Disclosure: As with any specification document in real life, this document may contain ambiguous descriptions, or not well explained parts, and even errors. When you read such a type of document, you need to clarify those parts wherever you find any type of problem that may hinder the process of designing a solution to the specified product (in this case, a computer program in Java). We have done our best to minimize such possible situations, but be sure that since nothing is perfect, we may need to modify this document in the coming days, either to correct errors found or to clarify parts of it. In this process, you as the developer will help a lot; please, read it as soon as possible and report any errors or inconsistencies that you may find.

Stay in touch with this online document in order to have the most recent version of these specifications at any moment.

1 Introduction

The short story that is written herein is fictional and is not really necessary to describe the project that you will be working on. The purpose of the story is to illustrate the importance that particular computing problems and their efficient solutions have to real life applications. The processes here presented are possible nowadays because of the pervasiveness of data collection devices that we constantly are in contact with, and because of the efficient computational strategies known to process and analyze large chunks of data. In recent years, new professional practices have risen, mainly as a combined application of Computer Science, Statistics and Mathematics disciplines. They are part of what is now usually referred to as Data Science. What we present here can be considered as a single instance, and we should emphasize, a very simple one, of the large number of activities carried out by a data scientist; whose main purpose is to effectively extract useful information from huge amounts of data.

And the story begins. There is an island where several recent crimes have occurred in a considerable number of its cities. All of the criminal incidents have a common operational mode: criminals forcing their access to private homes while residents are inside the house. Once gaining access to the premises, they steal money and other valuable property, as well as credit cards, debit cards, while forcing their victims to reveal the private information needed to use them (passwords, secret numbers, etc.). All incidents have been reported to last approximately 10 minutes. The police and other security agencies on that island are actively investigating those crimes, trying to end them by identifying and arresting their authors. So far, the only clues that they have is that all those crimes follow a similar pattern. Using that information and basic principles of criminal sciences, it is suspected that those crimes are being perpetrated by the same group of people, perhaps members of a criminal gang. For each of the criminal incidents, they have the following precise data:

1. The exact location where the crime occurred
2. A good approximation of the time when it happened; perhaps within ±5 minutes of the real time

Sergeant Espargata, who is the person leading the investigation, has decided to make use of modern technology to investigate those crimes and try to capture the perpetrators as fast as possible. She has contacted all the different telephone companies whose wireless services cover at least one of those areas and has requested, backed by a court order, for each of them to provide the list of all mobile devices (telephone numbers) that their systems have detected to be in the areas surrounding locations where the crimes occurred within a small time interval (not longer than 15 minutes) including the moment of the particular crime. Sergeant Espargata's approach is to explore all that data to determine all phone numbers that are common to all crime scenes. Her hypothesis is that individuals showing up on all of these locations at a time close to the moment of the crime should be treated as potential suspects or somehow related to those crimes[1].

[1] It would be better perhaps to find those appearing more than a particular number of times (a threshold value), but for the purpose of this project, we shall work only with those showing in all locations (if any).

An agreement has been made for the names of those companies to be kept confidential. Hence, they have identified n companies, and each is given a different pseudonim C_0, C_1, ... , C_{n-1}. The crime events are similarly identified as E_0, E_1, ... , E_{m-1}; where m is the number of crimes under this investigation. Each company is requested to provide m files, hence producing, among all of them, a total of *n*x*m* files. Files shall be named in a way that the company generating it, as well as the crime event that it corresponds to, can be extracted from that name. In particular, the file named *F_i_j.txt* (we shall use F_{ij} for short in the discussion that follows) contains all the telephone numbers that company C_i has generated for the crime event E_j, for $0 \le i < n$ and $0 \le j < m$. Since a particular company may not offer coverage on a particular crime location, some of the files may be empty; but it is known that for any target location, at least one of the companies has full coverage in the area. The files that are not empty contain all the phone numbers that their systems have registered as being in the surrounding area of a particular crime scene, in a radius of 1 mile, and at a time that falls within $\pm\frac{1}{2}$ hour from the time when the particular incident occurred.

Sgt. Espargata, who also has a degree in Mathematics, has correctly determined that the immediate problem at hand is a problem of finding unions and intersections of sets. Her plan is then to pursue the following approach.

1. For each crime event j, construct the set[2] $T_j = \bigcup_{0 \le i < n} F_{ij}$, for $j = 0, 1, \cdots, m-1$.
2. Construct the set[3] $T = \bigcap_{0 \le j < m} T_j$

Each set T_j is the set formed by all those numbers that are part of the combined data that those companies have provided with respect to crime event E_j. Set *T* will contain precisely those numbers that were detected in all crime events (the intersection of the T_j's).

