Hashtables, Sorting and Complexity

CS 151 - Introduction to Data Structures

Assignment 8 - due Friday 4/14

In this assignment, we'll explore hashtables, sorting and compare the efficiency of different methods of removing data duplication. Note that the data set we will be working with this time is very large (685,725 rows and 116 columns). Using Excel or Numbers to look at it is NOT recommended. It will take forever to open. VSCode or Emacs/Vi is really your best option here. For the same reason, you should write efficient code - pay attention to the big-O and no unnecessary loops!

1 Data Deduplication

One important, and potentially expensive task when handling data is data deduplication, whose goal is to make sure that each item in a data set only appears once. We'll explore three different ways this can be done and determine which one is most efficient.

1.1 The Data: Stop, Question, and Frisk

The New York City police department follows a practice of regularly stopping, questioning, and frisking people on the street. A 2011 lawsuit against this practice by the NYCLU successfully curbed the police department's frequent use of this practice, though concerns remain. The NYCLU's description of this practice is below:

> The NYPD's stop-and-frisk practices raise serious concerns over how the police treat the people they're supposed to protect. The Department's own reports on its stop-and-frisk activity confirm what many people in communities of color across the city have long known: The police continue to target, stop, and frisk people of color, especially young black and Latino men and boys, the vast majority of whom have done nothing wrong.

> The NYCLU's 2011 lawsuit, along with other litigation, helped curb the NYPD's stop-and-frisk program. Today, the number of stops are just a fraction of what they were at their peak, though troubling racial disparities remain. The NYCLU continues to monitor, analyze, and shape stop-and-frisk practices through data analyses, advocating for legislation to bring more transparency to policing practices, and working in coalition with community partners.

More information about the data can be found via the NYCLU's website:
`https://www.nyclu.org/en/issues/racial-justice/stop-and-frisk-practices`
`https://www.nyclu.org/en/stop-and-frisk-data`
and via the NYPD:
`https://www1.nyc.gov/site/nypd/stats/reports-analysis/stopfrisk.page`.

We'll be looking at the data collected in 2011 at the height of the NYPD's practice of stop-and-frisk. Each row in the data represents a single stop by a police officer, and the attributes (columns) include information about the stop (e.g., whether a weapon was found), information about the person stopped (e.g., age, race, etc.), and information about the stop location (e.g., the address). The dataset (distributed to you in the usual `handouts/` directory) can be found via the NYPD website:
`https://www1.nyc.gov/assets/nypd/downloads/zip/analysis_and_planning/stop-question-frisk/sqf-2011-csv.zip`
along with the codebook describing what is included in the data:
`https://www1.nyc.gov/assets/nypd/downloads/zip/analysis_and_planning/stop-question-frisk/SQF-File-Documentation.zip`

1.2 Determining Equality

The deduplication goal we will consider here is to collect a list of *individuals* who were stopped during the year. One way to think about this is that we would like to know which people were stopped multiple times by the NYPD in 2011.

In order to determine if two rows contain the same person, we need to develop a method that checks the information that should be unique to each person and compares it to determine equality. How exactly the quality test works is entirely upto you. You should decide what information (which columns) to use by examining the row data (while referencing the codebook) carefully, or deduplication will fail.

Requirements:

1. Design a class to hold a single row from the CSV. It does *not* need to hold

all fields in the CSV - you should store only the fields that will be necessary in order to deduplicate the data.

2. Implement the `Comparable` interface by providing a `compareTo` function in the class you just created that returns 0 if and only if the compared items are the same person according to the way you have decided to check uniqueness.
3. Describe and justify how you are determining uniqueness in your `README` file.

1.3 Data Storage

You'll need a class to store the full data set.

Requirements:

1. Make a class to store the full data set. Your constructor for this class should take the name of a CSV file as input and should do the work of reading in the data from the CSV and parsing it into an `ArrayList` containing the objects you developed in the previous section.

1.4 Deduplication Methods

You will add five data deduplication methods to the data set class you created in the previous section. Each of these methods should return an `ArrayList` of your designed objects that contains only the non-duplicated individuals who were stop-and-frisked.

All Pairs Deduplication One way to find the duplicates in a data set is to compare each item to each other item in the data set, checking duplicates as you go. Make sure not to count the comparison of an item to itself as a duplicate. We will call this the "all pairs" method of data deduplication.

Requirements:

1. Create a method that uses this "all pairs" method of deduplication to return a list of non-duplicated items in a given data set. The method signature should be:
   `ArrayList<E> allPairsDeduplication()`
   where `E` can be substituted with the specific object you have created to store a single row.

Hash Table Deduplication Another way to find the duplicates is to create a key for each item in the data set and insert these items into a hashtable. Items hashing to the same key will overwrite each other, therefore resulting in deduplication. The choice of key will determine whether two items are considered to be duplicates, so choose it carefully.

