Sustainability Complexity, Hash Tables, and Sorting

Haverford CS 106 - Introduction to Data Structures

Lab 7 - two weeks

In this lab, we'll be exploring the relationship between computational complexity - the efficiency of the programs we write - and the energy consumption our programs use. An important initial caveat - the calculations we'll be doing here are really "back of the envelope" style calculations. They're an estimate, and may actually be far from the real values, but hopefully they'll give us a sense of how computational complexity relates to energy consumption.

A few initial reminders / things to know for the less physics savvy among us:

- Watts (W) - watts are a measure of power and they meausre Joules per second.
- Watt Hours (Wh) - watt-hours are a measure of energy consumption and they measure watts used in an hour.

Note also that we'll be converting from seconds used to energy consumption (watt hours). Computational complexity is more often measured not based on seconds, since the seconds used can vary from computer to computer, but based on computation steps used. If we were going to convert that to energy consumption, we'd first need to determine the performance per watt (number of computation steps per watt) that our computer could perform.

1 Getting Started

We'll start by setting up a few variables and functions that will be useful for the later steps of this lab. It's up to you how to organize and name these variables and functions. Remember: always try to follow best coding practices such as style guides. This includes, e.g., naming static variables in all caps and appropriately determining which methods should be private or public. Also add anything that will be useful in order to use your created class in later parts of the lab (e.g., initialization and accessor methods).

1. Start off by setting up a static variable that will be useful in this lab: the watts used by your computer. You can use a default value of 200, or look up the specific value for the computer you're using.

2. Find something that consumes (a small amount of) energy and determine how many watt-hours that uses. For example, you might find the number of watt-hours it takes to make a cup of coffee. Make a variable for this amount. In later parts of the lab you'll be expressing your programming work in, e.g., energy it would take to make a cup of coffee.

3. We'll be determining the complexity of a program by timing it. Make a method that takes a floating point number of seconds and returns the number of watt-hours consumed.

4. Make a method that computes the number of the item you've chosen above (e.g., cups of coffee) that could be achieved with a given amount of energy (watt-hours). Then use that to make another function that takes seconds of computation time and returns the number of your chosen items that could be achieved.

5. Now that we can convert computation seconds used to units of something (e.g., cups of coffee), we can determine how many of these thing could be created from the energy that one of our programs uses! In order to do that, we'll need to be able to determine the number of seconds taken by a program. Make methods that allow you to start a timer, end a timer, and get the number of seconds that elapsed using the System class.

We'll be trying out these methods and analyses on more interesting questions later in this lab, but for now let's put all of this together by analyzing the running time of the Fibonacci function:

1. Create a function that returns the nth Fibonacci value when given n.

2. Then use the functions you've built to time the function and convert it to the number of other things that could have been created using that energy (e.g., cups of coffee). Getting the precise time of a specific function is tricky, since there are other things being run on your computer at the same time. To mitigate some of this issue, run the given function multiple times and return the average number of seconds taken by the function.

3. Determine and print the number of seconds it takes to calculate the 500th Fibonacci value and the number of items that could be created using that amount of energy.

4. Make a function that can plot the value of n on the x-axis versus the number of items on the y-axis. You can do this using a library called Tablesaw. First add the Maven dependencies to your `pom.xml`:

   ```
   <dependency>
       <groupId>tech.tablesaw</groupId>
       <artifactId>tablesaw-core</artifactId>
       <version>0.30.2</version>
   </dependency>
   <dependency>
       <groupId>tech.tablesaw</groupId>
       <artifactId>tablesaw-jsplot</artifactId>
       <version>0.30.2</version>
   </dependency>
   ```

   Then you can use LinePlot to graph it. Here's a small example of how that works:

   ```
   double[] xvals = {1, 2, 3, 4};
   DoubleColumn column1 = DoubleColumn.create("x-axis", xvals);
   double[] yvals = {2, 4, 6, 8};
   DoubleColumn column2 = DoubleColumn.create("y-axis", yvals);
   Table table = Table.create("for plot");
   table.addColumns(column1, column2);
   Plot.show(LinePlot.create("title", table, "x-axis", "y-axis"));
   ```