Since we may be talking of possibly very large sets, she then decides to consult each of the four technical persons that the agency has hired to serve as technical consultants in computing matters, each holding a degree in computer science and engineering. For security reasons, their names are also kept confidential; hence, they are simply identified as P_1, P_2, P_3, and P_4. To solve the problem, each suggests one programmable solution that is different from the others, although perhaps using common ideas or components.

The solutions suggested by P_1 and P_2, are both based on the following scheme or general algorithm to construct the final set T:

```
T ← T_0
for j in 1, ..., m-1 do
    for (each x ∈ T) do
        if (x ∉ T_i) remove x from T
return T
```

As we will see, the differences between those two solutions reside on how the sets are represented and how each set operation involved is implemented.

The solutions by P_3 and P_4 are based on the following idea[4]:

[2] This is the union of n sets. Also, you should note that, under the assumptions earlier described, none of the sets T_j is empty. Why is so?

[3] This is the intersection of m sets.

[4] If A and B are sets, then A+B is the collection formed by all elements of A plus all the elements of B, repetitions allowed (multiset).

> Count the number of occurrences of each element in the multiset $T_0 + T_1 + \ldots + T_{m-1}$.
>
> Those elements whose count (frequency) is equal to *m* are the elements of *T*.

The differences between the two solutions by P_3 and P_4 is in the way in which the elements in the multiset are counted.

Sgt. Espargata is happy that she can count on several options to solve the problem. However, in addition to having those four solutions implemented, she also wants to choose the one that gives the best performance in terms of execution time: the one that executes faster. She does not want a theoretical result here, she wants empirical evidence establishing which implemented solution is the best option to use. Because of existing policies within the police department, the final solution must be written in Java. Unfortunately, none of those four consultants know that computer language, and given the urgency, there is no time for them to learn it now.

This is the moment when you become an important actor in this story. Your task is as simple as this: you will implement a Java application that includes the following:

1. An implementation of each of the four strategies to solve the problem. Those implementations will be part of the same Eclipse/IntelliJ project and they all share whatever code is possible to share without altering the essential features of each proposed solution.
2. An implementation of an experimentation framework which allows to empirically test each of the four solutions and get execution time data that will eventually be used to decide which of the four is the fastest solution. This will also be a part of the same Eclipse/IntelliJ project.

You need to meet the deadline that has been established to solve the case, which happens to coincide with the deadline for this project. We shall talk about that later in this document.

Your Preliminary Analysis

And hence you begin analyzing the four solutions being proposed. In your preliminary analysis, you discover that all the proposed solutions are based on the following specification of a set: the My`Set<E>` ADT. In all, it is assumed that the generic data type `E` must always be instantiated to an object data type (a class in Java) that has overwritten the `equals` method of the `Object` class.

```
public interface MySet<E> extends Iterable<E>, Cloneable {
    /** returns the current size of the set **/
    int size();

    /** returns true if the set is non-empty; otherwise, returns false. **/
    boolean isEmpty();

    /** To determine if a given object belongs to the set.
      @param e the object to test
      @return true if e is element in the set; false, otherwise.
    **/
    boolean contains(E e);

    /** To remove an element from the set. If the element to remove is not a
        member of the set, then this operation has not effect. Otherwise, the
        specified element is removed from the set.
      @param e the element to remove.
```

```
    **/
    void remove(E e);

    /** To add an element to the set. If the specified element is already
        an element in the set, then this operation has no effect. Otherwise,
        the element is added to the set.
      @param e the element to add.
    **/
    void add(E e);

    /** Returns an Iterator object through which all the elements of the
        set can be accessed, one by one, by properly invoking that iterator
        methods.
    **/
    Iterator<E> iterator();

    /**
     * To return a shallow copy of the current set.
     * @return a shallow copy of this set.
     * @throws CloneNotSupportedException
     */
    Object clone() throws CloneNotSupportedException;
}
```

You immediately discover that the difference between the solutions proposed by P_1 and P_2 is in the way in which the set is to be implemented. Based on that, you propose to implement them as classes named `Set1` and `Set2`, respectively. You also want each class to provide the feature of displaying the elements of a set in standard set format; and for that, you plan to add a `toString()` method. Your design has also taken into consideration the desire for code reusability as much as possible; hence your plan is to implement both as subclasses of the following abstract class:

```
public abstract class AbstractMySet<E> implements MySet<E> {
   public String toString() {
       String s = "";
       int count = 0;
       for (E e : this) {
              count++;
              if (count > 1)
                     s += ", " + e;
              else
                     s = "{" + e;
       }
       return s + "}";
   }
   public abstract Object clone()  throws CloneNotSupportedException;
}
```

You then remembered that you have already seen similar implementations of these two classes in lectures for the data structures course. Hence, the essential part of the solutions by P_1 and P_2 can be summarized as in the following discussion.

