UC Berkeley – Computer Science
CS61B: Data Structures

Final Exam, Spring 2017.

This test has 13 questions worth a total of 200 points, and is to be completed in 165 minutes. The exam is closed book, except that you are allowed to use three double sided written cheat sheets (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.**

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

These solutions are probably in good shape, but there are lots of tricky/subtle problems on the exam so it's

it's possible there are still errors. Last updated: 11:38 PM 5/6/2019

Video walkthrough: Nonexistent, sorry :(

Signature: EGG

#	Points	#	Points
0	0.5	7	23
1	12	8	0
2	28	9	16
3	14	10	16
4	12	11	12
5	16	12	12
6	12	13	26.5
		TOTAL	200

Name: _Egg_______________

SID: __6__________________

Three-letter Login ID: __ege___

Left SID: ___N/A_______________

My left neighbor has ID: __N/A__

Right SID: ___N/A_______________

My right neighbor has ID: _ N/A_

Exam Room: ___N/A_______________

Tips:

- There may be partial credit for incomplete answers. Write as much of the solution as you can, but we may deduct points if your answers are 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.
- **See the coding reference sheet on the last page for potentially useful data structures.**
- 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 in 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 which involve filling in a ○ or ☐, **please fill in the shape completely.**

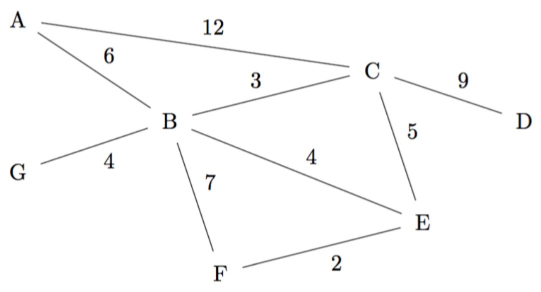
Optional. Mark along the line to show your feelings on the spectrum between ☹ and ☺.

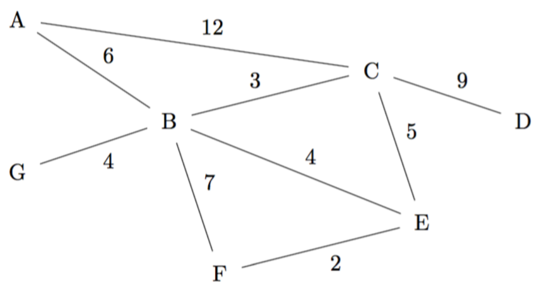
Before exam: [☹_______________☺].
After exam: [☹_______________☺].

0. So it begins (0.5 points). Write your name and ID on the front page. Write the exam room. Check the IDs of your neighbors. Write the given statement. Sign. Write your login in the corner of every page. Enjoy your free 0.5 points ☺.

1. Mystery Spanning Tree 3000 (12 points).

a) (4 pts) For the graph below, list the edges in the order they're added to the MST by Kruskal's and Prim's algorithm. Assume Prim's algorithm starts from vertex A. Assume ties are broken in alphabetical order (i.e. the edge $\overline{AB}$ would be considered before $\overline{AC}$). Denote each edge with alphabetical overbar notation $\overline{AB}$, which represents the edge from A to B. You may not need all blanks. For your convenience, the graph is printed twice (to make running algorithms easier).





Prim's algorithm order: AB BC BE EF BG CD

Kruskal's algorithm order: EF BC BE BG AB CD

b) (2 pts) Is there any vertex for which the shortest paths tree (SPT) is the same as your Prim MST above?

[✓ Yes], and it's ____B_____ (or A or the 'hipster vertex' G) ○ No

c) (6 pts) For the following propositions, fill in true or false <u>completely</u> **and provide a brief explanation**. For a proposition that is false, a counter-example suffices. Assume all edge weights are unique.

[✓ True] / ○ False: Adding 1 to the smallest edge across any cut of a graph G must change the total weight of its minimum spanning tree.

Since all edge weights are unique, either this smallest edge (now with weight +1) is included, or this smallest edge is not included, and some larger edge takes its place. Either way, total weight increases.

○ True / [✓ False]: The shortest path from vertex A to vertex B in a graph G is the same as the shortest path from A to B using only edges in T, where T is the MST of G.

No, consider vertices C and E in the graph above.

[✓ True] / ○ False: Given any cut, the maximum-weight crossing edge is in the maximum spanning tree.

Yes. We can simply use the proof of the cut-property from class, but replacing "larger" with "smaller".

That is: Suppose it were not. In that case, we could add it to the max spanning tree and a cycle would result. To get rid of the cycle in the MaxST, we could remove the smallest edge in the cycle (which is by definition not the maximum-weight crossing edge) and end up with a larger spanning tree.

