UC Berkeley – Computer Science
CS61BL: Data Structures
Final, Summer 2017

This test has 11 questions worth a total of 60 points, and is to be completed in 170 minutes. The exam is closed book, except that you are allowed to use three double-sided page of notes as a 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. **Any plagiarism, no matter how minor, will result in points deducted from your exam.**

"I have neither given nor received any assistance during the taking of this exam."

_____

_____

Signature: _____

Write your name and student ID on the front page. Write the names of your neighbors. Write and sign the above statement. Once the exam has started, write your class ID in the corner of every page.

Name: _____ **Your Class ID: _____**

SID: _____ **Name of person to left: _____**

TA: _____ **Name of person to right: _____**

Tips:

- There may be partial credit for incomplete answers. Write as much of the solution as you can, but bear in mind that we may deduct points if your answers are much more complicated than necessary.
- There are a lot of problems on this exam. **Work through the ones with which you are comfortable first. Do not get overly captivated by interesting design issues or complex corner cases you're not sure about.**
- Not all information provided in a problem may be useful.
- Unless otherwise stated, all given code on this exam should compile. All code has been compiled and executed before printing, but in the unlikely event that we do happen to catch any bugs during the exam, we'll announce a fix. Unless we specifically give you the option, the correct answer is not 'does not compile.'
- ○ indicates that only one circle should be filled in.
- ☐ indicates that more than one box may be filled in.
- For answers that involve filling in a ○ or ☐, **please fill in the shape completely.**

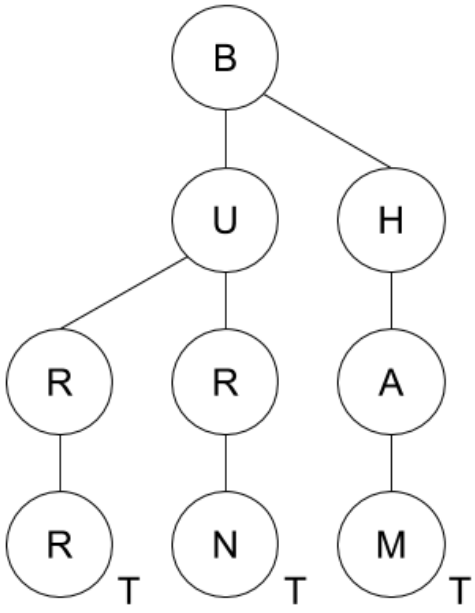
Optional. Mark along the line to show your feelings on the spectrum between ☹ and ☺.

Before exam: [☹_____☺].

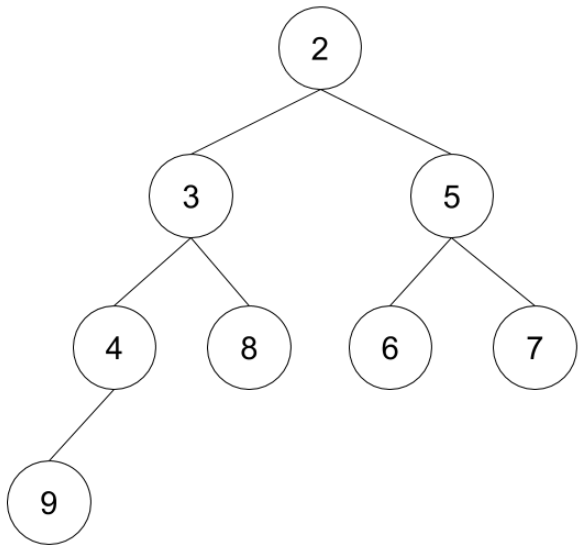
After exam: [☹_____☺].

1. Steven the Pusheen (4 pts)

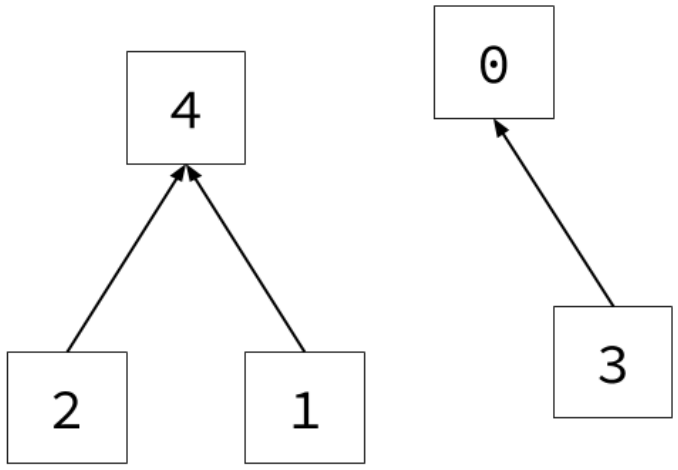
a. List the keys that are in the **ternary search trie** below in the middle box. Then, draw the result of inserting "HOT" into the ternary search trie. Mark the node that terminates the new key with a "T".