Solution by P_1

The solution given by P_1 is fundamentally based on an implementation of the `MySet` ADT using a `List` object to store the elements of the set instance at any moment. Elements in that internal list are stored in any order; that is, the order is not important. More specifically, that internal list is implemented using `ArrayList<E>`, and from the ideas shared by P_1, you have decided[5] to use class `Set1` that was studied in the course:

```
public class Set1<E> implements MySet<E> {
   private ArrayList<E> elements;
   public Set1() {
       elements = new ArrayList<>();
   }
   public int size() { return elements.size(); }
   public boolean isEmpty() { return elements.isEmpty(); }
   public boolean contains(E e) {
      int i = find(e);    // sequential search for e in list elements
      return i >= 0;
   }
   public void add(E e) {
      if (find(e) == -1)
         elements.add(e);
   }
   public void remove(E e) {
      int i = find(e);
      if (i == -1) return;
      int last = elements.size() - 1;
      if (i < last)
         elements.set(i, elements.get(last));
      elements.remove(last);
   }
   private int find(E e) {
      for (int i=0; i<elements.size(); i++)
          if (elements.get(i).equals(e))
             return i;     // e found at position i of list elements
      return -1;   // e is not found in the set
   }
   public Iterator<E> iterator() {
      return elements.iterator();
   }
}
```

Solution by P_2

[5] Of course, that was after you consulted me if that would represent a violation of copyrights, requiring you to have previous authorization for their use, to what my answer was "NO, to my best knowledge". This is because the two are based on common knowledge of what is already publicly available. But in general, that might be an issue to consider when you use solutions being proposed by others. We always keep in mind the code of ethics of the profession.

The alternative presented by P_2 to implement the `MySet` ADT is based on a Java class named: `HashMap<K, V>`. See a partial description of the `HashMap<K, V>` class at the end of this document. For more information, access the official web page for Java documentation.

In particular, you have discovered that his idea can be implemented by class `Set2` as was studied in the course.

```
public class Set2<E> implements MySet<E> {
   private HashMap<E, E> elements;
   public Set2() {
       elements = new HashMap<>();
   }
   public int size() { return elements.size(); }
   public boolean isEmpty() { return elements.isEmpty(); }
   public boolean contains(E e) {
      return elements.containsValue(e);
   }
   public void add(E e) {
      elements.put(e, e);
   }
   public void remove(E e) {
      elements.remove(e);
   }
   public Iterator<E> iterator() {
      return elements.keySet().iterator();
   }
}
```

In the solutions by P_3 and P_4, you discover that, wherever a set is needed, both use the same implementations as `Set2` that was also implemented by P_2. However, these two solutions differ in the way in which each count the frequency of each elements in the multiset $T_0 + T_1 + \ldots + T_{m-1}$. They both use a list (`ArrayList<Integer>`) named `allElements` = $T_0 + T_1 + \ldots + T_{m-1}$.

Solution by P_3

The solution given by P_3 is based on the following idea to fill a set T (the variable t in the code that follows), which is initially empty, with all those elements in list `allElements` which appear m times (whose frequency is m).

```
allElements.sort(null);
MySet2<Integer> t = new Set2<>();
Integer e = allElements.get(0);
Integer c = 1;
for (int i=1; i<=allElements.size(); i++) {
      if (i < allElements.size() && allElements.get(i).equals(e))
          c++;
      else {
          if (c == m)
             t.add(e);    // m is as in the previous discussion
```

```
            e = allElements.get(i);
            c = 1;
        }
}
```

Solution by P_4

The solution given by P_4 is based on using the `HashMap<Integer, Integer>` to count the frequency of the elements in `allElements`. The essential part of the algorithm is resembled in the following piece of Java code:

```
HashMap<Integer, Integer> map = new HashMap<>();
for (Integer e : allElements) {
     Integer c = map.getOrDefault(e, 0);
     map.put(e, c+1);
}
MySet2<Integer> t = new Set2<>();
for (Map.Entry<Integer, Integer> entry : map.entrySet())
     if (entry.getValue() == m)
        t.add(entry.getKey());
```

What you need to do to fulfill the assigned task? You have to provide an implementation of each one of those four strategies, which are identified by Sergeant Espargata as P1, P2, P3, and P4, as per the person who is proposing each. Once completed, you will submit those for personnel from the security department to test them with several dataset cases that they have developed, and which are independent of any test data of yourself.