2. Sorting (28 points).

a) (2 pts) How many inversions are in the list [A, D, B, W, K] assuming we want the list sorted alphabetically?

2 (D and B, W and K)

b) (3 pts) Suppose we want to sort the array [5, 6, 10, 17, 14, 12, 13] using in-place heapsort. Give the array after heapification and a single remove-max operation.

[14, 10, 13, 6, 5, 12, 17]

c) (14 pts) Suppose we have N items we want to sort. For each scenario below, pick the "best" sort to get them into sorted order. Assume for all scenarios that N is very large. **There may be multiple correct answers,** and the correct answer may even be ambiguous. Give the running time **for the absolute worst case** in the right-most column as a function of N. Running time may not be the only consideration for "best". In all cases, assume we're using Java.

Choose from among 1: Insertion sort, 2: Merge sort, 3: Quicksort (with Hoare partitioning), and 4: LSD radix sort. You may not need to use all four answers. Assume that we want stability when potentially useful. **Give your answer to "Best Sort" as a number.**

Below are listed our answers. We may give other answers credit than those listed below. People looking at these solutions in future semesters: Argue with each other about why we picked our answers. Try to decide if you can find other reasonable answers than the ones we listed and under what assumptions (the array of 10 strings is particularly interesting). Also, hello from May 2017! It feels so … now, right now, but I bet for you it feels like the past, possibly a long time ago.

Scenario (i.e. What You're Sorting)	Best Sort	Running Time
Array of N integers whose max value k is a constant. Do not include k in your runtime.	4	O(N)
Array of N `BigIntegers`[1] whose max value is N^3. Assume comparison takes `log(N)` time.	4	O(N Log N)
Array of N objects that implement `Comparable`, assuming comparison takes constant time.	2	O(N Log N)
Doubly linked list of N objects that implement `Comparable`, assuming constant time comparison and all variables `public`.	2	O(N Log N)
Array of 10 `Strings` of length W. Give runtime in terms of W.	1	O(W)
Array of N objects that implement `Comparable` with $\Theta(\sqrt{N})$ inversions, assuming compare takes constant time.	1	O(N)
Array of N objects that implement `Comparable` with $\Theta(N^2)$ inversions, assuming compare takes constant time.	2	O(N Log N)

d) (9 pts) We call a sort **monotically improving** if the number of inversions never increases as the sort is executed. Which sorts from the list below are monotically improving? Assume that all sorts are as presented during lecture on arrays. Assume insertion sort and selection sort are in-place. Assume heapsort is in-place and that the array acts as a max heap. Assume that Quicksort is non-randomized, uses the leftmost item as pivot, and uses the Hoare partitioning strategy (i.e. using "smaller than" and "bigger than" pointers) from lecture.

■ Insertion sort ■ Selection sort ☐ Heapsort ■ Quicksort ☐ LSD Sort ■ MSD Sort

3. Traversals (14 points). Suppose we have an `NAryIntTree`, defined as shown below. Any node may have any number of children. If a node is a leaf, `children` is `null`. Assume that `children[i]` is never `null` for any `i`.

[1] A `BigInteger` is an "immutable arbitrary precision integer." It can represent any integer, not just those that fit into 32 bits.

a) (6 pts) Fill in the `printTreePostOrder` method below, which prints the values of the tree in postorder, with one `val` per line. Your solution **must be recursive** and take linear time in the number of nodes.

```
public class NAryIntTree {
    private Node root;
    public class Node {
        public Node[] children;
        public int val;
    }
    public void printTreePostOrder() {
        printTreePostOrderHelper(root)
    }

    public void printTreePostOrderHelper(Node x) {
        if (x.children != null) {
            for (int i = 0; i < x.children.length; i += 1) {
                 printTreePostOrderHelper(x.children[i]);
            }
        }
        System.out.println(val);
     }
   /* ... */
}
```

b) **(8 pts)** Fill in the code below which prints out the values of the tree in level order with one `val` per line. Your solution **must be iterative** and take linear time in the number of nodes for any tree.

```
    private void printTreeLevelOrder() { // is a method of NAryIntTree
        Queue<Node> fringe = new Queue<>();
        fringe.enqueue(root);
        while (fringe.size() > 0) {
            Node x = fringe.dequeue();
            System.out.println(x.val);
            if (x.children != null) {
                for (int i = 0; i < x.children.length; i += 1) {
                    fringe.enqueue(x.children[i]);
                }
            }
        }
}
```

4. Algorithms and Data Structures (12 points).

a) (4 pts) In class we primarily considered two graph representations: the adjacency list and the adjacency matrix. Antares suggests that we can improve the performance of Dijkstra's algorithm with a third graph representation he calls an "adjacency heap". For each vertex `v`, `v`'s adjacency heap stores all of `v`'s neighbors in a heap ordered by edge weight, so that the smallest edge adjacent to `v` is at the root of its heap. Naturally, Antares stores these heaps as arrays. Antares reasons that by considering small edges first, Dijkstra's will be able to complete faster.

