CSC 385
Homework Assignment, Module 4
Finding Common Elements Between *k* Collections

Discussion

Introduction — A Real-World Problem

In the National Football League, when teams are seeded for the playoffs, there is always the possibility that two or more teams may have the same won-lost-tied record. There is an ordered, step-by-step, tiebreaker process by which such ties are broken. If the first step in the process fails to break the tie, the next step is applied, and so on, until the tie is broken. This assignment focuses on one particular step, which happens to be the third step in the process: the won-lost-tied percentages in common games between the two teams are compared. Two games are considered common between two teams if each of those teams faced the same opponent.

This problem can be abstracted into a more general problem:

> *Given two collections of elements, A and B, generate a third collection, C, containing only those elements common in collections A and B, with duplicates allowed.*

Note that any type of collection (list, array, etc.) can be used, and no guarantees are made regarding the order of the elements in either collection.

If we build a collection for each team in question that contains the name or identifier of the opponent that was faced in each game the team played, we end up with two separate collections of opponents. The goal is to generate a third collection that contains only the common opponents faced by both teams. For example, consider two teams from the AFC North Division from the 2011 season, the Pittsburgh Steelers and the Baltimore Ravens. Both teams finished the regular season with won-lost-tied records of 12-4-0. As it turns out, the Ravens won the tiebreaker not on the rule we're describing in this assignment, but on the rule that the Ravens won both head-to-head matchups during the season. But let's assume the tiebreaker had to be resolved by the won-lost-tied percentage in common games. Table 1 shows the entire regular season schedules for both teams, and indicates common opponents in bold. Although in this example, most of the games are common;i.e., both teams faced common opponents, this is not a requisite condition for the problem.

Game	Pittsburgh Steelers	Baltimore Ravens
1	Baltimore Ravens	Pittsburgh Steelers
2	**Seattle Seahawks**	**Tennessee Titans**
3	**Indianapolis Colts**	**St. Louis Rams**
4	**Houston Texans**	New York Jets
5	**Tennessee Titans**	*bye week*
6	**Jacksonville Jaguars**	**Houston Texans**
7	**Arizona Cardinals**	**Jacksonville Jaguars**
8	New England Patriots	**Arizona Cardinals**
9	Baltimore Ravens	Pittsburgh Steelers
10	**Cincinnati**	**Seattle**

	Bengals	**Seahawks**
11	*bye week*	**Cincinnati Bengals**
12	Kansas City Chiefs	**San Francisco 49ers**
13	**Cincinnati Bengals**	**Cleveland Browns**
14	**Cleveland Browns**	**Indianapolis Colts**
15	**San Francisco 49ers**	San Diego Chargers
16	**St. Louis Rams**	**Cleveland Browns**
17	**Cleveland Browns**	**Cincinnati Bengals**

Table 1. Regular season opponents of the Pittsburgh Steelers and Baltimore Ravens, listed in the order in which the games were played. Note that each team had a "bye" week, during which they did not play. Common opponents are indicated in bold.

The result returned by an algorithm solving this problem should return a third collection containing only those teams which both the Steelers and the Ravens faced during the season. The order in which these common teams are present in this collection is not important, and duplicates should be included. Table 2 shows one possible correct result.

Seattle Seahawks
Indianapolis Colts
Houston Texans
Tennessee Titans
Jacksonville Jaguars
Arizona Cardinals
Cincinnati Bengals
Cincinnati Bengals
Cleveland Browns
San Francisco 49ers
St. Louis Rams
Cleveland Browns

Table 2. One possible correct result showing all common elements (duplicates included) between the two collections in Table 1.

An Inefficient Solution

This problem can be solved in a number of ways. The most straightforward approach to this problem would be to set up a nested loop structure, where the collection #1 is traversed, and for each element in collection #1, collection #2 is traversed to see whether or not the element is present. Table 3 shows the regular season schedules for both the Steelers and the Ravens as generically labeled collections. Below is the algorithm, as pseudocode. Note that once a common element has been found and added to the collection of common elements, the inner loop is immediately terminated, to avoid adding the same element to the common collection multiple times.

```
for each element X in C1
{
```

```
    for each element Y in C2
    {
        if X.compareTo(Y) == 0
        {
            add X to common collection Z

            break
        }
    }
}
```