Original TST	List of Keys	`insert("HOT")`
		

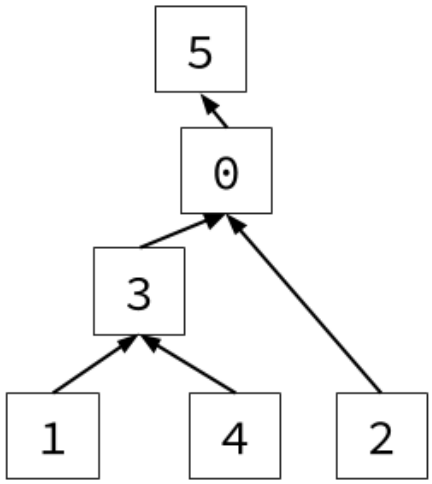
b. Consider the **binary min-heap** below. Draw the corresponding heap after `removeMin()` has been called once.

Original Binary Min-Heap	`removeMin()`
	

c. Consider the **weighted quick union** object below. In the space provided, draw the result of calling `union(3, 4)`. Assume the object implements **path compression**.

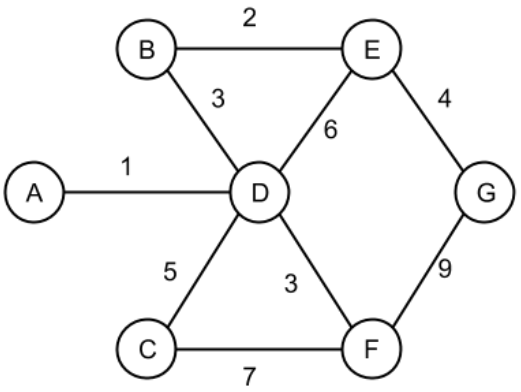
Original WQU w/ path compression	**`union(3, 4)`**
	

d. Consider the **weighted quick union** object below. In the space provided, draw the object after calling `find(4)`. Assume the object implements **path compression**.

Original WQU w/ path compression	**`find(4)`**
	

2. Changz (6 pts)

a. Consider the weighted undirected graph below. Write the order in which the vertices are visited using the specified algorithm. In the event of a tie, visit the vertices in alphabetical order. The starting vertex for each traversal is already written for you.

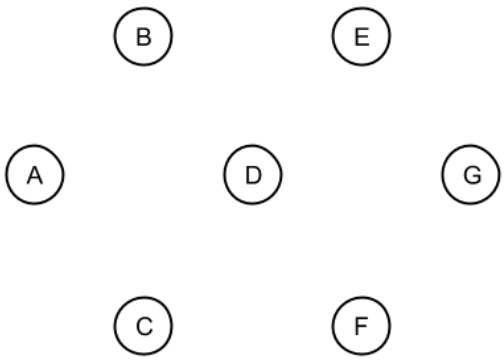


DFS: **G** ___ ___ ___ ___ ___ ___

BFS: **A** ___ ___ ___ ___ ___ ___

Dijkstra's: **A** ___ ___ ___ ___ ___ ___

b. For the same weighted undirected graph as in part **a**, draw the minimum spanning tree given by running Kruskal's algorithm. A blank graph has been provided below for your convenience. You need not include the edge weights.



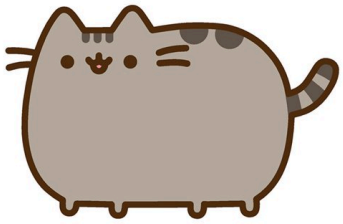
c. Suppose we wish to change the weights of the edges in the graph from part **a.** Is there an edge that will always be in the minimum spanning tree regardless of what its weight changes to? If so, write down the edge. If not, write *none exists*. No justification is necessary for either.

d. Draw a directed graph that has **only** the following 3 topological sorts. Each vertex in your graph must be "touched" by at least one edge. That is, every vertex in your graph must have at least one edge either arriving or departing from it. There may be more than one answer. You only need to label your vertices with letters.

