

Problem Set 6

Due: 6pm Friday, April 9

Exam 2 Notice:

In class on Friday, April 9, the second take-home exam will be handed out. It will be due at midnight on Friday April 16. **This is a hard deadline. No extensions will be given after this time.** The exam will cover the material in lecture through Lecture 16 (Fri. April 2) and the material in problem sets through PS6, mainly focusing on material introduced since the last exam: binary trees and contracts and implementations of stacks, queues, priority queues, sets, bags, and tables. Because you should focus on the exam, it is **strongly** recommended that you submit PS6 on time (6pm Friday April 9), although you may use lateness coupons to turn in PS6 late. But since the exam is worth much more than the problem set, it might be best to cut your losses on PS6 if you are not done with it so that you can focus on the exam.

Overview:

The purpose of this assignment is to give you practice with manipulating trees and implementing collections via trees.

Working Together:

If you worked with a partner on a previous problem set and want to work with a partner on this assignment, you are encourage to choose a different partner. However, you may also work with someone you worked with in the first half of the semester.

Submission:

Each team should turn in a single hardcopy submission packet for all problems by slipping it under Lyn's office door by 6pm on the due date. The packet should include:

1. a team header sheet (see the end of this assignment for the header sheet) indicating the time that you (and your partner, if you are working with one) spent on the parts of the assignment;
2. For Problem 1, submit your final version of `ObjectTreeParser.java`.
3. For Problem 2, submit your final version of `BagMBSTEntries.java` and a testing transcript showing that your class works as expected

Keep in mind that your final project proposal is also due on Friday, April 9 (see Handout #20). This should be submitted separately.

Each team should also submit a single softcopy (consisting of your final `ps6` directory) to the drop directory `~cs230/drop/ps6/username`, where `username` is the username of one of the team members (indicate which drop folder you used on your hardcopy header sheet). To do this, execute the following commands in Linux in the account of the team member being used to store the code.

```
cd /students/username/cs230
cp -R ps6 ~cs230/drop/ps6/username/
```

Problem 1 [30]: Parsing Strings Into Trees

Background

It is easy to convert a binary tree into a string representation via the following `toString` method:

```
public String toString (ObjectTree t) {
    if (OT.isLeaf(t)) {
        return "*";
    } else {
        return "(" + OT.left(t) + " " + OT.value(t) + " " + OT.right(t) + ")";
    }
}
```

Indeed, such a `toString` method is included in the contracts for `ObjectTree` (Appendix A) and `ObjectMTree` (Appendix B).

In this problem you will define a `fromString` method with the following method header that inverts this process:

```
public static ObjectTree fromString (String s);
```

`fromString` converts a string representation `s` of an object tree into an `ObjectTree` instance `t` where every node has a string value and where `t.toString()` is equal to `s` (modulo possible differences in whitespace). This process is called "parsing" the string into a tree. If `s` is a well-formed tree representation for an object tree, then `fromString` returns an appropriate instance of `ObjectTree`. However, if the string parameter to `fromString` is ill-formed, then `fromString` should throw an exception indicating this fact.

For example, here are the results of `fromString` on some sample input strings. First are some examples of well-formed trees:

```
fromString("*") =
*

fromString("( * A * )") =
( * A * )

fromString("(( * A * ) B ( * C * ))") =
(( * A * ) B ( * C * ))

fromString("((( * This * ) is (( * an * ) example * )) of ( * an ( * ObjectTree! * )))") =
((( * This * ) is (( * an * ) example * )) of ( * an ( * ObjectTree! * )))
```

Fig. ?? shows some examples involving strings that are not interpretable as trees. In these cases, the exceptions thrown by `fromString` have been caught by the testing program and an error message has been printed out.

A parsing process is usually decomposed into two separate passes. The first pass, known as **scanning** or **tokenizing**, breaks the input string up into so-called tokens that represent the primitive units being parsed. In the case of tree parsing, the tokens are an open parenthesis "(", a close parenthesis ")", and any contiguous sequence of non-whitespace characters, such as "*", "253", "foo", and "\$a-b_c!#?73".