The complexity of this algorithm is quadratic. For 2 collections, both of length *N*, the worst case number of comparisons would be: N^2. The number of comparisons increases linearly with each additional collection. It should be apparent that for each additional collection, the worst case number of comparisons effectively adds 1 to the coefficient of the formula above, assuming all collections are of the same length. For example, for 3 collections, all of length *N*, the maximum number of comparisons would be $N * 2N$, or $2N^2$; for 4 collections, all of length *N*, the maximum number of comparisons would be $3N^2$; and so on. Note that the coefficient of the quadratic term is always equal to the number of collections minus 1. This is due to the fact there must be at least 2 collections in order to perform any comparisons. If there is only a single collection, then by default the collection of common elements is simply the collection itself. A more general formula, allowing for an arbitrary number of collections of differing lengths, is:

> ***Formula 1:*** *For k collections C_1, C_2, C_3, ..., C_k, of lengths N_1, N_2, N_3, ..., N_k, respectively, where one of the collections is chosen as the query collection, C_q, of length N_q, the worst case number of comparisons for finding all elements common to all k collections is proportional to: $N_q * (\Sigma[N_1, N_2, N_3, ..., N_k] - N_q)$.*

In Formula 1, the length, N_q, is the length of the collection chosen to be the "query" collection, which corresponds to the collection labeled `C1` in the pseudocode. In the formula, the size of the query collection is subtracted from the sum of the sizes of all the collections, since the elements in the query collection are not compared against themselves. If all the collections have the same length, the general formula becomes:

> ***Formula 2:*** *For k collections C_1, C_2, C_3, ..., C_k, all of which have length = N, the maximum number of comparisons needed for finding all elements common to all k collections is proportional to: $N * (kN - N)$, or $(k - 1)N^2$.*

As you can imagine, the complexity of this algorithm can rise dramatically depending on the value of *k*. When $k = N + 1$, for example, the algorithm becomes cubic, and continues to rise as the number of collections increases. We'll refer to it as quadratic, though, because of the N^2 component. One thing that should be apparent is that the size of the query collection imposes an implicit upper bound on the number of elements common to all the collections. That is, the number of common elements must be less than or equal to the size of the query collection. Although any one of the collections can be designated as the query collection, the number of comparisons will be minimized if the smallest collection is used as the query collection. This is stated more generally as:

> *For any k collections, where N_q represents the size of the smallest collection, the number of elements common to all collections will be less than or equal to N_q.*

C1	***C2***
Baltimore Ravens	Pittsburgh Steelers
Seattle Seahawks	Tennessee Titans
Indianapolis Colts	St. Louis Rams
Houston Texans	New York Jets
Tennessee Titans	Houston Texans
Jacksonville Jaguars	Jacksonville Jaguars
Arizona Cardinals	Arizona Cardinals

New England Patriots	Pittsburgh Steelers
Baltimore Ravens	Seattle Seahawks
Cincinnati Bengals	Cincinnati Bengals
Kansas City Chiefs	San Francisco 49ers
Cincinnati Bengals	Cleveland Browns
Cleveland Browns	Indianapolis Colts
San Francisco 49ers	San Diego Chargers
St. Louis Rams	Cleveland Browns
Cleveland Browns	Cincinnati Bengals

Table 3. Regular season opponents of the Steelers and Ravens, with the bye weeks eliminated, generically referred to as C1 and C2, respectively.

What You Need to Do:

If we were only interested in two relatively small collections, as shown in Table 3, the quadratic algorithm described above is adequate. However, if *N* is very large, and/or there are many collections (i.e., *k* is very large), and/or the complexity of the comparison operation is high (i.e., the data are more complex than integers), the quadratic algorithm may be too slow to be useful. There are a number of strategies that can be employed to reduce the complexity below quadratic, and at this point we have covered enough topics to design a more efficient algorithm. Based on the material covered thus far in this course, your goal is to design and implement a more efficient algorithm for finding the common elements of a set of collections. Ideally, the goal is to achieve an algorithm that will only need to perform at most on the order of *(k - 1)N* comparisons. This can only be achieved if each element in the non-query collections only participates in at most 1 comparison (with a few exceptions).

Your algorithm should satisfy the following criteria:

1. It should be able to accept as input 0 to *k* collections, stored as simple arrays. We're restricting the data structure to arrays since we haven't covered higher order data structures yet.