Topological Sort 1	Topological Sort 2	Topological Sort 3
J. Go to Soda	J. Go to Soda	J. Go to Soda
Y. Pair program	C. Ace the quiz	C. Ace the quiz
C. Ace the quiz	Y. Pair program	D. Pull code to repo
D. Pull code to repo	D. Pull code to repo	Y. Pair program
G. Find bugs in code	G. Find bugs in code	G. Find bugs in code
K. Pass the tests	K. Pass the tests	K. Pass the tests

Chill-Out Corner

Remember to breathe. Don't be afraid to chill with the Shadow below. He's nice. Draw him something. Steven suggests a Pusheen (reference on the left).



3. Aunty Tares (4 pts)

You've just been hired by a hot new DNA sequencing startup called *DAN'S DNA SHACK*. Your job is to build software that accepts full DNA sequences of length ***L*** and allows users to quickly check if certain DNA subsequences of varying length are present in the full sequence. Fill in the `DNAMatcher` class below so that its functionality matches its comments. The `isSubsequence` method should run in $\Theta(R)$ time in the worst case and $\Theta(1)$ time in the best case, where ***R*** is the length of the subsequence. There is no limit on the runtime of `DNAMatcher's` constructor.

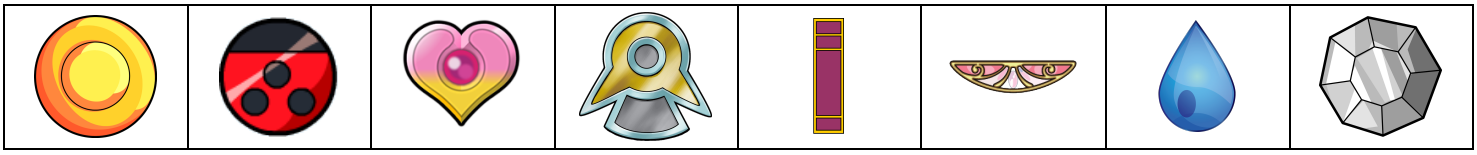
Hint: You may find the `Trie` class helpful. You may also find helpful the `String` class' `substring(int startIndex, int endIndex)` method, which gets a `String's` substring from `startIndex` (inclusive) to `endIndex` (exclusive).

```
public class DNAMatcher {

    ____________________________________________________________;

    ____________________________________________________________;

    /** Constructs a DNAMatcher object for a DNA sequence full. */
    public DNAMatcher(String full) {

        ____________________________________________________________

        ____________________________________________________________

        ____________________________________________________________

        ____________________________________________________________

        ____________________________________________________________

    }


    /**
     * Returns true if sub is a subsequence of the DNA sequence
     * given by full. Returns false otherwise.
     */
    public boolean isSubsequence(String sub) {

        ____________________________________________________________;

    }
}
```

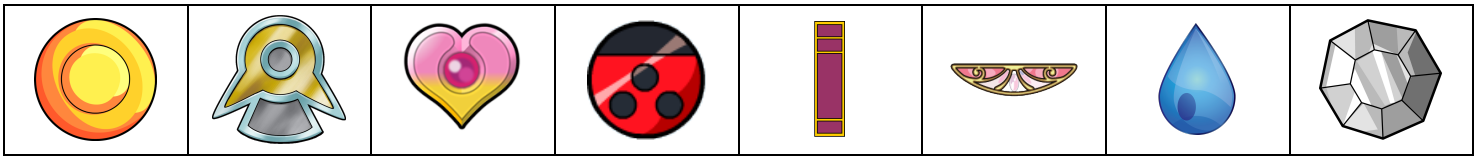
4. Green Eggs and Sam (4 pts)

Consider the following unsorted array, and the same array after **4** iterations of insertion sort as discussed in lab and lecture (where we sweep through the array from left to right, and move an element to its *relative* correct position by swapping). The first iteration starts at index 1 (the second element in the array). *Assume no two elements are equal. Insertion sort arranges the elements from least to greatest.*

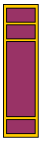
Unsorted:



After 4 iterations of insertion sort



For each row, fill in the correct bubble **fully** that corresponds to the relationship between the symbols. If there is insufficient information to determine the relationship between two objects, fill in the **?** bubble.

Symbol 1	<	>	?	Symbol 2
	○	○	○	
	○	○	○	
	○	○	○	

○ True / ○ False:  must be the smallest element in the array.

