get(r.order)) {
            if (neighborhall.howFar() < dist) {
                Room neighborroom;
                if (neighborhall.to() == r) {
                    neighborroom = neighborhall.from();
                } else {
                    neighborroom = neighborhall.to();
                }
                for (int index = 0; index < currentlimit; index++) {
                    if (neighborroom == visited.get(index)) {
                        bound = index;
                        closestroom = neighborroom;
                        dist = neighborhall.howFar();
                    }
                }
            }
        }
        r = closestroom;
    }
}

}
```

The key to this problem was seeing that there was an “order” in the Rooms, which allowed us to give them indices or “rightful places”. Another thing to realize is that the rooms can be treated as vertices and the halls can be treated as edges (reducing this problem just another graph traversal

one). We need to have a few inelegant checks to make sure that we do not keep going to the same room over and over again. We have to do this check because we have undirected edges, which means, at any time, we do not have the idea of "to" or "from.

Leaving the museum is more confusing than the entering. You are only allowed to visit rooms that you've visited already. This means that you cannot just go to the closest room, you must be limited to the list of rooms that we went to. To do this, we use the visited arraylist that we added to in the goIn method. We only want to go down paths that we know for sure will allow us to get better or stay the same. Say we are currently at some room r, and it has a parent r' meaning that in the path to the gem, we went from r' to r. Now assume that there is some ancestor of r', r'', which was visited earlier on the way from the entrance to r and that the distance of our path from the entrance to the room with the gem is P. At any room, r, if you skipped r', and instead went to r'', you know it would be cheaper than P because you would be skipping the hall from r' to r because you found a cheaper hall that goes up further in the path. We know the vertex that is chosen is on the path, since, in our initial path from the entrance to gem, each vertex has at most 1 parent and 1 child. We know that, on the way back, we would have had to go to that vertex eventually, if we went in the directly opposite route. This means our path is cheaper than P by at least $r' - r$ making our cost upper bounded by P- $r' - r$.



Here we have the case where we would find the gem by going from room 0 to room 1 to room 2 and finally reaching room 3 (where the gem is). However, now the alarms have been tripped, and it would take us way too long to get out, instead we could go directly to the entrance.

b) (5 pts) What is the worst case runtime of stealing in terms of rooms (r) and halls (h)? Only keep necessary terms.

$\Theta(2r + h) = \Theta(2r + \frac{r(r-1)}{2}) = \Theta(r^2) = \Theta(h)$. We go to each room twice, and at each step, we examine r edges to see which room we should go to next. In this problem, h = $\frac{r(r-1)}{2}$ because there are halls between each room (making it equivalent to a complete graph).

5. Trick or Tree-t (12 pts). Mark true or false for the following problems, a brief justification is required. Each question is worth 3 points

a) The fastest time for any comparison based can ever be is **Ω**(Nlog(N)) regardless of input.

☐ True ■ False

Heapsort, if all the elements are the same, will have a runtime of Θ(N). Additionally insertion sort could run in Θ(N) if no inversions are needed.

b) When looking through a binary search tree, if we are looking for a number of items between 2 elements, the best case runtime is Θ(N) because we will need to go through every single element to see if it fits into our range?

☐ True ■ False

This is false, as this is the definition of pruning, or range finding. The runtime for this process is Θ(logN + R) where R is the total amount of elements that you need to find in your range.

c) All valid left leaning Red Black Trees can become a valid Binary Search Tree by replacing all red edges with black ones.

■ True ☐ False

By definition, Red Black Trees are self balancing Binary Trees. Essentially, by making all nodes black, you would just be removing the quality of self balancing.

d) 2-3 Trees are faster than Red-Black Trees for most major operations because of the height of the 2-3 tree is smaller than the height of a Red-Black Tree.

☐ True ■ False

This is false, simply because, in a 2-3 tree, instead of traversing down a level like we would in a Red-Black Tree, we would have to make another comparison. This makes the overall runtime the same.

6. The Lavender Leopard Strikes Again (15 pts). You were examining your gem after successfully stealing it only to realize that you had stolen a fake gem, and that the real Lavender Leopard is in a much more secure museum. In this museum, not every room is connected. To avoid ridiculous amounts of backtracking, you have decided to place teleporters in every room you visit. At any point, if there is any path that is closer to the entrance than the one you are taking, you will teleport to the room before and then travel to it. Once you find the gem, you need to get out as soon as possible. To get out, you will teleport in the reverse order that you arrived so you can collect all your teleporters (they aren't that cheap!). Note that Hall and Room have the same definition that they did for #4. Additionally, you **do not** need to redefine class level variables and the constructor (you can if you want to), as they are the same as the Museum class's . You have 3 pages to complete this.

If you cannot write the code, but can write the idea behind the problem, you can get up to % of the points.

Hint 1: Use old algorithms that we've gone over and see how you can modify them to match this problem

Hint 2: Look at other data structures.