2 What do you Need to do?

You need to submit two parts (software components) of this project as described next. Note: In the following discussion about those two parts, we make references to two classes that contain a main method, and which need to be part of your project. They are `Part1Main` and `Part2Main`. The Java files for both must be part of a package of the project that is called `p1MainClasses`.

Part 1: This part shall include a working version of each one of the four strategies. Each such working version should be able to process the input data by itself. Each must implement the following Java interface:

```
/**
 * An object of this type is capable of finding the intersection of a family
 * of sets of elements of a particular data type.
 *
 * @param <E> the data type of elements in the sets. It is assumed that it
 * is always instantiated to an object data type that has overridden its
 * own version of the equals method from the Object class.
 */
public interface IntersectionFinder<E> {
       /**
        * Intersects a family of sets.
        * @param t array containing the family of sets to be intersected.
        * @return the final intersection set (the result of intersecting all sets in t)
```

```
        */
       MySet<E> intersectSets(MySet<E>[] t);
       String getName();
}
```

by extending the following abstract class:

```
public abstract class AbstractIntersectionFinder<E>
implements IntersectionFinder<E> {
       private String name;   // to identify the strategy
       public AbstractIntersectionFinder(String name) {
              this.name = name;
       }
       public String getName() {
              return name;
       }
}
```

For this, you shall provide three different classes, each as a subclass of the above abstract class. One of those subclasses will implement the set intersection approach that is followed by P_1 and P_2, adaptable to use sets of type `Set1` or of type `Set2` as for the specification given. The second class is to implement P_3's approach, and the third one is to implement that of P_4. However, to better code reusability, you may add other intermediate subclasses. You should also provide another class that is able to integrate the execution of all those strategy classes or to execute specific ones. That class shall include a main method, and its name shall be `Part1Main`. Its main method is based on the following process:

1. Read all the input data; in particular,
   a. Read an initial file named `parameters.txt`, that contains the two integer values for n and m, in that order and one per line.
   b. Read all the `F_i_j` files expected as per the values for n and m. The program should end with an exception, carrying an appropriate message, in the case that at least one of those files is missing from the input directory.
2. From that input construct the objects corresponding to sets $T_0, T_1, \ldots, T_{m-1}$ as described.
3. Apply the selected strategy (or each one of the four strategies) to construct the final intersection set that results (or sets that result) from the particular input used; and then show results.

More details about this class are given next.

INPUT FOR PART 1: The input data shall be the received value for main method's parameters (to be illustrated in the section explaining the output), the content of the file `parameters.txt`, as well as the content of all the corresponding *F_i_j* files. All those input files will be located in a local directory (at the same directory from where the program is executed) to be named `inputFiles`. As per the discussion earlier, file `F_i_j.txt` will contain all the data (numbers) that company i has submitted related to location of crime event j (i and j are explicit numbers as already explained). Each file contains a sequence of numbers, one per line, representing the phone numbers that companies have provided for the processing. Some of those files may be empty; but, as suggested by the earlier discussion, there must be one file for each pair (i,j), where $0 \leq i < n$ and $0 \leq j < m$.

OUTPUT FOR PART 1: The output data shall be written to the console, as suggested by the following cases, which are the results for a particular input in which the intersection of all the sets is the set containing the four elements 2, 4, 5 and 13:

NOTE: IN WHAT FOLLOWS, WE ARE ASSUMING THAT THE CURRENT DIRECTORY THE COMMAND IS ISSUED FROM IS THE DIRECTORY THAT ECLIPSE HAS CREATED FOR THE PROJECT, WHERE THE `BIN` AND `SRC` FOLDERS ARE LOCATED. IF ISSUED FROM A DIFFERENT DIRECTORY, THEN WE NEED TO INCLUDE ALL WHAT IS NECESSARY FOR THE CLASSPATH TO BE PROPERLY SET TO THE DIRECTORY WHERE THE COMPILED PACKAGES OF THE PROJECT ARE.