5. Lights Off Zeitsoff (4 pts)

a. We have regular expressions in the leftmost column of the table below. For each string, fill in the circle if the regular expression for that row fully matches that string.

For example, if the string **char** matches the regular expression `(jar)*migon`, shade the bottom left box. Fill in the bubbles completely. If none of the strings match a given regular expression, leave that row blank.

	char	**charmigon**	**jarmigon**	**charm**	**jar**	**migon**	**igon**
`charm`	○	○	○	○	○	○	○
`charm+`	○	○	○	○	○	○	○
`(char|jar)migon`	○	○	○	○	○	○	○
`[charm]*igon`	○	○	○	○	○	○	○
`(jar)*migon`	○	○	○	○	○	○	○

For parts **b.** and **c.** write a regular expression that will fulfill the conditions below. Be sure to escape special characters. The regular expressions are **not** Java Strings.

b. Match a valid group repo name, where a repo name is the string **"group"** followed by at least one digit.

Fully matched inputs: `"group17", "group200", "group000", "group0001"`
Non-matching inputs: `"groups17", "", "17", "group"`

Regex: _____

c. Match a valid US phone number. Match **only** the two formats shown below.
Hint: "`\s`" matches a single whitespace.

Fully matched inputs: `"(650) 123-6402", "408-244-1023"`
Non-matching inputs: `"1234567", "", "122-123", "12-123", "(65) 432-4596",`
`"(65)-432-4596", " (6) 432-4596"`

Regex: _____

6. Some-Ting Wrong (4 pts)

For each of the methods below, bound the overall runtime of the method using Big-Theta notation in terms of the input ***N***. If not possible, write "N/A". For full credit, your answer should be as simple as possible with no unnecessary leading constants or lower order terms.

_____
```
private static void f(int N) {
   for (int i = 0; i < N; i++) {
       for (int j = 0; j < N; j++) {
           linear(N); // runs in linear time with respect to input
       }
   }
}
```

_____
```
private static void g(int N) {
    if (n < 1) return;
    for (int i = 0; i < N; i++) {
       g(100);
    }
    g(N/2);
    g(N/2);
}
```

_____
```
private static void h(int N) {
    Random generator = new Random();
    for (int i = 0; i < N; i++) {
       if(generator.nextBoolean()) { // returns true with probability .5
           break;
       }
    }
}
```

_____
```
public static void i(int N) {
    TreeSet<Integer> t = new TreeSet<>(); // Uses a Red-Black Tree
    for (int i = N; i > 0; i /= 2) {
        t.add(i);
    }
}
```

7. A Supposedly Fun Ching I'll Never Do Again (8 pts)

a. Which of the following types of graphs **always** have a topological sort?

☐ Non-circular singly linked list
☐ Binary search tree with no parent pointers
☐ Undirected cyclic graph
☐ Directed acyclic graph
☐ Connected graph

b. For each of the following, indicate whether the statement is true or false. If true, provide a **brief** 1-2 sentence justification. If false, provide a graph as a counterexample.

○ True / ○ False: Suppose we run Dijkstra's algorithm on a graph **G**, starting from vertex ***v***. If the only negative edges in **G** are outgoing edges from ***v*** (and there are no negative cycles), then Dijkstra's will be able to find the shortest-paths from ***v***.

○ True / ○ False: Suppose we topologically sort some graph **G**. If vertex ***u*** is the first item in the sort, then for some other vertex ***v*** in the sort, we know there is a path from ***u*** to ***v*** in **G**.

○ True / ○ False: Given a graph **G** with unique edge weights, the shortest paths from some start vertex ***s*** to all other vertices are unique.

○ True / ○ False: Suppose you want to run breadth-first search and Dijkstra's algorithm on a graph **G**, whose edge weights are all the same positive number. Assuming the two algorithms make the same tie-breaking choices, the order in which nodes will be polled from the BFS queue is the same as the order in which nodes will be polled from the priority queue of Dijkstra's algorithm.

c. We are trying to find the minimum spanning tree (MST) of a graph where the edge weights may range between 0 and 255. Christine suggests that it is possible to find the MST in a time faster than $O(|E| \lg |E|)$. Is this true? If so, **briefly** explain how can this be done. If not, **briefly** explain why.