Will using an adjacency heap result in better, equivalent, or worse **asymptotic** runtime performance for Dijkstra's algorithm than using a regular adjacency list? Assume that we only care about worst case asymptotic performance. Briefly justify your answer.

○ Adjacency heap is better [✓ Performance is the same] ○ Adjacency heap is worse

Justification: Vertices get dequeued in the same order, so only difference is the time to iterate through adjacency heap vs. list. Iteration takes the same amount of time for both assuming Antares just iterates through the array. Or even if he deletes from the PQ, the overall runtime is still the same (see partial credit answer below).

Or for partial credit:

○ Adjacency heap is better ○ Performance is the same [✓ Adjacency heap is worse]

Justification: Vertices get dequeued in the same order, so only difference is the time to iterate through adjacency heap vs. list. Since Antares wants to go through "small edges first", perhaps this means that he iterates through in decreasing order, which must takes D log D time, where D is the degree of the vertex. In the worst case, the totality of these iterations costs E log V. This does not change the runtime of Dijkstra's algorithm asymptotically, so is not correct, but we gave partial credit for recognizing (as long as your answer was very clear) that Antares's idea was simply going to slow things down with no benefit whatsoever.

b) (4 pts) Suppose Antares has conjured up the **Gulgate Priority Queue (GPQ)**. Given a GPQ containing N elements, the worst-case running time for insertion, deletion, and change-priority are given as follows: Insertion: $\Theta(N)$, Deletion: $\Theta(N)$, Change-Priority: $\Theta(1)$.

Suppose we run the implementation of Dijkstra's algorithm provided in class (where every vertex is initially inserted into the PQ with infinite priority) using a GPQ on a graph with V vertices and E edges. What is the worst case runtime of Dijkstra's? **Give your answer in big O notation in terms of V and E**. Assume that E >> V (this means E is much greater than V).

	# Ops	Runtime per op	Total Runtime
Insertion	O(V)	O(V)	O(V^2)
Deletion	O(V)	O(V)	O(V^2)
Change Priority	O(E)	O(1)	O(E)

Runtime: O(E+V^2), but since E is bounded above by V^2, it's fine to also say O(V^2). Simplifications which remove V^2 log V are not correct, e.g. O(E log V) and O(E log E): Suppose we build graphs where E = $V^{1.5}$. In this case, E is certainly much larger than V as both grow very large, but the function E log V would grow more slowly than V^2 log V.

c) (4 pts) Suppose Antares has also created a **Xelha Quick Union (XQU)** to check if two vertices are connected while running Kruskal's. Given that there are N items in an XQU, the running time for XQU operations is as follows: Constructor: $\Theta(N)$, Union: $\Theta(N \log N)$, Is-Connected: $\Theta(\log N)$

Suppose we run the implementation of Kruskal's algorithm as presented in class using a XQU and a heap-based priority queue. Recall that in our version of Kruskal's from class, all edges are initially inserted into a regular heap-based priority queue and removed one by one, and added to the MST so long as there are no cycles. What is the worst case runtime of Kruskal's algorithm? **Give your answer in big O notation in terms of <u>V and E</u>**. Assume that E >> V.

	# Ops	Runtime per op	Total Runtime
insert into PQ	O(E)	O(log E)	O(E log E)
remove from pQ	O(E)	O(log E)	O(E log E)
union	O(V)	O(V log V)	O(V^2 log V)
isConnected	O(E)	O(log V)	O(E log V)
constructor	O(1)	O(V)	O(V)

Overall runtime is O(E log E + V^2 log V + E log V + V). We can simplify with the following observations:

- Observation 1: Since E >> V, the V term is irrelevant.
- Observation 2: E is O(V^2) since at most every vertex is connected to every vertex.
- Observation 3 : Using observation 2, E log E is O(E log V^2) which is O(2E log V) which is O(E log V). Thus E log E and E log V are redundant.

Using observation 1 and 3, this means we can simplify to O(E log V + V^2 log V), which can be simplified to O(V^2 log V) using observation 2 again, since E may grow more slowly than V^2.

Simplifications which remove V^2 log V are not correct, e.g. O(E log V) and O(E log E): Suppose we build graphs where E = $V^{1.5}$. In this case, E is certainly much larger than V as both grow very large. However, the function E log V would grow more slowly than V^2 log V.