For example, the string

```
(( * foo * ) 230 ( * # $ ? ! * ))
```

```

fromString("") =
ObjectTree.fromString: expected ( but got )

fromString("A") =
ObjectTree.fromString: expected ( but got A

fromString("(") =
ObjectTree.fromString: too few tokens!

fromString("* ") =
ObjectTree.fromString: too few tokens!

fromString("* A") =
ObjectTree.fromString: too few tokens!

fromString("* A *") =
ObjectTree.fromString: too few tokens!

fromString("* A * *") =
ObjectTree.fromString: expected ) but got *

fromString("A B C") =
ObjectTree.fromString: expected ( but got A

fromString("* A B") =
ObjectTree.fromString: expected ( but got B

fromString("(* A *) B (C D E)") =
ObjectTree.fromString: expected ( but got C

fromString("(* A *) B (* C D)") =
ObjectTree.fromString: expected ( but got D

fromString("(* A *) B (* C *)") =
ObjectTree.fromString: too few tokens!

fromString("(* A *) B (* C *) D") =
ObjectTree.fromString: expected ) but got D

fromString("(* A *) B (* C *) D") =
ObjectTree.fromString: superfluous token D

fromString("* A *) B (* C *)") =
ObjectTree.fromString: superfluous token B

```

Figure 1: Examples of tree parsing errors encountered by fromString.

consists of the following tokens:

```
"(", "(", "*", "foo", "*", ")", "230", "(", "*", "#$?!", "*", ")", ")"
```

Whitespace is ignore in the tokenizing process, so "(* A *)" has exactly the same tokens as

```
"(*
      A
   *)"
```

The second pass, known as **parsing** builds a tree from the token sequence according to rules for tree formation. In the case of `ObjectTree`, the rules are:

- a leaf is represented by "*"
- a node is represented by "(*left-subtree-tokens value-token right-subtree-tokens*)".

The first pass of splitting the string into tokens is tricky, so this process has already been encapsulated for you in the `StringTreeTokens` class. The class method invocation

```
new StringTreeTokens(string)
```

creates an enumeration whose elements are the tokens in **string** for the tree parsing process. If there are no more tokens, an instance of `StringTreeTokens` will respond to `nextElement()` by throwing a `RuntimeException`.

The second pass is captured by a `fromTokens` method that takes a stream of tokens (represented as an instance of `StringTreeTokens`) and consumes all the tokens that it needs to to form a tree. It leaves behind any tokens not consumed by this process. In particular, when parsing an object tree:

- If `fromTokens` encounters a "*" token, it should return a leaf.
- Otherwise, `fromTokens` “expects” that the tree should begin with a "(" token. If not, it throws an exception. If so, it should consume all the tokens up to and including the matching ")" token, and then create an appropriate node. Of course, `fromTokens` will need to be invoked recursively to parse the left and right hand subtrees of this node.

Here is the definition of `fromString` based on this approach:

```
public static ObjectTree fromString (String s) {
    // Parses a string into an integer tree.
    // Either an ObjectTree, or throws an exception.
    Enumeration tokens = new StringTreeTokens(s);
    ObjectTree t = fromTokens(tokens);
    if (tokens.hasMoreElements()) {
        throw new RuntimeException("ObjectTree.fromString: superfluous token "
            + (tokens.nextElement()));
    } else {
        return t;
    }
}
```

Your Task

Your problem is to flesh out the `fromTokens` method:

```
public static ObjectTree fromTokens (Enumeration tokens);  
Parses the given enumeration of tokens into an object tree. Either returns an ObjectTree,  
or throws an exception indicating the tokens were not parsable into an ObjectTree.
```

Notes

- The skeleton for `fromTokens` can be found in the file `ObjectTreeParser.java`, which you should flesh out for this problem.
- A contract for the `ObjectTree` class can be found in Appendix A.
- In `ObjectTreeParser.java`, `ObjectTree` operations are accessible via the `OT.` prefix.
- The following are helpful auxiliary methods with which you have been provided:

```
public static void check (String expected, String actual) {  
    if (! expected.equals(actual)) {  
        throw new RuntimeException("ObjectTree.fromString: expected " + expected  
                                + " but got " + actual);  
    }  
}  
  
public static String nextToken (Enumeration tokens) {  
    try {  
        return (String) tokens.nextElement();  
    } catch (RuntimeException e) { // no more tokens  
        throw new RuntimeException("ObjectTree.fromString: too few tokens!");  
    }  
}
```

If you use the above auxiliary methods, then you should not have to explicitly throw any exception in your definition of `fromString`.