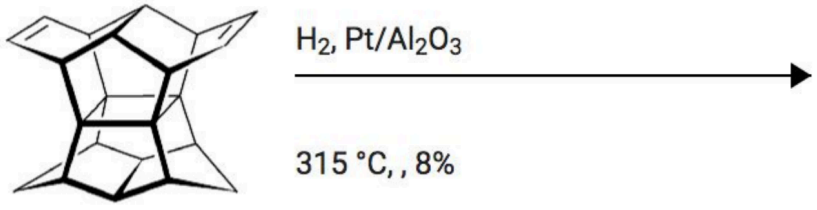
d. Steven needs to find the shortest path from his home to a Pusheen Conference. Let us define a graph ***G*** where there are |***V***| vertices representing different locations, and |***E***| edges representing the distances between locations. Steven's home is represented by the start vertex ***s***, while the Pusheen Conference location is represented by the target vertex ***t***. Steven will need to make one stop for gas on his route from ***s*** to ***t***. These gas stations are represented by an *array* of vertices. For example, the array $[g_1, g_2, \ldots, g_n]$ indicates that there are ***n*** gas station locations.

Consider how one would write an algorithm that would find the shortest path from ***s*** to ***t*** such that at least one gas station is on the path. Describe how your algorithm would work in 3-4 sentences. For full credit, your algorithm should run in $O((|V| + |E|) \lg |V| + n)$. We will give partial credit to algorithms that provide the correct solution, but run in slower time. You must provide your algorithm's runtime.

Runtime: _____

0. PNH

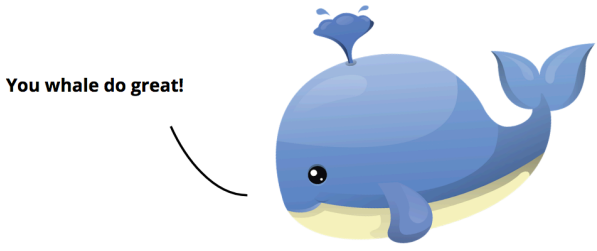
What is the name of the simplest hydrocarbon with full icosahedral symmetry? The final reaction step to synthesize this molecule is provided below.



8. JC and the Whale Pod (3 pts)

You are given a list of **unsorted** sublists containing `Comparable` elements. Using streams, complete the method below to sort the **unsorted** sublists and combine them into one final list containing all the sorted elements. If `lst` is empty, return an empty `LinkedList`.

```
public class StreamSort {

    /**
     * Merges two sorted lists, lst1 and lst2, into one sorted list.
     * Returns the merged list.
     */
    public static <T extends Comparable<T>> List<T> merge(List<T> lst1,
                                                          List<T> lst2) {...}

   /** Returns a sorted List containing all the elements in each list of LSTS. */
   public static <T extends Comparable<T>> List<T>
                                streamSort(LinkedList<LinkedList<T>> lsts) {
      return ____________________________________________________________
      ____________________________________________________________
      ____________________________________________________________
      ____________________________________________________________
      ____________________________________________________________
      ____________________________________________________________;
   }
}
```



This page is left intentionally blank.

Gigi and Diana encourage you to continue.

9. Big Baller Alex (6 pts)

Consider an undirected graph representing social relationships, where vertices represent people and edges represent connections. Person A and Person B are considered to be in the same network if a path exists in the graph from Person A's vertex to Person B's vertex. Implement the constructor and methods of the `SocialNetworks` class, which models these social relationships. You may assume everything on the reference sheet is imported.

A series of y `isSameNetwork` and x `addConnection` calls should run in $\Theta(x + y \cdot \alpha(x + y,\ x))$.

```
public class SocialNetworks {

    _____;

    _____;

    /**
     * Initializes a social graph initialized with users.size() people
     * whose names are stored in users.
     */
    public SocialNetworks(Set<String> users) {

        _____

        _____

        _____

        _____

        _____

        _____

        _____

    }
```

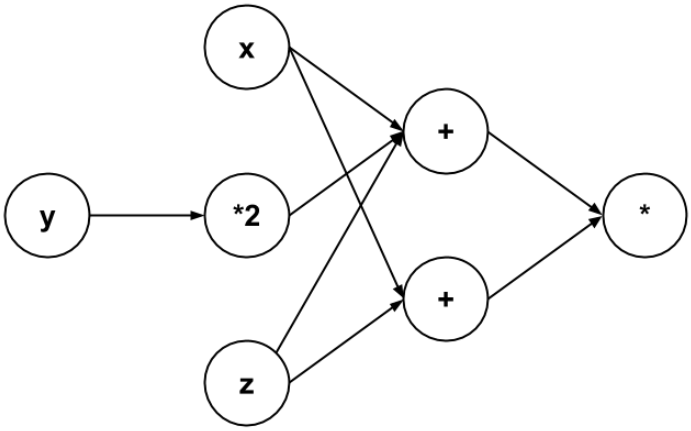
... (continued on next page)