5. Potpourri (16 points).

a) (6 pts) We learned in lecture and in lab that we can use an array to compactly store a min/max heap, with formulas to calculate the parent, left child, and right child given a node. Now suppose we want to store a ternary heap, where every node has 0, 1, 2, or 3 children. Would the compact array representation work? If your answer is **yes**, give formulas on the **left** to calculate the parent, left child, middle child, and right child. If your answer is **no**, explain why on the **right**.

`Assuming root is at index:        0`
`public int parent(int k) { return (k-1)/3;}`
`public int left(int k)   { return k*3+1;}`
`public int middle(int k) { return k*3+2;}`
`public int right(int k)  { return k*3+3;}`
	Impossible, because:

`Assuming root is at index:        1`
`public int parent(int k) { return Math.round(k/3);}`
`public int left(int k)   { return k*3 - 1;}`
`public int middle(int k) { return k*3;}`
`public int right(int k)  { return k*3 + 1;}`
	Impossible, because:

b) (2 pts) In class, we said that anytime you override `equals`, you must also override `hashCode`. Suppose the `Yarg` class overrides the `equals` method, but does not override `hashCode`. Suppose that `yargSet` is a `HashSet<Yarg>`. What are the potential direct consequences of not overriding `hashCode`?

[✓ True] / False: `yargSet.contains()` may return an incorrect result.
True / [✓ False]: `yargSet.contains()` runs a much higher risk of taking linear time.

c) (6 pts) If we wanted to build a generic `TrieSet` that could hold many different types, we'd need to require all such types to implement some interface, much like items in a `TreeSet` must implement the `Comparable` interface (shown below). Give a declaration of an appropriate interface and describe any methods with comments. Provide useful names for your methods and interface (not silly ones, sorry). You may not need all blanks.

```
public interface Comparable<Item> {
    // Returns negative int if this < x, positive if this > x, 0 if equal.
    int compareTo(Item x);
}
public interface HasDigits {
    /** Returns the ith digit. */
    public int digit(int i);
}
```

Other schemes for deconstructing an object into sub-pieces that are indexable could work as well. The digits don't necessarily need to be comparable to work for Trie construction. Alternate stranger possibilities like prefix testing methods are also possible, though unwieldy.

d) (2 pts) Suppose the creator of a new `DogPicture` class is deciding whether or not to implement interface X, where X is the interface from part c. What is the primary consideration of the creator? “Will somebody ever want to build a `Trie` of `DogPictures`” is not enough of an answer.

For me the best answer is “Will anyone want to do prefix operations with DogPictures”, but other answers are possible.

6. **Stocks (12 pts).** Define the price of a stock as the price of its most recent trade. Suppose we want to track the highest priced stocks at the end of the day using the code below. Assume the `MaxPQ` is heap based and uses the same approach as detailed in lecture (where insertion and deletion operations take worst case logarithmic time). Assume all stock names are unique. Assume `STOCK_LIST` is a list of `Stock` objects with prices equal to yesterday's final price. A stock may be traded multiple times in one day.

```
public static List<Stock> getMostExpensiveStocks(int k) {
      MaxPQ<Stock> rankedStocks = new MaxPQ<>();
      HashMap<String, Stock> nameToStock = new HashMap<>();
      addAllStocks(STOCK_LIST, nameToStock);  //assume nice hashcode spread
      addAllStocks(STOCK_LIST, rankedStocks); //uses bottom up heapification
      while (marketIsStillOpenToday()) {      //market closes at 5 PM
          Trade t = getNextTrade();           //waits if no trade available
          Stock s = nameToStock.get(t.name);  //assume key always in map
          s.price = t.price;                 //may be higher or lower
      }
      ArrayList<Stock> returnStocks = new ArrayList<Stock>();
      for (int i = 0; i < k; i += 1) {        //assume k <= s
           returnStocks.add(rankedStocks.delMax());
      }
      return returnStocks;
}
```

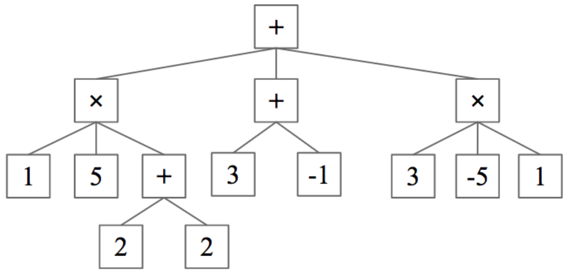
Where the `compareTo` method of `Stock` is defined as `public double compareTo(Stock s) { return this.price - s.price; }` where `price` is an integer.

a) Which of the correctness or compilation issues listed below are present in this code? Check all that apply. If you believe there are compilation errors, consider the other boxes assuming the compilation errors are fixed. Assume k is smaller than the number of stocks.