```
class Node {
  public Room r;
  public double myPriority;

  public Node(Room r, double priority) {
     this.r = r;
     myPriority = priority;
  }
}
public class TeleportMuseum {
  public int totalRooms;
  public int totalHalls;
  public LinkedList<LinkedList<Hall>> adj;
  public Room entrance;

  public TeleportMuseum(Room[] rooms) {
     this.totalRooms = rooms.length;
     this.totalHalls = 0;
     adj = new LinkedList();
     for (int room = 0; room < rooms.length; room++) {
        adj.add(room, rooms[room].adjacentHall);
     }
     entrance = rooms[0];
  }
 public void teleportInNOut() {
     Stack<Room> s = new Stack<>();
     teleportSteal(s);
     teleportLeave(s);
```

```
  }

public void teleportSteal(Stack s) {
  Hall[] edgeTo = new Hall[totalRooms];
  double[] distTo = new double[totalRooms];
  PriorityQueue pq = new PriorityQueue();
  for (int v = 0; v < totalRooms; v++) {
    distTo[v] = Double.MAX_VALUE;
  }
  distTo[entrance.order] = 0.0;
  pq.insert(entrance, distTo[entrance.order]);
  while (pq.size() != 0) {
    Room temp = pq.removeMin();
    temp.beenTo = true;
    s.push(temp);
    if (temp.hasGem) {
      break;
    }
    for (Hall h : temp.adjacentHall) {
      if (h != null && !(h.from().beenTo && h.to().beenTo())) {
        done(h, temp, distTo, edgeTo, pq);
      }
    }
  }
}
public void done(Hall h, Room temp, double[] distTo, Hall[] edgeTo, PriorityQueue pq) {
    Room r;
    if (h.from().equals(temp)) {
      r = h.to();
    } else {
      r = h.from();
    }
    if (distTo[r.order] > distTo[temp.order] + h.length) {
      distTo[r.order] = distTo[temp.order] + h.length;
      edgeTo[r.order] = h;
      if (pq.contains(r)) {
        pq.changePriority(r, distTo[r.order]);
      } else {
        pq.insert(r, distTo[r.order]);
      }
    }
  }
}
public void teleportLeave(Stack<Room> s) {
  while (!s.isEmpty()) {
```