Chirithy says to follow your dreams

```
    /**
     * Returns true if the users with names n and o are valid and in the
     * same network. Otherwise, returns false.
     */
    public boolean isSameNetwork(String n, String o) {
        ____________________________________________________________

        ____________________________________________________________

        ____________________________________________________________

        ____________________________________________________________

        ____________________________________________________________

    }


    /**
     * If n and o are names of valid users, add a connection between
     * them if such a connection doesn't exist already.
     * Does not modify the SocialNetwork if n or o are not valid names.
     */
    public void addConnection(String n, String o) {

        ____________________________________________________________

        ____________________________________________________________

        ____________________________________________________________

        ____________________________________________________________

    }

    public static void main(String[] args) {
        Set<String> users = new HashSet<>();
        users.add("Matt Owen");
        users.add("Alison Tanubrata");
        users.add("Wayne Li");

        SocialNetworks mySpace = new SocialNetworks(users);

        mySpace.addConnection("Matt Owen", "Alison Tanubrata");
        System.out.println(mySpace.isSameNetwork("Matt Owen",
                           "Alison Tanubrata")); // Should print true!
        System.out.println(mySpace.isSameNetwork("Matt Owen",
                           "Wayne Li")); // Should print false!
    }
}
```

10. Matt Sit Down, Be Humble (7 pts)

A computational network is a way to graphically model a mathematical function. For instance, given the function $f(x, y, z) = (x + 2 * y + z) * (x + z)$, the corresponding computational network would look as follows:



There are three kinds of nodes in a computational network: `InputNodes`, which represent variables and the value they take on, `MapperNodes`, which take the output of one node and maps it to a different value through its `Function's apply` method, and `CombinerNodes`, which can take an arbitrary number of inputs and combine them together through its `BinaryOperator's apply` method. Complete the code on the following pages such that the `main` method works as specified and your implementations match the behavior described in the comments.

Hint: You will need to use streams.

```
import java.util.Arrays;
import java.util.List;
import java.util.function.BinaryOperator;
import java.util.function.Function;

public class CompNet {
    private List<InputNode> input;
    private Node output;

    private interface Node {
        public double visit();
    }

    /** Sets the InputNodes for this CompNet. */
    public void setInputNodes(List<InputNode> lst) { ... }
    /** Sets the output Node for this CompNet. */
    public void setOutputNode(Node node) { ... }

... (continued on next page)
    /** Creates the CompNet described in the picture on the previous page. */
```

```
public static void main(String[] args) {
    CompNet c = new CompNet();
    InputNode x = new InputNode(-2.0);
    InputNode y = new InputNode(5.0);
    InputNode z = new InputNode(-4.0);
    MapperNode times2 = new MapperNode(y, (a) -> a * 2);
    CombinerNode add = new CombinerNode(
            Arrays.asList(x, times2, z), (a, b) -> a + b);
    CombinerNode secondAdd = new CombinerNode(Arrays.asList(x, z),
            (a, b) -> a + b);
    CombinerNode mult = new CombinerNode(Arrays.asList(add, secondAdd),
            (a, b) -> a * b);
    c.setOutputNode(mult);
    System.out.println(c.compute()); // Should print out -24.0
}

/** Returns the computation that this CompNet represents. */
public double compute() {

    ____________________________________________________________;
}


/**
 * A Node in a CompNet that represents a variable and the
 * value it takes on.
 * Returns its value upon being visited.
 */
private static class InputNode ____________________{

    private double value;
    public InputNode(double value) { ... }

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

}
```

... (continued on next page)

```
/**
```

```
     * A Node in a CompNet that combines the result of its inputs
     * through its BinaryOperator upon being visited.
     * The order in which inputs are combined does not matter.
     * If input is length 1, return the result of that one Node.
     * If input is input of length 0, return 0.0
     */
    private static class CombinerNode _____________________{

        List<? extends Node> input;
        BinaryOperator<Double> combiner;
        public CombinerNode(List<? extends Node> input,
                            BinaryOperator<Double> combiner) { ... }

        ____________________________________________________

        ____________________________________________________

        ____________________________________________________

    }

    /**
     * A Node in a CompNet that maps the result of its input
     * through its Function upon being visited.
     */
    private static class MapperNode _____________________{

        private Node input;
        private Function<Double, Double> mapper;
        public MapperNode(Node input, Function<Double, Double> mapper) { ... }

        ____________________________________________________

        ____________________________________________________

        ____________________________________________________

        ____________________________________________________

    }
}
```

This is a Chim Chim.