☐Compilation: The method is supposed to return a `List`, but returns an `ArrayList`.
☐Compilation: The `HashMap` cannot point to items inside of the `MaxPQ` because the `MaxPQ`'s instance variables are private.
☐Correctness: The algorithm actually returns the k cheapest stocks.
☐Correctness: The algorithm may return a list with duplicates if the same stock is traded multiple times.
■Other (explain): Mutating objects in a heap is very bad and can yield the wrong answer OR compareTo returns a double, not an int (this second answer wasn't intended, but if you spotted it, cool).

b) What is the worst case runtime and space complexity of the code above, assuming we fix only any compilation errors you identified in part a? Give your answer in O notation. Your bounds should be as tight as possible with no unnecessary lower order terms or constant factors. Give your answers in terms of S, T, and k, where S is the number of stocks, T is the number of trades, and k is the given argument.
Runtime complexity: $O(S + T + k \log S)$ or $O(S^2 + ST)$ if we take into account the fact that hashing COULD be bad.
Space (a.k.a. memory) complexity: $O(S)$

7. Arithmetic Tree (23 pts). An arithmetic tree is a tree that stores an arithmetic expression. For this problem, assume all nodes are either multiplication (represented with ⨉), addition (represented with +), or a number (represented as a written integer). For example, the following tree represents (1 ⨉ 5 ⨉ (2 + 2)) + (3 + -1) + (3 ⨉ -5 ⨉ 1), which would evaluate to 7.



Your job is to fill out the code below such that `evaluateTree(Node tree)` evaluates the arithmetic tree rooted at `tree` to its correct value. For example, if `evaluateTree` were applied to the ⨉ node at the top left of the figure above, it would return 20. Multiplication and addition operator nodes can have any number of children. You do not need to check for bad inputs (e.g. null children). You may find it easier to work your way from the end of the problem back to the front. You may not need all blanks.

```
public abstract class Node {
    public List<Node> children;
    public abstract void processNode(Stack<Integer> stk);
}

public class ArithmeticTreeEvaluator {
    public static int evaluateTree(Node tree) {
        Stack<Integer> stk = new Stack<>();
        evaluateTreeHelper(tree, stk);
        return stk.pop();
    }


    private static void evaluateTreeHelper(Node tree, Stack<Integer> stk) {
        for (Node c : tree.children) {
            evaluateTreeHelper(c, stk);
        }
        tree.processNode(stk);
    }
}
```

If you're stuck on this problem, come back later!

```
public abstract class OperatorNode extends Node {
    public int numArgs() { /* Returns number of args for this node. */ }
    public abstract int apply(int arg1, int arg2);
    public void processNode(Stack<Integer> stk) {
        int res = stk.pop();
        for(int i = 1; i < numArgs(); i += 1) {
            res = apply(res, stk.pop());
        }
        stk.push(res);
    }
}

public class ArgNode extends Node {
    public int value;
    @Override
    public void processNode(Stack<Integer> stk) {
        stk.push(value);
    }
}

/* Don't overthink this! */
public class MultiplicationNode extends OperatorNode {
    @Override
    public int apply(int arg1, int arg2) {
        return arg1 * arg2;
    }
}

public class AdditionNode extends OperatorNode {
    @Override
    public int apply(int arg1, int arg2) {
        return arg1 + arg2;
    }
}
```

8. PNH (0 points). This United States President won office with the smallest fraction of the popular vote in the history of United States presidential elections.

John Quincy Adams, who only got 30.9% of the popular vote in 1824 and didn't even get the majority of the electoral college. Instead, the House of Representative picked him. Dang, can you imagine?

9. Asymptotics (16 points). For each of the code snippets below, give the **best** and *worst case* runtimes in terms of N. Give the **best runtimes** in the column to the **left**, and the *worst* in the column to *the right*.