- This is definitely a problem in which spending time carefully working out your strategy on paper before jumping into coding will save you lots of time! The amount of code you have to write is rather small – `fromTokens` can be expressed in under 15 lines. However, you need to think carefully. The most important part of the solution strategy is *wishful thinking* – believing that when `fromTokens` is called recursively, it will correctly return the appropriate subtree after consuming all the tokens for that subtree.
- You can test your code by invoking:

```
java ObjectTreeParser tree-strings.txt
```

This invokes `fromString` on each of the lines in the test file `tree-strings.txt`. Feel free to add additional test cases to `tree-strings.txt`.

Problem 2 [70]: Mutable BST of Bag Entries Implementation of Bags

A *bag* is a collection of unordered elements that may contain multiple occurrences of each element. In the CS230 collection hierarchy, the `Bag` interface describes mutable bags. You should study this interface (Appendix C) before proceeding with this problem.

In this problem, you will implement a class `BagMBSTBagEntries` that represents bags as a mutable binary search tree of entries that pair elements in the bag with their number of occurrences. Each entry should be an instance of the following `BagEntry` class:

```
public class BagEntry {

    public Object elt;
    public int num;

    public BagEntry (Object elt, int num) {
        this.elt = elt;
        this.num = num;
    }

    public String toString () {
        return "BagEntry[" + elt + "," + num + "];"
    }

}
```

To improve the running time of the `size()` and `count()` bag operations, the values to be returned by these methods should be cached in instance variables of the `BagMBSTEntries` class.

So instances of `BagMBSTEntries` should have the following instance variables:

- `comp`: a comparator for determining the order of elements.
- `entries`: a mutable binary search tree whose elements are instances of `BagEntry`.
- `size`: the number of element occurrences currently in the bag (includes duplicates).
- `count`: the number of *distinct* elements currently in the bag (does not include duplicates).

For example, Fig. ?? shows one possible representation of an instance of `BagMBSTEntries` that contains two As, three Bs and one C. (The shape of the tree is determined by the order in which the elements were inserted.)

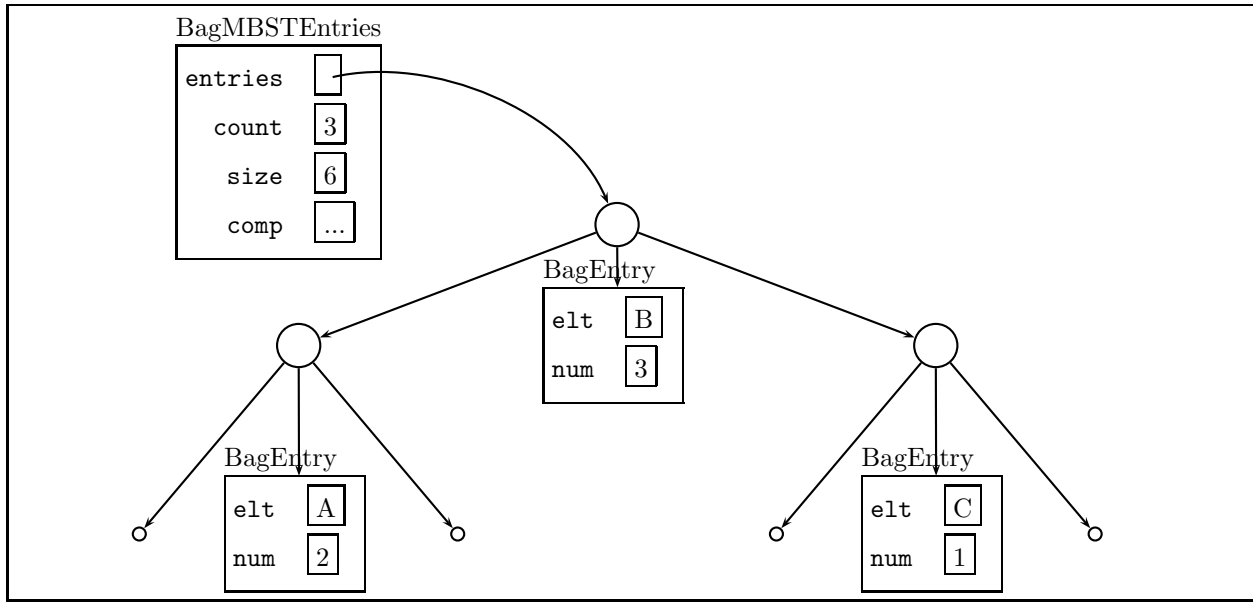


Figure 2: An example of a BagMBSTEntries instance.