11. Christine the Coding Machine (10 pts)

The *k-nearest neighbors* problem is defined as follows. Given a set of N points and a query `Point` q, output the k points closest to q.

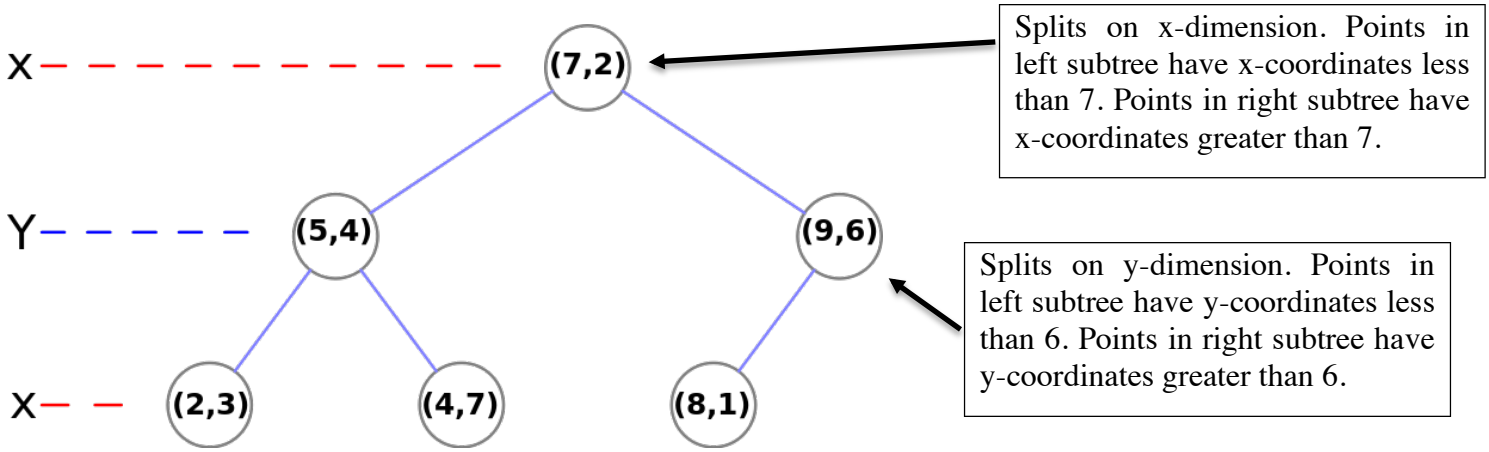
a. Complete the following method that finds the k-nearest neighbors by scanning through the input array, and keeping track of the k nearest points. The outputted array should be ordered with the closest point at index 0 and the farthest point at index $k - 1$. The code should run in $\Theta(N \lg k)$ time in the worst case, where N is the length of `arr`.

```
import java.util.PriorityQueue;

/** Returns the distance between a and b. */
private static int dist(Point a, Point b) { ... }

public static Point[] slowKNN(Point[] arr, Point q, int k) {
    PriorityQueue<Point> pq = ____________________________________________________;
    for (_____________________________________________________) {
        ____________________________________________________;
        if (_____________________________________________________) {
            ____________________________________________________;
        }
    }
    ____________________________________________________;
    for (_____________________________________________________) {
        ____________________________________________________;
    }
    ____________________________________________________;
}
```

b. You can complete this part without having done part **a**. A *k-d tree* is a data structure specialized for storing points. It can be used to optimize the k-nearest neighbors query.

More specifically, a k-d tree is a binary tree, where each node in the tree contains a point of dimension *k*. Each node stores the dimension it splits on (to determine how to divide the rest of the points). At each level of the tree, we will cycle through the dimensions, splitting on a different dimension for each level.