Best: Worst:

$\Theta(N^2)$ $\Theta(N^2)$

```
public static void f1(int N) {
    if (N == 0) { return; }
    f1(N / 2);
    f1(N / 2);
    g(N); // runs in Θ(N²) time
 }
```

$\Theta(1)$ $\Theta(N)$

```
public static int f2(String[] x, int i) {
    int N = x.length;
    int total = 0;
    try {
        while (i < N) {
            total += x[i].length();
            i += 1;
        }
    } catch(NullPointerException e) {
        x[i] = "null";
        total += f2(x, i);
    }
    return total;
}
```

$\Theta(N)$ $\Theta(N)$

Assume `t` is a binary `IntTree` with N nodes:

```
public static void f3(IntTree t) {
    t.value = t.value * 2;
    if (t.left != null)  {  f3(t.left);   }
    if (t.right != null) {  f3(t.right);  }
}
```

$\Theta(1)$ $\Theta(2^N)$

Assume `t` is a binary `IntTree` with N nodes:

```
public static void f4(IntTree t) {
    t.value = t.value * 2;
    if (t.left != null)  {  f4(t.left);   }
    t.right = t.left;
    if (t.right != null) {  f4(t.right);  }
    t.left = t.right;
}
```

10. Return of the XelhaTree (16 points). Write a function `validXelhaTree` which takes an `IntTree` and a `List` and returns true if the `IntTree` is a `XelhaTree` for the list. You may not need all lines. A `XelhaTree` is valid if it obeys the min heap property, and if an in-order traversal of the `XelhaTree` yields the list of items passed to `createXelhaTree` (in the same order). One line if statements with `{}` on the same line are fine. You may not need all the blanks. Assume there are no duplicates.

```
public class XelhaTreeTest {
    public static class IntTree {
        public int item;
        public IntTree left, right;
    }
    public static IntTree createXelhaTree(List<Integer> x) { ... }
    /** If x is null, returns largest possible integer 2147483647 */
    private static int getItem(IntTree x) {
        if (x == null) { return Integer.MAX_VALUE; }
        return x.item;
    }
    public static boolean isAHeap(IntTree xt) {
        if (xt == null) { return true; }
        if (xt.item > getItem(xt.left)) { return false; }
        if (xt.item > getItem(xt.right)) { return false; }
        return isAHeap(xt.left) && isAHeap(xt.right);
    }
    public static void getTreeValues(IntTree xt, List<Integer> treeValues){
        if (xt == null) { return; }
        getTreeValues(xt.left, treeValues);
        treeValues.add(xt.item);
        getTreeValues(xt.right, treeValues); }
    }
    public static boolean validXelhaTree(IntTree xt, List<Integer> vals) {
        List<Integer> treeValues = new ArrayList<Integer>();
        /* getTreeValues adds all values in xt to treeValues */
        getTreeValues(xt, treeValues);
        return isAHeap(xt) && treeValues.equals(vals); }
```

11. MaxPQ (12 points). Complete the implementation of MaxPQ using data structures from the reference sheet on the last page of the exam. You may not need all blanks. Write at most one statement per line.

```
public class MaxPQ<Item extends Comparable<Item>> {
    private MinPQ<Item> pq;

    public MaxPQ() {
        pq = new MinPQ<Item>(new ReverseComparator());
        ____________________________________________
    }

    public class ReverseComparator implements Comparator<Item> {
        public int compare(Item i1, Item i2) {
            return i2.compareTo(i1);
                // note: -i1.compareTo(i2) fails if compare returns
                //       Integer.MIN_VALUE
        }
    }

    public Item delMax() {
        return pq.delMin();
    }

    public void insert(Item x) {
        pq.insert(x);
    }
}
```

Welcome to the Chill Out Zone. [a cool place for friends!!!!]

12. Danger and Optimization (12 points).

a) (5 pts) Suppose you provide a computing service where users can upload lists of integers and receive back the numbers in sorted order. Which sorts below would be appropriate to choose for this task, assuming you want to prevent users from submitting inputs that either result in terrible[2] runtime or cause an exception? **Assume you are using Java**.

[✓ Appropriate] / Inappropriate : Merge Sort
Appropriate / [✓ Inappropriate] : Insertion Sort – Can be N^2!
[✓ Appropriate] / Inappropriate : Quicksort using Hoare partitioning and that starts by shuffling
[✓ Appropriate] / Inappropriate : LSD – An earlier version of this had LSD as inappropriate, but given that we're using Java integers, LSD sort is fine since items are of a short maximum width.
[✓ Appropriate] / Inappropriate : Recursive MSD – An earlier version of this had LSD as inappropriate, but given that we're using Java integers, LSD sort is fine since items are of a short maximum width.

Note: Quicksort can also be N^2, but probability is effectively zero, and no input the user provides has any chance of causing slow runtime. It's literally not a thing worth worrying about unless you are worried about asteroids killing you on the way to school.

b) (5 pts) Suppose you provide a service where users can upload a list of names (stored as Java Strings) and it will return the list of all unique Strings. Which set implementations below would be appropriate to choose for this task, assuming you want to prevent users from submitting inputs that either result in terrible runtime or cause an exception?