2. The elements of the collections should all be of type `Comparable`, and they should all be derived from the same base class (not counting the `Object` class). Implementation of the `Comparable` interface is necessary since the elements must be compared to each other in order to determine commonality. They must all be derived from the same base class since comparisons between different data types is undefined.

3. Duplicate elements should be allowed; e.g., if there are M instances of the value, "XYZ", in all the input collections, there should be M instances of the value, "XYZ", in the collection of common elements. For example, suppose you have the following collections:

banana	quince	plum
apple	raspberry	pomegranate
pear	banana	lime
banana	lemon	banana
pomegranate	apple	jujube
pineapple	banana	blueberry
cherry	cherry	apple

jujube	blueberry	cherry
cherry	jujube	grape
orange	mango	banana

The collection of common elements would be (order doesn't matter):

banana
apple
banana
cherry
jujube

4. The collections should be allowed to be of varying lengths; i.e., some collections may have more items than others.

5. One of the collections must be designated as the "query" collection, which is the collection containing the elements to which the elements in the other collections are compared.

6. The total number of element comparisons performed should be less than the value for the quadratic solution described above. That is, the total number of comparisons in the worst case should be less than $(k - 1)N^2$. Do not be concerned about average performance or best case performance. Also, the total number of comparisons is defined, for this assignment, to be only those comparisons that are performed once the traversal of the query collection begins, and the other collections are checked for the presence of the elements in the query collection. Any comparisons performed to
manipulate the data prior to searching for the common elements should be ignored.

The framework for your algorithm should satisfy the following criteria, for ease in testing:

1. Create a class called `CommonElements`, to contain your algorithm and associated methods and attributes.

2. In your `CommonElements` class, encapsulate your algorithm within a method called `findCommonElements`, that has the following signature:

   ```
   public Comparable[] findCommonElements(Object[] collections)
   ```

   The argument to this method, `collections`, will be the set of *k* collections discussed earlier. Each collection will be represented as an array of objects of type `Comparable`. Note that in Java, a 2D array will support arrays of varying sizes provided it is initialized without first specifying the two dimensions. For example:

   ```
   Comparable[][] collections = {{"A"}, {"A", "B"}, {"A", "B", "C"}};
   ```

   results in an array of 3 `Comparable` arrays of varying sizes. The following syntax also works:

   ```
   Comparable[] col_1 = {"A"};
   Comparable[] col_2 = {"A", "B"};
   Comparable[] col_3 = {"A", "B", "C"};
   Comparable[][] collections = {col_1, col_2, col_3};
   ```

3. The value returned by your `findCommonElements` method should be a collection of `Comparable` elements that contains only the elements common to all the input collections.

4. Since you are being asked to evaluate your algorithm based on the number of comparisons performed, you will need to have your `findCommonElements` method maintain a running total of comparisons performed for each set of collections tested. You should create an attribute called

`comparisons` in your `CommonElements` class to store the number of comparisons, and provide a getter method called `getComparisons()` to return this value. In order to keep a running total of comparisons, you will need to instrument your code by incrementing the `comparisons` attribute each time a comparison between two elements is made. Since element comparisons are typically performed in `if` statements, you may need to increment `comparisons` immediately before each comparison is actually performed. Although that may sound counter-intuitive, if you try to increment `comparisons` inside the `if` statement, after the element comparison has been made, you will miss all the comparisons that cause the condition inside the `if` statement to evaluate to `false`.

It is important that you adhere to the framework specification above. To facilitate testing of your program, I will use a test harness that will do the following:

1. Creates an instance of your `CommonElements` class.
2. Calls your `findCommonElements` method with a set of test collections as input.
3. Verifies that the collection that is returned by `findCommonElements` correctly contains the elements common to all the input collections.
4. Retrieves the number of comparisons that were performed, via your `getComparisons()` method.
5. Compares the number of comparisons performed to the target value stated in criterion #5 above for the algorithm.

Thus, it is essential that you name your class and methods as described above, or my test harness will not work, and it will take longer to test your program.

