0. Community Standard acknowledgment (signature):

1. Just sort() it!
   Give two reasons why you would use Comparator instead of Comparable?
   Multiple sort orders   Works in all classes
   Allows you to implement without modifying the class (when you are using “someone else’s class”)
   Easier to write compare for beginners because you don’t have to worry about “this”

2. Searching!
   a. To find connectivity, is it better to use Depth First Search or Breadth First Search? Accept both answers as I didn’t go into details – BFS works but isn’t as efficient.
   b. To traverse nodes in Depth First Search, which data structure do you have to use (implicitly or explicitly) if you can’t use recursion? (Hint: The answer is Stack or Queue) Stack
   c. To traverse nodes in Breadth First Search, which data structure do you have to use in order to maintain the order the vertices visited? (Hint: The answer is Stack or Queue) Queue

3. Binary Search: Consider the following binary search tree. Show the result of adding 25 and then deleting 72 without using tombstones:

   (2 points for each part – 1 point for decent attempt on each part)
4. (10 points) Prioritize This!
   a. (3) Draw the array representation of a priority queue below as a binary heap with the minimum value (3) at the top (root).

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>3</td>
<td>9</td>
<td>4</td>
<td>26</td>
<td>11</td>
<td>18</td>
<td>20</td>
<td>35</td>
<td>12</td>
<td>15</td>
<td>30</td>
<td>19</td>
<td>26</td>
<td>24</td>
<td>71</td>
<td>80</td>
<td>52</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

   (3 points for correct answer, 2 points for heap shape, 1 point for tree structure)

   ![Binary Heap Diagram]

   b. (3) For your answer of part a, show the result of removing the min key from the priority queue:

   (3 points for correct answer, 2 points for some sinking, 1 point for tree shape)

   ![Result Diagram]

   c. (4) For your answer in part a, show the result of adding 100 and then 1:

   (4 points for correct answer, 3 points for some swimming, 2 points for heap shape, point for tree shape)

   ![Result Diagram]
5. (13 points) 2-3 Search Trees

a. (1) What is the maximum height a 2-3 with n nodes in terms of big-Oh? \( O(\log n) \)

b. (3) Consider the 2-3 tree below. Show the result of adding Q, then T, and then D. Draw a separate tree for each addition for a total of 3 trees that you draw. To get full credit, you must follow the correct insertion process for each step and show the final result.

1 point each for each step
1. \( Q \leftrightarrow P \Rightarrow PQ \)
2. \( T \leftrightarrow SX \Rightarrow STX \) then \( R \Rightarrow RT \) with \( P S X \) as kids
3. \( D \leftrightarrow AC \Rightarrow ACD \) then \( CEJ \) with \( A C H L \) as kids
   Then \( M \Rightarrow EM \) with \( C J \) and \( RT \) as kids
   & C has A D has kids & J has H L as kids

c. (3) Assume that it takes \( O(\log m) \) time to add the \( m^{th} \) key from an unsorted list to a 2-3 Tree. How much time would it take to add \( n \) keys to a 2-3 Tree starting from an empty tree?

3 points for correct answer.
2 points for having correct answer for summation but not being able to reduce it to \( O(n \log n) \) (such as the summation or \( O(\log n!) \))
1 point for an honest attempt

\[
\int_1^n \ln(x) \, dx = n \ln(n) - n + 1
\]

\( \log 1 + \log 2 + ... + \log n = O(n \log n) \)

3. (1) How much time would it take to search for a particular key among \( n \) keys stored in a 2-3 Tree with \( n \) keys? \( O(\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_) \)

e. (1) How much time would it take to search for a particular key in an unsorted list of \( n \) keys? \( O(\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_) \)

f. (1) If you were searching for just one key in an unsorted list of \( n \) keys, is it faster for you to store it in a 2-3 tree and then search for it? No (because building the tree would cost more than you’ll save)

g. (3) You need to be searching for at least \( O(\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_) \) keys in an unsorted list in before you can have a faster algorithm by storing the keys in a 2-3 Tree and then searching for them.

3 points if their answer is correct for the above parts.
2 points if for wrong answer but they compare trade-offs of building the tree.
1 point if they know that they have to compare trade-offs.
6. **(15 points) Gridster Analysis**: Consider the 2-player game of 4x4 Gridster below.

a. (3 points) Mark and count all possible unique non-isomorphic moves in the Grid on the left. (Hint: One such move is already marked by X below). The game ends when all the squares are marked (Everyone who plays wins! 😊).

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>2</th>
<th>2</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>2</td>
<td>2</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

*2 points for getting close  1 point for doing any counting*

b. (3 points) Mark and count all possible unique non-isomorphic responses does the second player have if the first move is indicated by X below?

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

*2 points for getting close  1 point for doing any counting*

c. (3 points) Calculate the total number of possible board positions for 2-player game of 4x4 Gridster. Note: For this part, you do not have to account for isomorphic moves. For example, the positions below of 2x2 Gridster count as 4 different positions:

<table>
<thead>
<tr>
<th>X</th>
<th>X</th>
<th>X</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>Y</td>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>Y</td>
<td>X</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Y</td>
<td>Y</td>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

\[
\frac{16!}{(8!)^2}
\]

*3 points for \(16\binom{4}{2}\) or similar expression  2 points for getting close  1 point for doing any counting*

d. (3 points) Calculate the total number of possible board non-isomorphic positions for 2-player game of 4x4 Gridster. For example, the 4 positions below count as 1 isomorphic position.