[✓ Appropriate] / Inappropriate : 2-3 Tree based Set
[✓ Appropriate] / Inappropriate : LLRB based Set
Appropriate / [✓ Inappropriate] : Hash based Set – Someone trying to embarrass you could precalculate a bunch of Strings with same hashCode and submit them to your service.
[✓ Appropriate] / [✓ Inappropriate] : Trie based Set – This really depends on your Trie implementation. If you're using an array of links like in the optional textbook, then you're going to use all your memory and get OutOfMemory errors. If you're using a HashMap or TreeMap for links, things will be slower and safer, though probably appropriate (but honestly it's a pretty tough analysis).
Appropriate / [✓ Inappropriate] : TST based Set – If you're using something like 16 bit Unicode (as in Java), someone can insert a sequence of strings so that insertions into the TST take 2^{16} times as long as is necessary.

c) (2 pts) Suppose you provide a service where users can upload their own custom graphs and a start vertex, and you will find the list of all vertices reachable from the start. Which graph search algorithms would be

[2] By terrible we mean: Imagine you are demoing your website and want to impress someone with its speed. Does it run so slow that you are embarrassed? If so, that is terrible.

appropriate to choose for this task, assuming you want to prevent users from submitting inputs that either result in terrible runtime or cause an exception?

○ Appropriate / [✓ ● Inappropriate] : Recursive DFS – Spindly graphs can cause stack overflow exception
[✓ ● Appropriate] / ○ Inappropriate : BFS

d) (0 pts) What should Josh name his future kid (assume female if you want a gender specific name)?

Two popular answers were "Joshina" (or similar) and anything starting with a T. Actually, my dad's work email was thug@...

13. Reductions (26.5 points). Often in computer science, problems are just other problems in disguise. Complete each problem below according to the directions given. Many of these problems are very challenging.

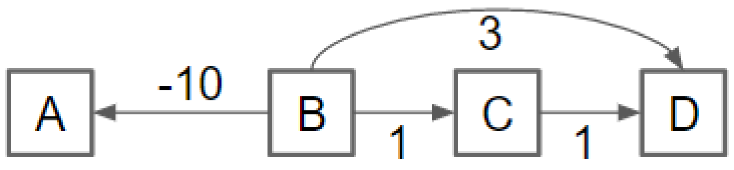
a) (3 pts) Describe an algorithm to find a **maximum** spanning tree. Your algorithm must use Kruskal's as a "black box," that is, without any modifications. Your answer should be brief.

Build a copy of the graph with every edge weight negated. Run Kruskal's.

b) (5 pts) Suppose you want to find the SPT of a graph, but where you redefine the total cost of a path as follows. Let `cost(List<Edge>)` be the sum of the weights of the edges, plus the number of edges. In other words, we want to run Dijkstra's taking into account not just the weights of the edges, but also the number of edges. Describe an algorithm to find this shortest paths tree. Your algorithm must use Dijkstra's as a "black box". Your answer should be brief.

Build a new copy of the graph with +1 added to every edge. Run Dijkstra's.

c) (5 pts) Dijkstra's algorithm sometimes fails on graphs with negative edges. Suppose we have a graph G with a single negative edge with weight -Q, and we want to find the shortest path. Suppose we construct a new graph G' where every edge has Q added to its weight. If we run Dijkstra's on G', is the resulting shortest paths tree always a correct shortest paths tree for G? If yes, explain why. If no, provide a counter-example.

○ Yes, because:	[✓ No, counter-example:]  SPT with B as source is not correct if we add 10 to every edge (in the new graph G, it'll prefer the path from B to D directly, but that's not actually the shortest path in the original graph from B to D)

d) (6 pts) Suppose that we're using a programming language Zulg where instead of comparison we have a *zelch* operation. Suppose that we prove that "puppy, cat, dog"[3] requires $\Omega(N \log \log N)$ *zelch* operations. Assume that zelch takes constant time. For each of the following statements, determine whether the answer is false, true, or the answer depends on whether P = NP.

We're now at the *really* hard part of the test! Note: P=NP was an "extra" topic, but even if you don't know what it is, it was irrelevant to this problem.

[✓ True] / ○ False / ○ P=NP?: A *zelch* based sort requires **at least** $\Omega(N \log \log N)$ *zelch* operations.
Yes, same exact argument as in class about comparison based sorting lower bound.

○ True / [✓ False] / ○ P=NP?: Sorting an array in Zulg requires $\Theta(N \log \log N)$ time in the worst case.
No, it might still be possible to do radix sort in Zulg even without comparison.

○ True / [✓ False] / ○ P=NP?: The optimal sorting algorithm in Zulg requires $O(N \log \log N)$ time in the worst case.

