CSC 385

Module 4 Assignment
Finding Common Elements Between k Collections

30 points

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.

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.

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 Bengals**	**Seattle 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.

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.

A Quadratic 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 one collection is chosen as the "query" collection. We will call this collection, Query_Collection. Query_Collection is traversed, and for each element in Query_Collection, we traverse all the remaining collections. Each of the remaining collections, represented by the variable, current_collection, is traversed to see whether or not the current element from Query_Collection is present. Below is this algorithm, represented as pseudocode.

```
for each element X in Query_Collection
{
      for each other collection
      {
            for each element Y in current_collection
            {
                  if X.compareTo(Y) == 0
                  {
                        X is common between Query_Collection and current_collection

                        break
                  }
            }
            if X not in current_collection
            {
                  X not common to all collections
            }
      }

      if X exists in all collections
      {
            add X to common_collection
      }
}
```

This algorithm bears some explanation. First, let us assume all the collections have an equal number of elements. The outermost loop, therefore, will iterate N times. The middle loop traverse the set of collections, not the elements of a collection. Since there are k collections, and the query collection is not compared to itself, the middle loop will iterate k minus 1 times. The innermost loop is similar to the outermost loop, and will iterate N times in the worst case. The total number of loop iterations is therefore given by N*(k - 1)*N, or the (k - 1) N^2. The starting point for each loop is independent of the current positions of the traversals of the other loops, so no counting formulas apply here.

The running time for this algorithm is thus quadratic. For 2 collections, both of length N, the worst case running time would be exactly N^2. The number of comparisons increases linearly with each additional collection. It should be apparent that each additional collection 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. Note that 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 collection of which 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 for finding all elements common to all k collections is proportional to: $N * (kN - N))$, or $O((k - 1)N^2$.*

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	bye week
Jacksonville Jaguars	Houston Texans
Arizona Cardinals	Jacksonville Jaguars
New England Patriots	Arizona Cardinals
Baltimore Ravens	Pittsburgh Steelers
Cincinnati Bengals	Seattle Seahawks
bye week	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 possibly 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.

5. Your algorithm should designate one of the collections as the "query" collection, which is the collection that will be compared against the other collections.

7. Your algorithm may manipulate the input collections as needed to accomplish its purpose.

8. 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(Comparable[][] collections).
   ```

   The argument to this method, collections, will be the set of k collections discussed earlier The type of the argument is an array of Comparable arrays. 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, the following statement:

   ```
   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. If your code doesn't meet the above specifications, I will notify you to change your implementation so that it meets the specifications I have described above.

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.

Some Hints

1. This algorithm can be implemented without using higher order data structures, so don't assume the solution lies in using something we haven't discussed in class.

2. In order to achieve the most efficient algorithm you will need to do some processing of the collections before you start trying to find the common elements. Think about what we've talked about thus far, and see if you can apply any of it to your solution.

3. Don't forget to handle special cases. It is valid to have an empty set of collections as input, as well as a set of collections that contains only a single collection.

4. You can make your algorithm more efficient if you abort a search as soon as you have determined that a given element is either common to all collections, or it is not.

5. Keep in mind that once you have determined whether an element is common to all collections or not, you should not need to use that element in any further comparisons, since it is impossible for the commonality of that element to change.

What you need to submit:

1. Your `CommonElements.java` file, implemented as specified above.

2. The set of test collections you used to test your algorithm.

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 ≈ $(k - 1)N^2$	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 ≈ $N*(k - 1)\log N$	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 runs in quadratic time. 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 runs in linear time. 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's running time is better than quadratic, but worse than linear. Your total score would be: 3 + 6 + 6 + 10 = 25.