Hint for Achieving the Linear Solution

You can see that for the quadratic solution to this problem you will need a triple nested loop structure, where:

1. the outermost loop traverses the query collection
2. the middle loop traverses the collection of non-query collections
3. the innermost loop traverses the current non-query collection

Modifying the pseudocode from above, this might look like:

```
for each element Q in queryCollection
{
    for each collection C in non_queryCollections
    {
        for each element in C
        {
            if Q.compareTo(C) == 0
            {
                continue with next collection in non_queryCollections
            }
        }
        if Q not found in C
        {
            // Q cannot be common to all collections
            go to next element in queryCollection
        }
    }

    if Q found in all non_queryCollections
    {
        add Q to common collection Z
    }
}
```

If you trace through this code on a sample set of collections, you'll notice right away that each non-query element could be used in as many comparisons as there are query elements—in other words, there will be an enormous number of repeated comparisons. To achieve the linear solution you will need to find a way to eliminate the repeated comparisons. Ideally, each element should only have to be

involved in a single comparison (with few exceptions), but you need to have some way of keeping track of whether or not that comparison has occurred. Think about ways you can manipulate the collections, and possibly even use additional variables to trade space for faster performance.

A Note About Testing

You will need to develop several sets of test collections for testing your algorithm. The grading rubric mentions covering the case where all the test collections have the same length, as well as covering the case where the test collections are of different lengths. You will also need to think about what constitutes the worst case scenario for this algorithm, since only that scenario will make your analysis of total comparisons performed a meaningful one. You can use the formulas in the grading rubric to tell you how many comparisons you should expect in the quadratic and linear cases. For example, if you have 5 total collections (1 query collection and 4 test collections), each of which contains 10 elements, the total number of comparisons performed in the worst case should be: $(k - 1)N^2$, which for $k = 10$ and $N = 10$ is: $(5 - 1)10^2$, or 400 comparisons. For the linear algorithm, you should only have $N*(k - 1)$, which is $10*(5 - 1)$, or 40 comparisons.

Solution & Discussion:

Number of Collections:

The example used in the introduction to the problem only used a set of 2 collections, which makes the problem relatively trivial, since there is a single query collection and a single test collection. Scaling the algorithm up to k collections involves the need for setting up a loop to traverse the test collections.

Collection Size:

Allowing variable sized collections adds another wrinkle to the algorithm. The process of finding all the common elements is made easier if the smallest collection is used as the query collection. Finding out which collection is the smallest, however, adds a small amount of overhead to the algorithm proportional to k. If there are few collections, and the collections are small, it can be more efficient to simply pick any collection as the query collection rather than search for the smallest collection.

Order of the Elements in Each Collection:

The quadratic algorithm will work whether the elements in each collection are ordered or not. Sorting the elements in each collection allows for improvement with respect to finding the common elements, but at the expense of incurring the overhead cost of sorting all the collections. For k collections, all of length N, the overall average running time required to sort all k collections using quicksort is $O(kNlogN)$. If there are few collections, or if the collections are small, it may be more practical to leave the collections unsorted and use the quadratic algorithm. If the collections are large, or if there are many of them, the overhead of sorting the collections may be offset by the decreased running time required for finding the common elements, obtained by exploiting the ordering of the collections.

Using the Number of Element Comparisons as a Measure of Complexity:

The number of element comparisons performed during the finding of common elements will ultimately be the deciding factor in terms of running time. Although there will be other statements, such as loop variable increments and array assignments, that will be executed the same number of times as the number of element comparisons, the comparison process itself is somewhat of a wild card. As we've discussed before, comparing numerical data and even short String data, are relatively quick. Comparing complex objects, or bytewise comparison of files, introduces a large constant with respect to the running time of the comparisons that isn't applicable for other statements. For example, if the input collections contain paths to files of length X, and finding common elements entails comparing files byte by byte to determine equality, the Big-Oh term for the comparison statement in the quadratic algorithm will increase from $(k - 1)N(N + 1)/2$ to $X*(k - 1)N(N + 1)/2$. Both cases are still technically $O(N^2)$, but it should still be apparent that one case would be expected to take less time than the other.

What is the Worst Case Scenario?