Execution Command	Generated Output[6]
`java -classpath ./bin p1MainClasses/Part1Main 1`	`Final Set by P1: {4, 13, 2, 5}`
`java -classpath ./bin p1MainClasses/Part1Main 2`	`Final Set by P2: {2, 4, 5, 13}`
`java -classpath ./bin p1MainClasses/Part1Main 3`	`Final Set by P3: {2, 4, 5, 13}`
`java -classpath ./bin p1MainClasses/Part1Main 4`	`Final Set by P4: {2, 4, 5, 13}`

`java -classpath ./bin p1MainClasses/Part1Main`	`Final Set by P1: {4, 13, 2, 5}`
	`Final Set by P2: {2, 4, 5, 13}`
	`Final Set by P3: {2, 4, 5, 13}`
	`Final Set by P4: {2, 4, 5, 13}`

Part 2: This part requires you to write a software component to produce the empirical results for the execution times of each one of the four strategies. Your experiments shall be based on the following scheme[7]. The main class for this part shall be named `Part2Main.`

The goal is to generate empirical data that allows visualization of how the four strategies compare in execution times. The system should include testing the execution of the different strategies on randomly generated data of given sizes. For each strategy, the system will provide a list of pairs consisting of the following two items:

- size - the size of the data sets tested
- time - the average time that the particular strategy takes to fully process the datasets of that size.

For this, the proposed scheme is to generate, for any given size s, a fixed (we will use 200 as default) number of data sets. Each of those data sets is to be processed by the four strategies with the final goal of measuring the time (nanoseconds) that each takes to process the data set. When all those data sets of size s have been processed, the system computes, for each strategy, what the average execution time was, say $t_{avg}(s)$. Then, from the pair $(s, t_{avg}(s))$ corresponding to a particular strategy we interpret that such strategy has an estimated time $t_{avg}(s)$ for data sets of size s.

For that experimentation, we need to set six parameters. These, as well as their respective default values[8] to use, are n = 10 telephone companies and m = 50 crime events, initial size is 1000, final size is 50000, size incremental

[6] These are actual results as generated by my implementations.

[7] Although we are fixing parameter values here (m, n, the range of s, and number of repetitions per size), the program should allow for them to be easily modifiable and the whole program automatically adapted to the changed values. See the discussion about the input.

[8] Our discussion here is based on those default values, but the program should easily adjust to any other group of valid parameter values.

is 1000, and number of repetitions is 200. The total size s ($s = \sum_{i=1}^{n}\sum_{j=1}^{m} |T_{ij}|$) of all sets to process will vary, per experiment, from the initial size up to the final size, while incrementing size by the incremental value. For example, for the default values of parameters, the sizes to be tested are: $s = 1000$, $s = 2000$, $s = 3000$..., $s = 50000$. For each size, 200 datasets of the given size will be randomly generated, and each one of those data sets will be processed using the four strategies. The time taken on each case is recorded. When all the 200 data sets for a given size are processed by the four strategies their average value is computed for each strategy. That average value shall be used as the estimated time that the particular strategy takes on the given size. As mentioned before, your program should allow for easy change of those parameter values.

The following partial Java code that follows illustrates the idea. Here, we are assuming the `resulsPerStrategy` contains references to objects representing the strategies to be used. These are objects of type `StrategiesTimeCollection`. Each such strategy can be executed by applying method `runTrial(dataset)`, where `dataset` is the data to be used on that execution. Each data set is assumed to have the content of what would be the simulated data for each one of the files F_i_j. For example, I have used a 3-dimensional array in which position [i][j] is an array containing the generated data for F_i_j. An object of type `StrategiesTimeCollection` is an object, which, among other instance variables, at the end it will also contain a list of all pairs (*size, estimated time for that size*). It also contains auxiliary instance values to eventually compute the average of the times determined on each trial for that particular strategy and size. This will support the final computation of the average time after all the trials for the particular size are completed.

```
for (int size=initialSize; size<=finalSize; size+=incrementalSizeStep) {
    // For each strategy, reset the corresponding internal variable that will be used to
    // store the sum of times that the particular strategy exhibits the current size.
    for (StrategiesTimeCollection<Integer> strategy : resultsPerStrategy)
        strategy.resetSum();  // set to 0 instance variable to accumulate sum of times

    // Run all trials for the current size.
    for (int r = 0; r<repetitionsPerSize; r++) {
      // Common dataset to be used in the current trial by all the strategies being
      // tested. Data set is generated by a method that gets as input (parameter values):
      // n, m, size. Where n and m are the number of companies and number of crime events,
      // respective. The generated data must satisfy: size = Σ_{i=1}^{n} Σ_{j=1}^{m} |dataset[i][j].length|
        // data set for this trial of given size
        Integer[][][] dataset = generateData(n, m, size);

        // Apply each of the strategies using previous dataset as input;
        // and, for each, estimate the time that the execution takes.
        for (StrategiesTimeCollection<Integer> strategy : resultsPerStrategy) {
            long startTime = System.nanoTime();  // Measure system's clock time before.
            strategy.runTrial(dataset);          // Run strategy using data in dataset.
            long endTime = System.nanoTime();    // Measure system's clock time after.

            int estimatedTime = (int) (endTime-startTime);   // The estimated time.
            // Accumulate the estimated time to sum of times that the current strategy has
            // so far accumulated on the previous trials for datasets of the current size.
            strategy.incSum(estimatedTime);
        }
    }
```

```
    // For each strategy, compute the average time for the current size.
    for (StrategiesTimeCollection<Integer> strategy : resultsPerStrategy)
        strategy.add( new AbstractMap.SimpleEntry<Integer, Float>
                      (size, (strategy.getSum()/((float) repetitionsPerSize)))
                    );
}
```