To complete this problem, you need to flesh out the following methods of the bag implementation in `BagMBSTEntries.java` using the representation described above:

Constructor Methods

```

public BagMBSTEntries (Comparator c);
public BagMBSTEntries ();

```

Instance Methods

```

public Object choose();
public void clear();
public Object clone();
public int count();
public boolean delete(Object x);
public boolean deleteAll(Object x);
public Object deleteOne();
public boolean isMember(Object x);
public void insert(Object x);
public ObjectList toList();
public int occurrences(Object x);
public int size();

```

Note that many instance methods from the `Bag` interface in Appendix C are missing from the above list. This is because the `BagMBSTEntries` class inherits implementations of these other methods from its superclasses.

Test your implementation by executing `java BagMBSTEntries`, which invokes the bag methods on various simple `BagMBSTEntries` instances. Study the output carefully to make sure that the methods behave as expected. You should turn in the transcript of this test for your final version of the code as part of your hardcopy submission.

Notes:

- Mutable object trees are instances of the class `ObjectMTree`, whose contract is given in Appendix B. The `BagMBSTEntries` class has been configured so that the `ObjectMTree` operations are accessible via the `OMT.` prefix.
- `ObjectList` operations are accessible via the `OL.` prefix and `ObjectListOps` operations are accessible via the `OLO.` prefix.
- You may define any private auxiliary methods and additional classes that you find helpful for completing this problem.
- Many methods require searching through the `entries` binary search tree to find an existing `BagEntry` or the insertion point for a new `BagEntry`. Additionally, many of these methods not only need to keep track of the current tree node, but also need to keep track of its parent and whether the current node is the left or right child of the parent node. To avoid writing similar code many times, it's a good idea to write a single `findEntry` auxiliary method that abstracts over the process of finding an entry in `entries` and can be invoked many times. The result of `findEntry` is an instance of the `FindEntryInfo` class (Fig. ??), whose instance variables summarize the information needed by various other methods. (This class is provided for you in the `ps6` directory.) Here is a specification of the `findEntry` method:

```
private FindEntryInfo findEntry (Object x);
```

If a `BagEntry` for `x` exists in the `entries` tree, returns a `FindEntryInfo fei` where:

- `fei.entry` is the entry whose `elt` is `x`,
- `fei.child` is the tree node containing `fei.entry`.
- `fei.parent` is the parent of `fei.child` (or null if `fei.child` is the root of `entries`).
- `fei.isChildToLeft` is true if `fei.child` is to the left child of `fei.parent` (or there is no parent) and false otherwise.

If a `BagEntry` for `x` does not exist in the `entries` tree, returns a `FindEntryInfo fei` where:

- `fei.entry` is null.
- `fei.child` is the leaf where `x` would be inserted into the tree.
- `fei.parent` is the node below which `x` would be inserted into the tree.
- `fei.isChildToLeft` is true if `x` would be inserted to the left of `fei.parent` and false otherwise.

- Depending on how your binary search operations are defined, you might directly use `comp`, or you might need to “lift” `comp` via the `BagEntryComparator` class presented in Fig. ?. This class is provided for you in the `ps6` directory.

```

public class FindEntryInfo {

    public ObjectMTree parent, child;
    public boolean isChildToLeft;
    public BagEntry entry;

    public FindEntryInfo (ObjectMTree parent, ObjectMTree child, boolean isChildToLeft) {
        this.parent = parent;
        this.child = child;
        this.isChildToLeft = isChildToLeft;
        if (ObjectMTree.isLeaf(child)) {
            this.entry = null;
        } else {
            this.entry = (BagEntry) ObjectMTree.value(child);
        }
    }
}

```

Figure 3: The FindEntryInfo class.

```

import java.util.Comparator;

public class BagEntryComparator implements Comparator {

    private Comparator eltComp;

    public BagEntryComparator (Comparator eltComp) {
        this.eltComp = eltComp;
    }

    public int compare (Object x, Object y) {
        return eltComp.compare(((BagEntry) x).elt, ((BagEntry) y).elt);
    }

    public boolean equals (Object c) {
        if (c instanceof BagEntryComparator) {
            return eltComp.equals(((BagEntryComparator) c).eltComp);
        } else {
            return false;
        }
    }
}

```

Figure 4: The BagEntryComparator class.