The worst case scenario for this problem is a bit difficult to derive, since it depends on several factors. One of these factors is how duplicates are handled. Another factor is how the test collections are searched—sequentially or non-sequentially. This will be determined by the algorithm implementation. A

third factor is whether the loop that traverses each test collection breaks once a query element is found, or continues to the end of the test collection. Let's assume the following:

1. The quadratic algorithm is being used (see Figure 2 for the source code used as a reference).
2. The query collection, and all test collections are of size *N*.
3. The test collections are searched sequentially, from the first element to the last element.
4. The loop that traverses each test collection breaks once the current query element is found.
5. If there are multiple occurrences of a query element in a test collection, the search for that element always ends at the first occurrence of that element encountered by the search. For example, if the current query element is "ABC", and there are multiple occurrences of "ABC" in a test collection, a search for "ABC" in the test collection will end as soon as the first occurrence of "ABC" is encountered. In other words, we are assuming the search does not distinguish between multiple occurrences of a query element.

Given the above assumptions, the worst case would be a query collection in which all the elements are the same value, and a set of test collections in which each test collection contains exactly 1 occurrence of that value, located at the last position that is examined in each test collection. An example is shown in Figure 1. If we mentally run through the quadratic algorithm, we can see that for each occurrence of ABC in the query collection, each test collection must be traversed to the very last element, and since a match is found, ABC is added to the collection of common elements. Assuming the algorithm has not been implemented to "realize" that there is actually only a single, common value (ABC), it will blindly return a collection containing *N* occurrences of ABC, one for each occurrence in the query collection.

Previously in the course, we have discussed worst case search scenarios where the worst case is when the item being searched for is not found, since this case causes one extra iteration of the loop that controls the search. Here, that logic does not apply, since the common elements algorithm is not simply returning a boolean value indicating the presence or absence of an item. Instead, the algorithm does 3 things when an item is found, instead of only causing one more loop iteration:

1. Stores a reference of, or copy of, the item in the common elements collection.
2. Increments the index variable that references the position in the common elements collection where the next common element will be added.
3. Breaks the search loop.

Also, if the query element were not present in one of the test collections, the algorithm could be designed to immediately stop searching the test collections, and move on to the next query element, since an element must be present in all test collections in order to be a common element.

Query Collection	Test Collection #1	Test Collection #2	Test Collection #3	Test Collection #4
ABC	XYZ	XYZ	XYZ	XYZ
ABC	XYZ	XYZ	XYZ	XYZ
ABC	XYZ	XYZ	XYZ	XYZ
ABC	XYZ	XYZ	XYZ	XYZ
ABC	XYZ	XYZ	XYZ	XYZ
ABC	XYZ	XYZ	XYZ	XYZ
ABC	XYZ	XYZ	XYZ	XYZ
ABC	XYZ	XYZ	XYZ	XYZ
ABC	XYZ	XYZ	XYZ	XYZ
ABC	ABC	ABC	ABC	ABC

Figure 1. Example of a worst case scenario for the quadratic algorithm, where duplicate values are allowed, but no distinction is made as to how many occurrences of a given element are in any of the collections.

3 Solutions

A Quadratic Solution

One naive implementation of a quadratic solution is shown in Figure 2. It adheres to the basic pseudocode mentioned in the assignment, but performs two intial checks for special cases. The first is a check for the empty set; i.e., the set of input collections is empty. If there are no input collections, the algorithm immediately returns an empty common collection. Otherwise, the algorithm checks for the case where there is only one input collection. In this case, the input collection is the collection of common elements, so this collection is returned immediately. Both checks are negligible, O(1) operations. This implementation does not guarantee the correct result will be returned if any of the collections contain multiple instances of elements. Also, note that this implementation uses generics, hence the return type of `T`. As it is defined in the method signature, `T` can be any type that implements the `Comparable` interface, as well as any type that is derived from `T`. For example, if `String` data are used, the input data could be `String` or any type having `String` as its superclass. The actual input data must be homogeneous, however; types cannot be mixed.

```
public <T extends Comparable<? super T>> T[] findCommonElements(T[][]
collections)
{
    // number of element comparisons performed
    comparisons = 0;

    // number of instances of current query element found
    long numFound = 0;

    // check for special case: empty set
    if (collections.length < 1)
    {
        // no input collections; return an empty collection
        // declare it as type Comparable since can't instantiate an array
of type T
        Comparable[] result = new Comparable[0];

        return (T[])result;
    }

    // check for special case: only 1 collection
    if (collections.length == 1)
    {
        // no input collections; return an empty collection
        return collections[0];
    }

    // the query collection
    T[] queryCollection = collections[0];

    // array to store common elements; can't be more than
queryCollection.length common elements
    // declare it as type Comparable since can't instantiate an array of
type T
    Comparable[] commonElements = new Comparable[queryCollection.length];

    // index where next common element will be stored in commonElements
    int commonIndex = 0;

    // outer loop: traverse query collection
    for (int i = 0; i < queryCollection.length; i++)
    {
        // reset instances of current query element found
        numFound = 0;
```

```
        // traversal loop for all other collections
        for (int k = 1; k < collections.length; k++)
        {
            // traverse current collection sequentially
            for (int j = 0; j < collections[k].length; j++)
            {
                // another comparison performed
                comparisons++;

                // check for element in test collection
                if (queryCollection[i].compareTo(collections[k][j]) == 0)
                {
                    // one more instance found
                    numFound++;

                    // no need to traverse rest of current collection
                    break;
                }
            }
        }

        // determine if current query element common to all collections
        if (numFound == collections.length - 1)
        {
            // common element found; store it
            commonElements[commonIndex] = queryCollection[i];

            // set index for next common element to be stored
            commonIndex++;
        }
    }

    // return the array of common elements
    return (T[])commonElements;
}
```