```
        Room r = s.pop();
    }
    System.out.println("Out again");
}
```

The initial part of the TeleportMuseum class is very similar to problem 4.

The Node class was created as a wrapper for the room and priority so we could use it within a priority queue.

Inside of the teleportInNOut method, we have a stack, which helps us keep track of the order we go to rooms, and we have 2 calls, one to teleportSteal and one following teleportLeave.

The teleportSteal method is extremely similar to Dijkstra's Algorithm. It pretty much runs the exact same, except that each time we remove an element from our priority queue, we add it to our stack.

In the teleportLeave method, all we do is pop our stack until it's empty. The stack naturally serves as the data structure for which we will have "reverse" order because it is FILO (First In Last Out).

This question was pretty coding intensive, more so than #4, but it required less thinking if you knew what the algorithms were. The key was understanding how to use data structures in conjunction with a graph algorithm.

*There was no need to write the class level variables and the constructor as it is identical to #4. It was only put on the answer key for completeness sake.

7. D-Arranged Lab (9 pts). You are viewing a research group, unfortunately, this research group is full of some bumbling buffoons who happen to make a lot of mistakes. Each part is worth 3 points.

a) While putting his chemicals into test tubes, one of the scientists realized that he accidentally made a duplicate of one of his n chemicals, he just doesn't know which one. However, he does know that the duplicate element will be, at most, 10 test tubes away. What sort should he use to quickly find the duplicate? Assume that all the chemicals are in their natural order, except for the duplicate.
This is very similar to finding a sort that has the fewest amount of inversions. Thus, we should use insertion sort. After running insertion sort, find the element that has an equivalent element next to it and remove one of them.

b) You have separated each of your n test tubes into k sections (so test tube 1 is now in separated into k test tubes and so on). The only problem is that while they were all getting analyzed, you mixed up chemicals made during the same time in chronological order. For example, you have a set of mixed up chemicals made at at 2 am then you have another set of mixed up chemicals made at 5:32 pm. Each test tube has the the name of the chemical that was created. You want to sort your test tubes by chemical and then by it's creation time. What sort can you use to ensure this is done as quickly as possible?
We will use merge sort. The main reason why is that it takes Θ(nlogn) time to sort as well as it is stable. Because it is stable, the relative ordering, which in this case is the order in which the k test tubes were oxidized.

c) Now the scientists are going to input the name of their chemicals into a spreadsheet to see if they are dealing with your list of hazardous materials. The list is in alphabetical order so to easily cross reference what chemicals are dangerous, the scientists want to put the name of their chemicals in alphabetical order, what sort should they use?
This is a direct application of Radix Sort. The key give away is the "alphabetical order"

8) Riddle me this:
What room can no one enter?

A mushroom

9. Algorithms (10 pts)

a) You are given a positive edge weighted graph and you wish to find a Minimum Spanning Tree based off of the products. Basically you want to find a tree that minimizes the total product of your edge weights. Describe how you would do this.
Take the logarithm of all your edge weights then run either Prims or Kruskals to find the minimum spanning tree. This works because $log(a * b) = log(a) + log(b)$. The edges that minimize the sum of the logarithm would then minimize the product of the edges in the graph.

b) You are living in the olden days, where cereal was given no milk and frozen food was just any animal in Antarctica. As a mind ahead of your time, you realized that all the villages around you very isolated from each other. You have taken it upon yourself to get them in communication with each other by building chariot roads between them. In addition to these roads, you can build ship ports- you can row your dinghy or canoe between any cities with ports and resume your travels via chariot. You are provided the cost it would take to create a chariot road between every village i and every other village j as well as the amount it would cost to create a ship port at any village- the cost to create a path between the 2 ship ports is nothing because one simply goes with the flow on the river. Your task is to find a way to create a system where it is possible for someone to get from one village to the other, and you should pay as little as possible for construction.
This is essentially a Minimum Spanning Tree problem. To approach this problem, connect all the villages to some dummy node. The weight of the edge between some village i and the dummy node will be the cost it takes to create the edge. Run Kruskal's algorithm as normal except allow the dummy node to have more than 1 edge coming out of it. If the algorithm terminates and the dummy node only has 1 edge coming out of it, then the cheapest system is one without any ship ports, otherwise ship ports are useful.

c) How many unique Minimum Spanning Trees are possible in any given graph? Define any variables that you need to. Examples of variables may be vertices, cycles that the ith node is a part of, connected components at the jth iteration etc. Provide the tightest bound.

10. JJJJ UNIT (15 pts).
You are starting your rapping career as a part of the prestigious rapping group J Unit. You want to recruit new members so you can have a hip possi. To prove their worth to your group, you have decided that you will test their vocabulary. To do this, you will make a TrieMap. This will make a Trie, but for each word, you also have a note to see how many times it has been used. If any word has been used more than 5 times, the person will not be able to join your group.
You are given the TrieNode class defined as follows:

```
public class TrieNode {
   char c;
   boolean isWord;
   String word;
   TrieNode[] children = new TrieNode[26];
   int count;}
```

Additionally, you are given that the TrieMap has a root class level variable and an **addWord** and **contains** method.
*The LinkedList has an addAll method which adds all items from a collection to the end of the list (e.g. LinkedList A = {A,B,C}, LinkedList B = {D, E,F}. A.addAll(B), A = {A,B,C,D,E,F}

Fill in the below methods

```
public LinkedList getAllWords(TrieNode start) {
   LinkedList list = new LinkedList();
   if (start.isWord) {
      list.add(start);
   }
   TrieNode[] node = start.children;
   TrieNode temp;
   for (int i = 0; i < node.length; i++) {
      temp = node[i];
      if (temp != null) {
         list.addAll(getAllWord(temp));
      }
   }
   return list;
}
```

```
public static boolean testCount(TrieMap t){
   LinkedList<TrieMap.TrieNode> words = t.getAllWords(t.root);
   for(TrieMap.TrieNode node : words){
       if ( node.count < 5){
       return false;
       }
   }
   return true;
}
```

11. Median Heap (20 pts).

We have earlier dealt with Min Heaps and Max Heaps, now we are going to explore a new type of heap, the **Median Heap**. The Median Heap will have 3 methods that you will need to fill in, **median**, which finds the median element, **insert**, which inserts an element into the Median Heap, and **balanceHeap**, which makes the heap balanced. You may need to use some of these methods inside of others. Feel free to use any data structure that we have gone over.

```
public class MedianHeap {
  private MinPQ top = new MinPQ();
  private MaxPQ bottom = new MaxPQ();

  //find the median element, if there are an even amount of elements, take the    average of the 2 elements closest to the middle
  public int median() {
     int minSize = top.size();
     int maxSize = bottom.size();
     if (minSize == 0 && maxSize == 0) {
        return 0;
     }
     if (minSize > maxSize) {
        return top.min();
     }
     if (minSize < maxSize) {
        return bottom.max();
     }
     return (top.min() + bottom.max()) / 2;
  }

  //used to insert elements into the MedianHeap
  public void insert(int element) {
     int median = median();
     if (element > median) {
        top.insert(element);
     } else {
        bottom.insert(element);
     }
     balanceHeap();
  }
```