Appendix A: ObjectTree Contract

The `ObjectMTree` class models immutable trees whose nodes hold object values.

Public Class Methods:

public static `ObjectTree leaf ()`;

Returns a leaf – i.e., a distinguished non-value-bearing node that denotes an empty tree.

public static `ObjectTree node (ObjectTree l, Object v, ObjectTree r)`;

Returns a tree node whose left subtree is `l`, whose value is `v`, and whose right subtree is `r`.

public static `boolean isLeaf (ObjectTree t)`;

Returns `true` if `t` is a leaf and `false` otherwise (i.e., if `t` is a tree node).

public static `Object value (ObjectTree t)`;

If `t` is a node, returns the value it holds. If `t` is a leaf, throws a `RuntimeException` indicating that a leaf has no value.

public static `ObjectTree left (ObjectTree t)`;

If `t` is a node, returns its left subtree. If `t` is a leaf, throws a `RuntimeException` indicating that a leaf has no left subtree.

public static `ObjectTree right (ObjectTree t)`;

If `t` is a node, returns its right subtree. If `t` is a leaf, throws a `RuntimeException` indicating that a leaf has no right subtree.

Public Instance Methods:

public `String toString ()`;

Returns a string representation of this tree. In this representation, a leaf is represented by `*` and a node `N` is represented by `(L V R)`, where `L` is the string representation of the left subtree of `N`, `V` is the string representation of the value of `N`, and `R` is the string representation of the right subtree of `N`. For example, the tree created via:

```
node(node(leaf(),"A",leaf()), "B", node(node(leaf(),"C",leaf()), "D", leaf()))
```

has the following string representation:

```
"(((* A *) B ((* C *) D *))"
```

Appendix B: ObjectMTree Contract

The `ObjectMTree` class models mutable trees whose nodes hold object values.

Public Class Methods:

```
public static ObjectMTree leaf ();
```

Returns a leaf – i.e., a distinguished non-value-bearing node that denotes an empty tree.

```
public static ObjectMTree node (ObjectMTree l, Object v, ObjectMTree r);
```

Returns a tree node whose left subtree is `l`, whose value is `v`, and whose right subtree is `r`.

```
public static boolean isLeaf (ObjectMTree t);
```

Returns `true` if `t` is a leaf and `false` otherwise (i.e., if `t` is a tree node).

```
public static Object value (ObjectMTree t);
```

If `t` is a node, returns the value it holds. If `t` is a leaf, throws a `RuntimeException` indicating that a leaf has no value.

```
public static ObjectMTree left (ObjectMTree t);
```

If `t` is a node, returns its left subtree. If `t` is a leaf, throws a `RuntimeException` indicating that a leaf has no left subtree.

```
public static ObjectMTree right (ObjectMTree t);
```

If `t` is a node, returns its right subtree. If `t` is a leaf, throws a `RuntimeException` indicating that a leaf has no right subtree.

```
public static void setValue (ObjectMTree t, Object newValue);
```

If `t` is a node, its value is changed to be `newValue`. If `t` is a leaf, throws a `RuntimeException` indicating that a leaf has no value.

```
public static void setLeft (ObjectMTree t, ObjectMTree newLeft);
```

If `t` is a node, its left subtree is changed to be `newLeft`. If `t` is a leaf, throws a `RuntimeException` indicating that a leaf has no left subtree.

```
public static void setRight (ObjectMTree t, ObjectMTree newRight);
```

If `t` is a node, its right subtree is changed to be `newRight`. If `t` is a leaf, throws a `RuntimeException` indicating that a leaf has no right subtree.

Public Instance Methods:

```
public String toString ();
```

Returns a string representation of this tree. In this representation, a leaf is represented by `*` and a node `N` is represented by `(L V R)`, where `L` is the string representation of the left subtree of `N`, `V` is the string representation of the value of `N`, and `R` is the string representation of the right subtree of `N`. For example, the tree created via:

```
node(node(leaf(),"A",leaf()), "B", node(node(leaf(),"C",leaf()), "D", leaf()))
```

has the following string representation:

```
"(( * A * ) B (( * C * ) D * ))"
```

Appendix C: Bag Interface

The **Bag** interface describes mutable collections of unordered elements that may contain multiple occurrences of each element. In mathematics, *multiset* is a synonym for *bag*. Each bag instance has a **Comparator** that is used to determine element equality (and may be used in bag implementations to order elements).