2. Create a method that uses this hashtable-based method of deduplication to return a list of non-duplicate items in a given data set. You may use the `ProbeHashMap.java` from the book. You program should first creates a `ProbeHashMap` with the size 1000003 (use the one-parameter constructor) and then insert the items into this hash table.

   The deduplication method signature should be:
   `ArrayList<E> hashLinearDeduplication()`
   where `E` can be substituted with the specific object you have created to store a single row.

   Besides the list of non-duplicates, also collect the following statistics about hashing: average number of probes during insertions, max number of probes during insertions and load factor after insertions. Note that you will need to update `ProbeHashMap` class to compute these values. Print out these statistics once after you have inserted all elements into the hashtable in the following format:

   ```
   Average numer of probes: XXX
   Max number of probes: XXX
   Load factor: XXX
   ```

3. Now implement a `DoubleHashMap` class which extends `AbstractHashMap` and implements a hash table that supports double hashing on collision. Starting from `ProbeHashMap` and modifying it to add a secondary hash function is recommended. Then create a `DoubleHashMap` and repeat the above.

   The deduplication method signature should be:
   `ArrayList<E> hashDoubleDeduplication()`
   where `E` can be substituted with the specific object you have created to store a single row.

   Did the hashing statistics change? Discuss in your README (see details in Section 2).

Sorting for Deduplication The last method for deduplication that we'll look at sorts the data to check for duplicates. Note that the comparison method you use will play an important role in determining if you correctly find the duplicates. First, we'll need some functions that allow us to sort.

3. Write a method to perform a quicksort on your data. As usual, object comparisions should be performed via calls to `.comparedTo`.

4. Java has a library with a buit-in sort method! Write a method that uses `Collections.sort()` to sort your data.

5. Create two deduplication methods that use these different sorting functions. The method signatures should be:
   `ArrayList<E> quickSortDeduplication()`
   `ArrayList<E> builtinSortDeduplication()`
   where `E` can be substituted with the specific object you have created to store a single row.

2 Complexity Analysis

You will now explore the complexity of the deduplication methods you've developed. The answers to these questions should be given in your README file.

Requirements:

1. How much time (in milliseconds) does each duplication counting method take when run on the SQF data set?

2. Create a chart (with Excel or any other plotting software) showing the number of rows processed on the x-axis and the corresponding time used in milliseconds on the y-axis for each of these methods (where each method is a series on the graph). Note that this requires that you collect time data in your program using the following code snippet that you saw when we first talked about complexity and analysis of algorithms.

   ```
   long startTime = System.currentTimeMillis();
   // run code
   long endTime = System.currentTimeMillis();
   long elapsed = endTime - startTime;
   ```

   Save the graph and name as `complexity.png`. Include this in your submission directory.

3. Explain which method is most efficient in your README, also note which sorting method (the built-in method or your quicksort method) was more efficient and hypothesize why this might be.

4. Comparisons of the statistics of the two hash tables and hypothesize which one is more efficient and why.

3 Command Line Input

You will receive a stop-and-frisk records file on the command line as a single argument like this:
`2011.csv`
Remindert that input filenames may have additional (/-separated) path in front. You should process that file as described above and print the following information out in response.

```
Records given:2000
Attributes checked:X,Y,Z
Duplicates found:100
```

where in this example you were given a file with 2000 lines of police stop-and-frisk records, determined equality based on attributes X, Y, and Z, and found 100 duplicates in the data (i.e., deduplication returned a list of length 1900). Obviously you should only call one of your five methods. Use your most efficient one.

Note that we may choose to test your work using data from a different year, so be sure to actually read the data in from the command line and not hard-code the file name.

Requirements:

1. Take in a CSV file from the command line input.

2. Deduplicate the data and print out the results formatted as above.

4 Electronic Submissions

1. README: The usual plain text file `README`

Your name:

How to compile: Leave empty if it's just `javac Main.java`

How to run it: Leave empty if it's just `java Main`

Known Bugs and Limitations: List any known bugs, deficiencies, or limitations with respect to the project specifications. Documented bugs will receive less deduction versus uncaught ones.

Discussion: Design of equality comparison as explained in Section 1.2, and all required complexity discussions as explained in Section 2.

2. **Source files:** all `.java` files

3. **Chart image:** `complexity.png`

DO NOT INCLUDE: Please delete all executable bytecode (`.class`) files prior to submission. Please also *not* include 2011.csv in the submission, it's too large.

To submit, store everything (README, `complexity.png` and source files) in a directory called `A8`. Then follow the directions here:
`https://cs.brynmawr.edu/systems/submit_assignments.html`