For each trial, its execution must include the process to generate the sets T_0, T_1, … , T_{m-1} from the input data, followed by the intersection of those m sets. Those m sets are built directly from the elements in the arrays `dataset[i][j]`. Remember that here, the array in `dataset[i][j]` simulates content of file F_i_j. In particular, the following scheme can be used to construct those sets.

```
for (int j=0; j<m; j++) {
    t[j] = empty set of the particular type strategy uses per description given...
    for (int i=0; i<n; i++) {
        for (int k = 0; k < dataset[i][j].length; k++)
            t[j].add(dataset[i][j][k]);  // add to set t[j] the element dataset[i][j][k]
    }
}
```

Again, notice that what the above is doing is computing $T_j = \bigcup_{0 \le i < n} dataset[i][j]$, $j = 0, 1, \cdots, m-1$, where `dataset[i][j]` is the generated content for what would be file F_{ij}.

INPUT FOR PART 2: The input is to be received by your program as values to the parameters of the main method; if no such input is provided, they will be initialized to the default values previously established. So, when executing the program, we should be able to do it by typing the following command on the console. The command is as follows:

```
java -classpath ./bin  p1MainClasses/Part2Main  n  m  isize  fsize  istep  rep
```

where the six values following the class name are, in the given order:

n - the number of companies
m - the number of crime events
isize - the initial size for experimentations
fsize - final size for experimentations
istep - increment of sizes
rep - number of repetitions for a each size

You should notice that the order of the parameters need to be observed. However, if none of the parameters are included in the javac command, then they are all initialized to the default values given. In general, if the last i values are omitted, then the parameters corresponding to those i positions will be respectively initialized to their default values.

OUTPUT FOR PART 2: From that input data, your program will generate a file containing the results of the experimentations. That file is to be named `allResults.txt`, and it will be located inside a local directory called `experimentalResults`. This directory must exist inside the directory from where the program is being

executed. The format of file `allResults.txt` is as illustrated next. The following is a display produced by command "more" for one such file[9].

```
Size    P1      P2      P3      P4
1000    115692.42       110655.45       190051.38       148219.77
1100    79439.08        66471.37        97530.26        76653.66
1200    61045.586       40878.035       86872.15        64969.496
1300    60649.805       41483.46        74643.94        74065.69
1400    61781.88        44946.67        72130.63        39031.44
...
```

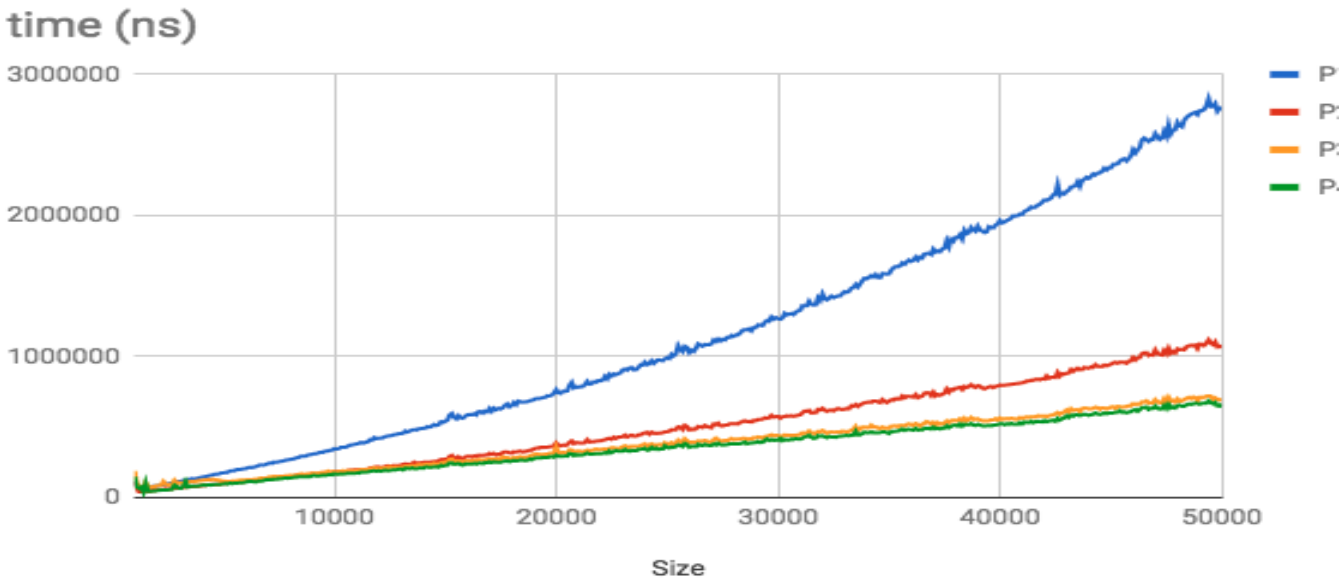
The first line contains the string "Size P1 P2 P3 P4", each item separated by tab character ('\t'). The content of each remaining line in the file shall be of the form "n t1 t2 t3 t4", where each of the last four values is the time that your program has estimated for each particular strategy, respectively, for the tests conducted for size n. Values are also separated by '\t' character.