```
  //Balance the heap
  private void balanceHeap() {
     int minSize = top.size();
     int maxSize = bottom.size();
     int tmp = 0;
     if (minSize > maxSize + 1) {
        tmp = top.delMin();
        bottom.insert(tmp);
     }
     if (maxSize > minSize + 1) {
        tmp = bottom.delMax();
        top.insert(tmp);
     }
  }
}
```

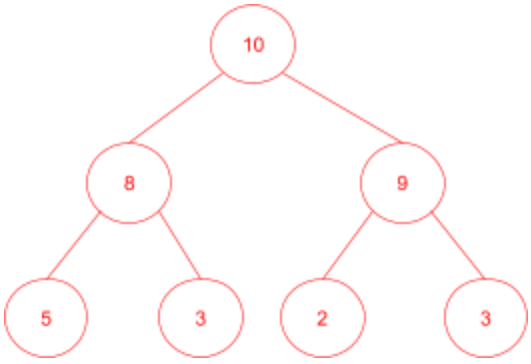
The key to this problem was to realize that you could make a MinHeap with the top 50% of the data inside of it and a MaxHeap with the bottom 50% of the data. To find the median, we need to first check if either of the heaps is bigger than the other (the maximum difference is 1, we will go over why later) if so, you pick the root element from the bigger heap. If they are the same size, you find the average of the roots of both the heaps.

Inserting into the Median Heap is relatively straightforward. You simply check to see if the element is greater than the median, if so you put it in the "top" MinHeap, if not you put it in the "bottom" MaxHeap. After is a call the balanceHeap().

The balanceHeap() method requires you to think about when the median element would not be either the root of one of the heaps or the average if 1 of them. This would only occur if the MinHeap was bigger than the MaxHeap by more than 1 element or vice versa. We make a check for both of these cases and delete the root of the bigger one each time and move it to the other heap.

12. Faster than a Speeding Runtime (17 pts)
Fill in the runtimes for each of the following operations. If there is no tight bound, provide a lower and upper bound. Provide a brief explanation as to why the runtime is what it is.

a) (5 pts) How much time does it take to create a heapify an array of N elements?
O(N). At first glance, it may seem that the answer should be O(NlogN) because of how we learned heapsort. The reason why the runtime is substantially smaller is because the maximum amount of times an element can sink is proportional to its initial spot in the heap. Keep in mind, that when heapifying, we start from the bottom and move up and that every root must



The bottom is full of singular nodes, all of which are their own heap. Since they are all valid heaps, they can sink a total of 0 levels. The 2nd layer to the bottom can sink a total of 1 layer and the top layer can sink a total of 1 layer. There are a total of 7 nodes $(2^3 - 1)$. The total possible amount of sinking operations is equal to $2(1) + 1(2) + 0(4) = 4$.

Now to generalize this, there are a total of N nodes. In a complete heap, approximately $\frac{1}{2}$ of the N nodes will be on the bottom layer, $\frac{1}{4}$ on the layer directly above the bottom one, $\frac{1}{8}$ above that layer and so forth. As we discussed earlier, the nodes on the bottom layer can sink a total of 0 layers, the second bottom layer can sink a total of 1 layer, and so forth. We can rewrite this as $0(\frac{N}{2}) + 1(\frac{N}{4}) + 2(\frac{N}{8})... = \sum_{i=0}^{log(N)} i(\frac{N}{2^{i+1}}) \leq N$.

b) (3 pts) How much time does it take to find the minimum element in a Binary Search Tree
Ω(1) if it is a spindly Binary Search Tree and it is leaning right
O(N) if it is a spindly Binary Search Tree and it is leaning left

c) (3 pts) How much time does it take to place all the contents of a given MinHeap into an array that is ordered from least to greatest.
Θ(NlogN). You will have to do a total of N deletions, each which will take logN time.

d) (3 pts) How much time does it take to find the minimum value in a MaxHeap.
O(N). You will have to go through the bottom layer, which, as we discovered earlier, contains about $\frac{N}{2}$ nodes. Since we disregard constants, our running time is O(N).
e) (3 pts) Find the runtime of getting all the words from a trie and then using MSD sort on them.

O(W*l), The runtime to find all the words in a trie is O(w*l), where w is the amount of words and l is the length of the longest word. To perform MSD sort on all of them, it takes (w*l). We add the two runtimes and get O(2(w*l)) = O(w*l)

13) Future-Roaming (30 points).
You are a croissant delivery boy in the 31st century. You want to deliver your croissants as quickly as possible; however, because you lost your client's addresses and they live all across the galaxy, so you don't want to go door to door. You have the phone numbers of all the people who ordered croissants from you, so you will attempt to use this to figure out in where you should go. Each phone number is about 10000000000000000000000 digits long and you have a total of N clients. You will use the following information about telephone numbers to help:

1. The sector code (the leftmost 10 digits) will allow you to figure out which sector a client is in. On average, there are a total of $\sqrt{N}$ of households you need to deliver to in the sectors.
2. In the 31st century, telephone numbers have a cool feature where the difference between the last 4 digits of any 2 telephone numbers in the same sector is equal to the distance between their two corresponding households.
3. You know the address of 1 of the houses you need to go in each sector.

You want to go to sectors based off how many of your clients are in them and how close it is . If a sector has 5 clients and is 2 Megadistances away, it is more appealing than a sector that has 1 client and is 1 Megadistance away. If a sector has 6 clients and is 2 Megadistances away, it is less appealing than a sector with 9 clients and 3 Megadistances. If a sector has 4 clients and it is 2 Megadistances away, it is considered equal to a sector that has 2 clients that is 1 MegaDistance away, at this point, you just pick which one to go to randomly . You will deliver to all the households in that sector before going to the next sector. You also know how many megadistances are between each sector, and you can assume that you will start at a location that is equidistant from all the sectors.

Provide an implementation where figuring out which households are in which sector in **O(N)** time, figuring out the distances between all the houses in a sector, must take, on average, **O(N)** time, and delivering croissants to all the households should take **O(** $NlogN$ **)**time.

Assume that the time it takes you to walk 1 Minidistance (inside of a sector) is the same amount of time it takes your spaceship to travel one Megadistance (between sectors). Justify why your design works in the given time.

What you first want to do is run a special version of MSD on all the telephone numbers. This version of MSD would stop sorting after we get 10 iterations, since we would have sorted by the 10 most significant digits of each telephone number, and have all the numbers sorted by sector. We are using a radix sort and we are not doing it all the way through.

Additionally, since we are performing only a partial radix sort, our work done will be O(3*N) which is just O(N), fitting our constraint for figuring out which houses are in which sector.

We will denote the amount of households in a sector by the letter T.

For each sector, we will create a graph with a total of T vertices. Following this, for each household in the sector, we will find the absolute value of the difference between it and every other household in the sector. After finding the absolute value of the difference between the 2 vertices, we would connect these two households with an edge of that weight.
In (somewhat detailed) pseudocode, the above looks like:

```
for(int i = 0; i < number of households; i++){
    for(int k= i + 1; k < number of households; k++){
        int distance = absolute value (last 4 digits of i's telephone # - last 4 digits of k's #);
        addEdge of weight distance between i and k;
     }
}
```

On average, there are $\sqrt{N}$ households per sector. This means that for each household in the sector, on average, we perform $\sqrt{N}$ comparisons to create edges between it and another household. Our recurrence relation would look as follows:

$$\sum_{i=1}^{\sqrt{N}} \sqrt{N} - i = \sqrt{N} + \sqrt{N} - 1 + \sqrt{N} - 2 ... \sqrt{N} - \sqrt{N} + 1 = O(\sqrt{N}^2) = O(N)$$

We will denote the amount of sectors with the letter M

Following this, we will create a graph with all the M sectors; however, the "weight" of each edge would be equal to *the Megadistance to the sector / amount of deliveries to make in that sector*, essentially reducing this problem to "which path maximizes the amount of Megadistance/amount of deliveries". This means that the more deliveries we have in a sector, the smaller the value, which means in Dijkstra's it would be prioritized more.

We would perform Dijkstra's from the sector on the graph of sectors and at we would perform Dijkstra's at each sector that we stop in. In the average case, there are a total of $\sqrt{N}$ sectors and $\sqrt{N}$ households per sector. The Dijkstra's between the sectors will take $O(\sqrt{N}\ log(\sqrt{N}))$ and each sector, running Dijkstra's will take $O(\sqrt{N}\ log(\sqrt{N}))$ time. Since, we know that there are a total of $\sqrt{N}$ sectors, we can multiply our sector Dijkstra runtime by $\sqrt{N}$ getting $O(\sqrt{N} * \sqrt{N}\ log(\sqrt{N})) = O(Nlog(\sqrt{N}))$. We can add the terms to get
O(
$Nlog(\sqrt{N}) + \sqrt{N}\ log(\sqrt{N})) = O(Nlog(\sqrt{N})) = O(Nlog(N^{1/2})) = O(\frac{1}{2}Nlog(N)) = O(Nlog(N))$

Easy Mistakes to make:

1. Attempting to perform full MSD or any form of LSD would be incorrect. Our digits are simply too large to perform either one efficiently. The reason why a truncated version of MSD works is because we fit the problem so that only the first digits were considered (essentially making it a constant factor).
2. Constructing a Trie would not work for seeing which numbers contain the same 10 digit prefix because we would have to go all the way down the Trie to see the last few digits that we need.
3. Not figuring out that the weight of an edge should be changed from just the distance or adding up/subtracting the amount of people in a sector and the distance. When adding the 2 values, it is impossible to know if you're dealing with a greater distance or amount of deliveries. When subtracting you could end up with negative edges/cycles. The ratio that we created allows us to properly scale the impact that the amount of people in each sector has on the algorithm.