Figure 2. One possible implementation of a quadratic algorithm for finding the common elements in *k* collections. This implementation does not correctly handle duplicate elements.

An *NlogN* Solution

If the collections are sorted, the inefficient inner loop of the quadratic algorithm can be replaced by a binary search. This is both good and bad. It is good in that the running time for the algorithm is drastically reduced. It is bad in that it will be difficult to determine whether or not there are multiple occurrences of a given query element common in all collections. Using the sample data from the introduction as an example, Collection C1 contains 2 occurrences of the item "Cincinnati Bengals". If Collection C2 is sorted so that a binary search can be used, the binary search will always halt at the same occurrence of "Cincinnati Bengals" both times. Further, if C2 contained only 1 occurrence of "Cincinnati Bengals", the binary search would find that 1 occurrence twice, thus incorrectly indicating that both collections have 2 occurrences of the element. Table 4 illustrates this problem. As can be seen from the table, the first element in collection C2 that a binary search will examine is the item at index 7: "Jacksonville Jaguars". The search will continue halving the collection, examining the item at `mid = (low + high) / 2`, until it reaches the occurrence of "Cincinnati Bengals" at index 1. This is the same occurrence of "Cincinnati Bengals" that will be found when the query item is the second occurrence of "Cincinnati Bengals" at index 4. The problem here is that the binary search proves there is one occurrence of "Cincinnati Bengals" in collection C2, but it can't determine if there are additional occurrences.

Resolving this problem requires two things:

1. keeping track of the number of occurrences of the current element in the query collection

2. a modification of the binary search algorithm to scan sequentially in either direction after finding an item to see if more occurrences of the item are present in the test collection.

While this can be done, the bookkeeping is somewhat tedious, and since we can actually do better than this (see below), the extra work is unnecessary.

Index	***C1***	***C2***
0	Arizona Cardinals	Arizona Cardinals
1	Baltimore Ravens	Cincinnati Bengals *(3)*
2	Baltimore Ravens	Cincinnati Bengals
3	Cincinnati Bengals	Cleveland Browns *(2)*
4	Cincinnati Bengals	Cleveland Browns
5	Cleveland Browns	Houston Texans
6	Cleveland Browns	Indianapolis Colts
7	Houston Texans	Jacksonville Jaguars *(1)*
8	Indianapolis Colts	New York Jets
9	Jacksonville Jaguars	Pittsburgh Steelers
10	Kansas City Chiefs	Pittsburgh Steelers
11	New England Patriots	San Francisco 49ers
12	San Francisco 49ers	San Diego Chargers
13	Seattle Seahawks	Seattle Seahawks
14	St. Louis Rams	St. Louis Rams
15	Tennessee Titans	Tennessee Titans

Table 4. Regular season opponents of the Steelers and Ravens, with the bye weeks eliminated, generically referred to as C1 and C2, respectively, and sorted in ascending order. In C2, numbers in parentheses indicate the order in which items will be examined during a binary search for "Cincinnati Bengals".