For example, given the point set [(2, 3), (5, 4), (9, 6), (4, 7), (8, 1), (7, 2)], (2 dimensional points), the corresponding k-d tree will look like below. First, we split on the x-dimension (which corresponds to dimension 0), then the y-dimension (which corresponds to dimension 1), and so forth.



Given an array of `Points`, we can recursively build up a balanced k-d tree with the following algorithm: At each level of recursion, we will find the median of the `Point` array based on the dimension index. We will create a new node containing that `Point`. Afterwards, we will recurse on all the `Points` before the median point, splitting on the next dimension, setting the results to be the left child. We do the same for the points after the median.

Complete the code below to construct a `KDTree` that stores the array of `Points` given by `arr`.

```
import java.util.PriorityQueue;
public class Point { ... }
public class KDTree {
    private TreeNode root;
    private int dim;

    private class TreeNode {
        Point item;
        TreeNode left;
        TreeNode right;
        int depth;
        private TreeNode(Point item, int depth,
                         TreeNode left, TreeNode right) {...}
    }

... (continued on next page)
```

```
    public KDTree(Point[] arr, int d) {
        dim = d;
        root = buildTree(arr, 0);
    }

    /**
     * Returns the i-th largest Point in arr based on the dimIndex-th dimension.
     * Partitions and modifies arr in-place accordingly.
     * Points less than the i-th largest will be to the left of index i.
     * Points greater than the i-th largest will be to the right of index i.
     * dimIndex must be between 0 and dim – 1.
     */
    private static Point partition(Point[] arr, int i, int dimIndex) { ... }

    /** Recursively builds a k-d tree. */
    public TreeNode buildTree(Point[] arr, int depth) {

        if (_____________________________________________________) {

            ____________________________________________________________;

        } else {

            ____________________________________________________________;

            ____________________________________________________________;

            ____________________________________________________________;

            for (_______________;_____________________________;_______________) {

                ____________________________________________________________;

            }

            for (_______________;_____________________________;_______________) {

                ____________________________________________________________;

            }

            ____________________________________________________________;

            ____________________________________________________________;

            ____________________________________________________________;

        }
    }
}
```

c. You can complete this part without having done part **b**. To perform a k-nearest neighbors query, we will search through the k-d tree in a manner similar to how you traversed your trie or TST in Autocomplete.

A `TreeNode` represents a bounding box in space (don't worry too much about it). We will keep a fringe of `TreeNodes` to visit, ordered by the distance from the query point to the bounding box that the `TreeNode` represents, calculated through the `dist` method below. We will also keep a `PriorityQueue` to keep track of the current nearest `Points` we have. In order to not have to explore the entire tree, we will only push `TreeNodes` onto the fringe if we still have not put ***k*** `Points` in the `PriorityQueue` OR the distance from the query point to the `TreeNode` is less than the distance from the query point to the ***k***th farthest `Point` in the `PriorityQueue`.

On the next page, complete the `kNN` method of the `KDTree` class below. The resulting array should be ordered with the closest point at index 0 and the farthest point at index ***k*** – 1. Assume there will always be at least ***k*** points in the k-d tree. Assume that all points in the `KDTree` have a unique distance from the query point.

Hint: Don't forget null checks

```
import java.util.PriorityQueue;
public class Point { ... }
public class KDTree {
    private TreeNode root;
    private int dim;
    ...
    /** Returns the distance between q and the bounding box represented by a. */
    private static int dist(Point q, TreeNode a) { ... }
    /** Returns the distance between a and b. */
    private static int dist(Point a, Point b) { ... }
```

... (continued on next page)

Chao chao!

```
    public Point[] kNN(Point q, int k) {
        PriorityQueue<TreeNode> fringe = new PriorityQueue<>((a, b) ->
                                                dist(q, a) - dist(q, b));
        PriorityQueue<Point> top = new PriorityQueue<>((a,b) ->
                                           dist(q, b) - dist(q, a));

        ____________________________________________________________;

        while (_____________________________________________) {

            ____________________________________________________________;

            ____________________________________________________________;

            ____________________________________________________________;

            if (_____________________________________________________) {

                ____________________________________________________________;

            }

            ____________________________________________________________;

            if (_____________________________________________________) {

                ____________________________________________________________;

            }

            if (_____________________________________________________) {

                ____________________________________________________________;

            }
        }

        ____________________________________________________________;

        for (_________________;_________________;_________________) {

            ____________________________________________________________;

        }

        ____________________________________________________________;

    }
}
```

That's it! Thank you for an awesome summer! ☺