Public Instance Methods Inherited from Collection Interface:

public void insert (Object elt);

Modifies this bag by inserting a new occurrence of **elt**.

public Object deleteFirst ();

Deletes and returns an arbitrary element of this bag. Throws a **CollectionException** if this bag is empty. The **deleteFirst** method is a synonym for **deleteOne**.

public Object first ();

Returns an arbitrary element of this bag. Throws a **CollectionException** if this bag is empty. The **first** method is a synonym for **choose**.

public int size ();

Returns the number of element occurrences in this bag.

public boolean isEmpty ();

Returns **true** if this bag has no elements, and **false** otherwise.

public void clear ();

Removes all elements from this bag.

public Object clone ();

Returns a "shallow" copy of this bag – i.e. the copied bag structure is new, but elements themselves are shared with the old bag. Operations on the copied bag do *not* affect the original bag and vice versa. Operations on mutable elements of the copied bag *do* affect elements of the original bag, and vice versa.

public Enumeration elements ();

Returns an enumeration of the element occurrences in this bag in an arbitrary order.

public ObjectList toList ();

Returns a list of the element occurrences in this bag in an arbitrary order.

public String name ();

Returns a name indicating the implementation of this bag.

Public Instance Methods Inherited from ComparatorCollection Interface:

public Comparator comparator ();

Returns the comparator used by this bag to compare elements.

public boolean delete (Object elt);

Deletes one occurrence of **elt** from this bag. Returns **true** if **elt** was a member of the bag and **false** otherwise.

public boolean isMember (Object elt);
Returns **true** if **elt** is a member of this bag and **false** otherwise.

public void union (Collection c);
Modify this bag to contain the union of its elements with those of **c**. The union is the result of inserting each element of **c** into this bag. All comparison operations use the comparator of this bag.

public void intersection (Collection c);
Modify this bag to contain the intersection of its elements with those of **c**. The intersection is the result of keeping in this bag only those elements that are also in **c**. The number of occurrences of an element **e** in this bag after the intersection is the minimum of (1) the number of occurrences of **e** in this bag before the intersection and (2) the number of occurrences of **e** in **c**. All comparison operations use the comparator of this bag.

public void difference (Collection c);
Modify this bag to contain the difference of its elements with those of **c**. The difference is the result of deleting each element of **c** from this bag. All comparison operations will use the comparator of this bag.

Public Instance Methods Inherited from UnorderedCollection Interface:

public Object choose ();
Returns an arbitrary element of this bag. Throws a **CollectionException** if this bag is empty. The **choose** method is a synonym for **first**.

public boolean deleteOne ();
Deletes and returns an arbitrary element of this bag. Throws a **CollectionException** if this bag is empty. The **deleteOne** method is a synonym for **deleteFirst**.

Public Instance Methods Added by Bag Interface:

public int count ();
Returns the number of distinct elements in this bag (i.e., ignoring duplicates).

public boolean deleteAll (Object elt);
Deletes all occurrences of **elt** in this bag. Returns **true** if this bag contained at least one occurrence of **elt** before deletion, and **false** otherwise.

public int occurrences (Object elt);
Returns the number of occurrences of **elt** in this bag. Returns 0 if **elt** is not a member of this bag.

*Problem Set Header Page
Please make this the first page of your hardcopy submission.*

CS230 Problem Set 6

Due 6pm Friday April 9

Names of Team Members:

Date & Time Submitted:

Collaborators (*anyone you or your team collaborated with*):

By signing below, I/we attest that I/we have followed the collaboration policy as specified in the Course Information handout.

Signature(s):

*In the **Time** column, please estimate the time you or your team spent on the parts of this problem set. Team members should be working closely together, so it will be assumed that the time reported is the time for each team member. Please try to be as accurate as possible; this information will help me design future problem sets. I will fill out the **Score** column when grading your problem set.*

Part	Time	Score
General Reading		
Problem 1 [30]		
Problem 2 [70]		
Total		