A Linear Solution

A linear solution is also obtainable if the collections are sorted. We will assume the elements are sorted in ascending order, but an equivalent algorithm can be designed for elements sorted in descending order. Rather than employing a binary search, this algorithm uses a collection of pointers, one for each collection. The pseudocode for this algorithm is shown in Figure 3. The basic idea is to maintain a set of pointers, one for each collection other than the query collection. For each element in the query collection, advance each of these pointers, one at a time, until an element is reached that is greater than or equal to the current element in the query collection, or until the pointer traverses past the end of the collection. If the element the pointer points to is greater than the current query element, or if the poiunter has gone past the end of the collection, we can immediately stop testing for the commonality of the current query element, since even one collection that does not contain the element precludes the element from being common in all *k* collections. This is illustrated by the inner `for` loop in the algorithm shown in Figure 3. If, when this loop ends, the loop variable happens to be equal to the number of collections, *k*, then we can deduce that the current query element must be present in all *k* collections, since the only way for the loop variable to be equal to *k* is if the inner `for` loop breaks naturally; i.e., it is not explicitly broken by the break statement. Thus, if the loop variable == *k*, we can add the current query element to the common collection and proceed to test the next element in the query collection.

Despite the fact that the algorithm uses a nested loop structure, it is still linear, since no element in any of the collections is ever examined more than once. The total number of comparisons performed will be on the order of (*k* - 1)*N, but can vary if the number of elements in each collection is not the same, and depending on how much each collection must be traversed before all the elements in the query collection have been tested. For example, if each collection contains 1000 elements, but all the common elements are found before half of each collection is tested, the number of comparisons will be considerably lower than the expected value of 1000*(*k* - 1). This algorithm also permits ideal handling of duplicate values. No special tricks need to be employed to ensure the correct number of occurrences of each common element is returned.

The algorithm's performance can be optimized in a couple of ways. Using the collection with the fewest number of elements as the query collection helps in a couple of ways. First, as discussed above, the maximum number of common elements can be no more than the number of elements in the smallest collection. This minimizes the number of element comparisons, and prevents repeated testing of collections where the pointers for those colections have gone beyond the end of the collection. Second, it provides an easy way of detecting the case where one or more of the collections is empty, in which case there can be no elements common to all collections. Swapping the smallest collection with the collection at collections[0] makes the inner loop easier to write. The query collection can be left where it is, but this requires that a check be made before any test collection is searched, in order to prevent querying the query collection against itself.

```
swap positions of collections[0] with collection with fewest elements

set commonsCollection = new empty collection

if (collections[0] is empty)
{
   return empty commonsCollection
}

sort all k collections

set queryPointer = 0

set pointerCollection = new int[k - 1]

for each queryElement in collections[0]
{
    for (k = 1; k < collections.length; k++)
    {
        move pointerCollection[k] until first element greater than or
equal to queryElement is reached
```

```
        if (pointerCollection[k] >= collections[k].length || element at
pointerCollection[k] != queryElement)
        {
            break;
        }
    }

    if (k == collections.length)
    {
        add queryElement to commonsCollection
    }
}

return commonsCollection
```

Figure 3. Pseudocode for a linear solution to the find common elements problem.

Rubric:

	Points	
Test Data		
The sets of test collections you used for testing your program.	3	
Functionality		
Correctly returns the common elements given *k* collections all of which have length = *N*.	6	
Correctly returns the common elements given *k* collections of varying length.	6	
Performance (The 3 possibilities below are mutually exclusive; you can only receive credit for 1 of them.)		
Total # of comparisons $\approx (k - 1)N^2$	5	This is the quadratic algorithm discussed in the assignment. Only worth minimal points. You are striving for better performance than this.
Total # of comparisons $\approx N*(k - 1)logN$	10	This is better than quadratic, but not as good as linear.

		Worth more, but not full credit.
Total # of comparisons *≈ (k - 1)N*	15	A linear algorithm is ideal. This is what you are shooting for.
Total:	**30**	

Examples of how the grading rubric works:

1. You submit your test data, and your algorithm correctly handles both cases listed under Functionality. Your algorithm uses approximately a quadratic number of comparisons. Your total score would be: 3 + 6 + 6 + 5 = 20.

2. You submit your test data, and your algorithm correctly handles both cases listed under Functionality. Your algorithm uses approximately a linear number of comparisons. Your total score would be: 3 + 6 + 6 + 15 = 30.

3. You submit your test data, and your algorithm correctly handles both cases listed under Functionality. Your algorithm uses a number of comparisons greater than linear, but less than quadratic. Your total score would be: 3 + 6 + 6 + 10 = 25.