   If you have multiple algorithms you would like to compare (as you will later in this lab), you can add series to the graph so that there are multiple colors as in the below example:

   ```
   double[] xvals = {1, 2, 3, 4, 1, 2, 3, 4};
   DoubleColumn column1 = DoubleColumn.create("x-axis", xvals);
   double[] yvals = {1, 2, 3, 4, 1, 4, 6, 8};
   DoubleColumn column2 = DoubleColumn.create("y-axis", yvals);
   String[] categories = {"a", "a", "a", "a", "b", "b", "b", "b"};
   StringColumn catcolumn = StringColumn.create("algorithm", categories);
   Table table = Table.create("for plot");
   table.addColumns(column1, column2, catcolumn);
   Plot.show(LinePlot.create(
        "title", table, "x-axis", "y-axis", "algorithm"));
   ```

In addition to loading the correct data into the above table, you should be sure to customize your axis and graph titles appropriately based on the energy units you chose to consider and graph.

5. **Hand in your work** for this section by saving your resulting line graph as the file `fibonacci.png` and adding, committing, and pushing that to the top level of your directory structure for this lab.

2 Data Deduplication

One important, and potentially expensive (in terms of efficiency and energy consumption), task when handling data is data deduplication. The goal of this task is to make sure that each item in your data set (e.g., each person) only appears once. We'll explore 3 different ways this can be done in this lab, and determine which one is most efficient.

2.1 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 (also distributed to you in your lab files) 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`

2.2 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 function that checks the information that should be unique to each person and compares it to determine equality. We need to decide what information this should be carefully, or deduplication will fail.

Requirements:

1. Create an object to hold a single row from the CSV. This object 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 making a `compareTo` function in the object 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.

2.3 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 the 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. Recall that you can use the `opencsv` library to do much of this work for you.

2.4 Deduplication Methods

You will add three 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 number of duplicates in a data set is to compare each item to each other item in the data set, counting 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 the non-duplicate 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 number of duplicates is to create a key for each item in the data set and insert these items into a HashMap, with the count of the number of times this item is seen as the value. 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, also included in your starter files. Your program should first create a `ProbeHashMap` with the capacity `1,000,003` 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 number of probes: XXX
Max number of probes: XXX
Load factor: XXX
```

Sorting for Deduplication The last method for deduplication that we'll look at involves sorting the data in order 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. You may find the fact that you implemented the Comparable interface useful.

4. Java has a library with an already implemented 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.

3 Complexity Analysis

Using the methods you developed from the "getting started" part of the lab, you'll now explore the complexity of the deduplication methods you've developed in terms of energy consumption. The answers to these questions should be given in your README file.

Requirements:

1. How many seconds does each duplication counting method take when run on the SQF data set?
2. How many items (e.g., cups of coffee) could be created by this amount of energy for each duplication counting method?
3. Create a graph showing the number of rows deduplicated on the x-axis and the energy consumption in number of items on the y-axis for each of these methods (where each method is a series on the graph). **Save the graph** as the file `deduplication.png` in your repository and be sure to `git add`, `git commit`, and `git push` to submit this graph.
4. Explain which deduplication method is most efficient in your README, also note which sorting deduplication method (the builtin method or your quicksort method) was more efficient and hypothesize why this might be.

4 Command Line Input

You will receive a stop-and-frisk records file on the command line as a single argument like this:
`2011.csv`
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 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). 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. Be sure to print out the *exact* name of the attributes checked as given in the first row of the CSV file.

Requirements:

1. Read in a CSV file from the command line input.
2. Deduplicate the data and print out the results formatted as above.

5 Extra Credit

All extra credit should only be done **after** successful completion of all of the base requirements for this assignment. The number of points awarded for extra credit will be smaller than those for completion of the base requirements and the extra credit is designed to be *harder* than those basic requirements as well. You may choose which of the extra credit options below to pursue and can receive credit for some and not others where that makes sense. *If you implement any of these options, identify the work that you did in the README file.*

1. Implement a `DoubleHashMap` class which extends `AbstractHashMap` and implements a hash table that supports double hashing on collision. Note that you will need to define a secondary hash function for this part. 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 in comparison to the linear probing hashtable? Discuss in your README.

2. Implement quicksort using an in-place method.