Just because we made a lower bound for the number of zelch operations doesn't mean an algorithm that meets the bound necessarily exists. For example, we can prove that 3SUM requires $\Omega(N)$ array access in Java, but that doesn't mean that the optimal 3SUM algorithm is O(N) in the worst case!

○ True / [✓ False] / ○ P=NP?: All sorting algorithms in Zulg require $O(N \log \log N)$ time in the worst case.
We don't even know if the optimal sorting algorithm is O(N log log N) so we certainly can't say the same about all sorting algorithms. For example, we might be able to write Bogosort in Zulg and it's certainly not O(N log log N) in the worst case.

[3] Recall that "puppy, cat, dog" is a game from lecture where we have N boxes, each containing a unique object (e.g. a puppy, a cat, and a dog) of known size, and our job is to determine which box contains which object.

e) (6 pts) Suppose we have the abstract data type `MinimumPQ`, defined as an interface in Java as shown below:

```
public interface MinimumPQ<Item extends<Comparable<Item>> {
   public void add(Item x);
   public Item removeMin();
   public Item min();
}
```

For each statement below, state whether it is true, false, or "depends on whether P = NP". Assume that all implementations are correct.

[✓ True] / False / P=NP?: There exists a possible MinPQ implementation for which add requires $\Theta(\log\log N)$ time in the worst case.
Trivial, just stick the item at the front and then do dummy operation that takes log log N time.

[✓ True] / False / P=NP?: There exists a possible MinPQ implementation for which removeMin requires $\Theta(\log\log N)$ time in the worst case.
Trivial, maintain ordered array (making insert slow), then removeMin grabs front item and does dummy operation that takes log log N time.

True / [✓ False] / P=NP?: There exists a possible MinPQ implementation for which add and removeMin require $\Theta(\log\log N)$ time in the worst case.
Any comparison based sort must take N log N time. If you can do both in log log N time, you are disobeying sort lower bound, since you could use this magical MinPQ to sort N items in N log log N time which is less than N log N.

[✓ True] / False / P=NP?: There exists a possible MinPQ implementation for which add requires $\Theta(1)$ time in the worst case. Same as above.

[✓ True] / False / P=NP?: There exists a possible MinPQ implementation for which removeMin requires $\Theta(1)$ time in the worst case. Same as above.

True / [✓ False] / P=NP?: There exists a possible MinPQ implementation for which add and removeMin require $\Theta(1)$ time in the worst case. Same as above.

f) (1.5 pts) Consider the `LongestPath` problem, i.e. given a graph, does there exist a path with total weight k or greater? Suppose that we prove that `LongestPath` cracks 3SAT (i.e. 3SAT reduces to longest path). For each of the following statements, determine whether the answer is false, true, or the answer depends on whether P = NP. Let N be the number of edges.

Note: The definition of k is a bit ambiguous, but the intent of the question was clear, I hope. Sorry. These very last 1.5 points were originally meant to be tiny amount of points for those who learned the extra topics, but now they are free points for everybody!

○ True / ○ False / [✓ P=NP?]: There exists an algorithm to solve `LongestPath` in $O(N^k)$ time.
This is essentially how P = NP is defined.

[✓ True] / ○ False / ○ P=NP?: There exists an algorithm to check a supposed solution to `LongestPath` in $O(N^k)$ time. Yep, there's a linear time algorithm, check all the proposed edges.

○ True / [✓ False] / ○ P=NP?: There exists an algorithm to calculate the length in bytes of the shortest Java program that solves `LongestPath`. This ended up being a bit more of a leap even from the bonus topics than I'd meant. This ends up being of equivalent difficulty to the "halting problem", which we have not discussed. Free 0.5 points!

… and that's it!

Nothing written on this page will be graded.

Data Structures Reference:

`HashSet<Key> {`	`MinPQ<Item extends Comparable<Item>> {`	
`  void add(Key k)`	`   MinPQ(Comparator<Item> c)`	
`  boolean contains(Key k)`	`   void insert(Item x)`	
`}`	`   Item min()`	
	`   Item delMin()`	
`HashSet` is the same except `Key` must	`}`	
implement `Comparable<Key>`	Uses natural order unless comparator given during construction.	
`HashMap<Key, Value> {`	`Stack<Item> {`	`Queue<Item> {`
`  void put(Key k, Value v)`	`  void push(Item x)`	`  void enqueue(Item x)`
`  boolean containsKey(Key k)`	`  Item pop()`	`  Item dequeue()`
`  Value get(k)`	`}`	`}`
`}`		
`TreeSet` is the same except `Key` must		
implement `Comparable<Key>`		
Assume all of these classes implement `Iterable` and have a `size()` method.		