<table>
<thead>
<tr>
<th>X</th>
<th>X</th>
<th>X</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>Y</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Y</td>
<td>X</td>
<td>Y</td>
<td>X</td>
</tr>
<tr>
<td>Y</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

This was tougher than intended. Any decent attempt & showing some reasoning gets 2 points. 1 point for anything (even stupid stuff)

e. (3 points) Calculate the total number of possible board positions for 2-player game of \(n \times n\) Gridster assuming \(n\) is an even number. Note: As in part c above, you do not have to account for isomorphic moves.

\[
\frac{n!}{((n/2)!)^2}
\]

*3 points for \(n\binom{n/2}{n/2}\) or similar expression  2 points for getting close  1 point for any explanation*
7. (7 points) **Doofenshmirtz Evil Incorporated**: Dr. Heinz Doofenshmirtz introduced a new sortinator algorithm that divides the input list into four parts and the recursively sorts each one and then merges all four. The pseudocode looks like this:

```plaintext
sortinator(list) {
    first-quarter = sortinator(first-quarter)  // 1st quarter of the list
    second-quarter = sortinator(second-quarter)  // 2nd quarter of the list
    third-quarter = sortinator(third-quarter)  // 3rd quarter of the list
    fourth-quarter = sortinator(fourth-quarter)  // 4th quarter of the list

    first-half = merge(first-quarter, second-quarter)
    second-half = merge(third-quarter, fourth-quarter)
    return merge(first-half, second-half)
}
```

a. (2 points) Assuming the merge method takes O(n) time where n is the sum of the length of the two parameters, write down the recurrence relation for the sortinator algorithm below:

\[ T(n) = 4T(n/4) + O(n) \]

2 pts for anything that has the format \( 4T(n/4) + O(n) \)

1 pt for anything close

b. (5 points) Using what you know about MergeSort analysis, solve this recurrence relation.

\[ T(n) = 4T(n/4) + cn = 4^2 T(n/4^2) + 2cn \] (by plugging \( n = n/4 \) in \( T(n) = 4T(n/4) + cn \))

\[ = 4^3 T(n/4^3) + 3cn = \ldots = 4^k T(n/4^k) + kc^n \]

Now plug \( k = \log_4 n \) to get \( n T(1) + cn \log_4 n = O(n \log n) \)

**Full points if they have the wrong answer for but show that they can reduce a recurrence relation with exponents and logs.**

4 points if they make an error but get the general idea

2 points their recurrence relation for a something simple like \( T(n) = T(n-1) + 1 \) but they have the correct answer for that

1 point for anything close
8. (10 points) Linker

Write a method called `merge` that merges two linked lists (`list1` & `list2`) so that the return value is the reverse of `list2` followed by the reverse of `list1` (without modifying the original list). You may assume that you already have a method `reverse(list)` that returns the reverse of `list` (without modifying the original list).

```java
public class Linker {
    public class Node {
        public int data;
        public Node next;
        public Node(int d) {
            data = d;
            next = null;
        }
    }

    public Node reverse(Node list) {
        // already implemented... assume this is available to you
        // returns the reversed list (without modifying the original)
    }

    public Node merge(Node n1, Node n2) {
        // TODO: Complete this so it returns a link to Node as described above
        Node rev1 = reverse(n1); // 2 pt
        Node rev2 = reverse(n2); // 2 pt
        // 4 pts (give 2 pts if they n2.next = rev1, wrong because reverse copies)
        Node current = rev2;
        while (current.next != null) {
            current = current.next;
        }
        current.next = rev1;
        return rev2; // 2 pts
    }
}
```

9. (10 points) APT Clone 2: Trie Hard

Write a method `updateSubtreeWeight(Node node)` that updates `mySubtreeMaxWeight`. The method takes `node` as a parameter, and updates `mySubtreeMaxWeight` in the nodes of the subtree rooted at `node` so that `mySubtreeMaxWeight` is equal to the maximum `myWeight` found in the subtree.

The class `Node` and `TrieAutoComplete` are identical to the ones you have seen in Autocomplete assignment but the method `updateSubtreeWeight(Node node)` is slightly different from what you saw in Autocomplete. You may assume all the children have the correct `myWeight`, but not necessarily the correct `mySubtreeMaxWeight`. (Hint: You can't just copy the code from the Autocomplete assignment because this is a slightly different problem).
public class Node implements Comparable<Node> {
    String myInfo;
    boolean isWord;
    String myWord;
    double myWeight = -1;
    double mySubtreeMaxWeight;
    Map<Character, Node> children;
    Node parent;

    public Node(char character, Node parentNode, double subtreeMaximumWeight) {
        myInfo = "" + character;
        isWord = false;
        children = new HashMap<Character, Node>();
        parent = parentNode;
        mySubtreeMaxWeight = subtreeMaximumWeight;
    }

    public void setWord(String word) { myWord = word; }

    public String getWord() { return myWord; }

    public void setWeight(double w) { myWeight = w; }

    public double getWeight() { return myWeight; }

    Node getChild(char ch) { return children.get(ch); }
}

public class TrieAutocomplete implements Autocompletor {
    protected Node myRoot; // Root of entire trie

    // Given a trie that is already built, update the mySubtreeMaxWeight
    public void updateSubtreeWeight(Node node) {
        // TODO: Complete (on the next page) as described in the problem statement
        // We can't trust current SMW of the node (what if it is larger than the
        // true value?). So we must initialize it to this node's current weight
        node.mySubtreeMaxWeight = node.getWeight();

        // This is optional. It is here just to demonstrate that this is the
        // base case. If node is a leaf, the for loop below won't execute anyway
        if (node.children.isEmpty()) return;

        node.children.forEach((k, v) -> {
            updateSubtreeWeight(v);
            // Must update children's SMW before update parent's
            // -2 points for swapping the order
            node.mySubtreeMaxWeight = Math.max(v.mySubtreeMaxWeight,
                node.mySubtreeMaxWeight);
        });
    }
}