And what do you do with the output file? You will upload it to your GDrive at upr.edu as a Google Sheet. For the above example, that Google sheet would contain the following in its first rows. Yours will be different.

Size	P1	P2	P3	P4
1000	115692.42	110655.45	190051.38	148219.77
1100	79439.08	66471.37	97530.26	76653.66
1200	61045.586	40878.035	86872.15	64969.496
1300	60649.805	41483.46	74643.94	74065.69
1400	61781.88	44946.67	72130.63	39031.44
...	...	...	...	...

And using the graphics tools of Google Sheets, you can generate graphs such as the following:

[9] Output is not necessarily aligned since the "more" command is a direct output of what is in the file. The spaces seen are the consequence of the tab characters that are part of the file as described.



This will allow visualization of the empirical results and hence a visual tool for comparing their behaviors.

You will **share that Google sheet** with us: your professor and lab instructor, giving access to view only.

3 Requirements

For this project you must comply with the following:

DESIGN:
You are required to design your project in accordance to Object Oriented Principles.

IMPLEMENTATION:
You will write your code in Java. Your code should be readable and non-redundant, meaning that it must be written with code reusability in mind. This means that wherever existing code can be reused by just properly invoking it, your code should be able to do it. Your code should also include full documentation and comments for every part of your code[10].

INPUT FORMAT:
As specified for each part.

OUTPUT FORMAT:
As specified for each part.

USE OF JAVA CLASSES
You are not allowed to use classes that Java may provide and which implement the Set ADT since here the idea is to explore the implementations that are being described. Of course, you can use the HashMap class wherever

[10] Remember, your work here might represent the door that needs to be opened for you to be able to get a real contract from Mrs. Espargata and perhaps other potential customers that she can recommend your services to; but most importantly at the moment, the final grade that you will get in the course for this project.

needed as per the specifications given. Any other Java class that you use must be fully understood by you, and you must be ready to answer questions that the grader may have about those classes.

4 Deadline and Submission Guidelines

There will be four specific deadlines for this project as described next. Not complying with these precise instructions will cause, in the best case, reduction of point from your final score in the project. In some instances, as explained, it may cause your project to be considered as not submitted. Note that these penalties are easy to avoid, just follow the instructions.

To submit final project including Part 1 and Part 2	Sunday, March 7, 2021	Electronic submission at online.upr.edu

Submissions are due at 11:59:59 pm on the announced date.
NO LATE SUBMISSIONS SHALL BE ACCEPTED.

On the final submission, you must submit the following:

1. Zip file containing a directory, inside which your project is located when extracted and saved. That zip file shall be named as: **`P1_4020_`*`nnnnnnnnn`*`_172.zip`**. (Where *nnnnnnnnn* are the 9 digits of your student id number)
    a. It should contain your project directory.
    b. The name of that extracted folder shall be: **`P1_`*`nnnnnnnnn`***. (... the number as before)
    c. Inside this last directory is where your project folder is located. The name of your project must be **p1_40204035_202**.
2. The extracted directory (`P1_nnnnnnnnn`) will also contain the pdf README file with important information about your project, including clear instructions describing how your program can be executed from the command prompt assuming that such execution is initiated from the terminal window while located at the directory `P1_nnnnnnnnn`.

At the beginning of each file (document or Java file) you should include your ***name***, ***student id***, and ***section number***. You should verify that those instructions work. Also, your program should be able to read the input files (as specified) as long as the directory specified for those input files is a subdirectory of the same directory from where the program is executed (from where the command to execute is typed from the terminal window).

Your code should include at least the proper documentation for its classes and methods, which must follow the standards required for the Javadoc tool. You don't need to submit html files that are produced by the Javadoc tool; just make sure that your comments follow that standard.

Once your zip file is ready, please, submit it in the course portal using the appropriate open project submission homework. ANY SUBMISSION IN A METHOD DIFFERENT FROM THIS, WILL MAKE YOUR PROJECT TO BE CONSIDERED AS NOT SUBMITTED. **Failing to comply with instructions for submission may cause deduction of points in your final grade or for the project to be considered as not submitted. Projects submitted after the specified deadline (day and time) will be considered as not submitted and therefore shall not be graded.**

5 Shared Project Files

A public Git repository where you can find some useful classes for this project is available. You can click here to access it. You can freely use whatever you find there.

Let us know if you find any problem.

6 General Idea Of How Are We Going To Grade Your Project

Evaluation criteria will be shared in the following days, please stay tuned to updated versions of this document.

7 HashMap Class

The following is part of its official description (copied from the official Java web page)[11].

Class HashMap<K,V>

java.lang.Object
  java.util.AbstractMap<K,V>
    java.util.HashMap<K,V>

Type Parameters:

K - the type of keys maintained by this map
V - the type of mapped values

---

public class **HashMap<K,V>**
extends AbstractMap<K,V>
implements Map<K,V>, Cloneable, Serializable

Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits null values and the null key. (The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

This implementation provides constant-time performance for the basic operations (get and put), assuming the hash function disperses the elements properly among the buckets. Iteration over collection views requires time proportional to the "capacity" of the HashMap instance (the number of buckets) plus its size (the number of key-value mappings). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

An instance of HashMap has two parameters that affect its performance: *initial capacity* and *load factor*. The *capacity* is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created. The *load factor* is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is *rehashed* (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.

As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost (reflected in most of the operations of the HashMap class, including get and put). The expected number of entries in the map and its load factor should be taken into account when setting its initial capacity, so as to minimize the number of rehash operations. If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operations will ever occur.

If many mappings are to be stored in a HashMap instance, creating it with a sufficiently large capacity will allow the mappings to be stored more efficiently than letting it perform automatic rehashing as needed to grow the table. Note that using many keys with the same hashCode() is a sure way to slow down performance of any hash table. To ameliorate impact, when keys are Comparable, this class may use comparison order among keys to help break ties.
...
The iterators returned by all of this class's "collection view methods" are *fail-fast*: if the map is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will throw a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.
...

(The following is summary of selected constructors and methods...)

Constructor Summary

HashMap() - Constructs an empty HashMap with the default initial capacity (16) and the default load factor (0.75).

[11] We can use this, only based on the description given and without the need to know how such collection is implemented. Any valid implementation for an ADT must comply with the specification for that ADT. However, later in this course, we will study how to implement the Map ADT.

HashMap(int initialCapacity) - Constructs an empty HashMap with the specified initial capacity and the default load factor (0.75).

HashMap(int initialCapacity, float loadFactor) - Constructs an empty HashMap with the specified initial capacity and load factor.

Public Methods Summary

Return Type	Method and Description
void	**clear**() - Removes all of the mappings from this map.
Object	**clone**() - Returns a shallow copy of this HashMap instance: the keys and values themselves are not cloned.
boolean	**containsKey**(**Object** key) - Returns true if this map contains a mapping for the specified key.
boolean	**containsValue**(**Object** value) - Returns true if this map maps one or more keys to the specified value.
Set<**Map.Entry**<**K**,**V**>>	**entrySet()** - Returns a Set view of the mappings contained in this map. **[NOTE ADDED: The important thing for now, with regards to this method, is that the collection it returns is Iterable<Entry<K,V>>; hence, through it we can iterate over all the entries in the map using a for each loop: for (Map.Entry<K,V> e : map.entrySet()) ...]**
V	**get**(**Object** key) - Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
V	**getOrDefault**(**Object** key, **V** defaultValue) - Returns the value to which the specified key is mapped, or defaultValue if this map contains no mapping for the key.
boolean	**isEmpty**() - Returns true if this map contains no key-value mappings.
Set<**K**>	**keySet()** - Returns a Set view of the keys contained in this map.**[NOTE ADDED: The important thing for now, with regards to this method, is that the collection it returns is Iterable<K>; hence, through it we can iterate over all the entries in the map using a for each loop: for (K key : map.keySet()) ...]**
V	**put**(**K** key, **V** value) - Associates the specified value with the specified key in this map.
V	**remove**(**Object** key) - Removes the mapping for the specified key from this map if present.
int	**size**() - Returns the number of key-value mappings in this map.

8 End