Homework 3

GWU CSCI 1112 - Fall 2019

September 26, 2019

1 Introduction

This homework is intended to reinforce your understanding of working with referential copies of array, sorting algorithms, recursion, and the implicit costs involved with algorithms in general. Your task is to implement and analyze the cost of different algorithmic approaches using the provided framework.

1.1 Deadline

October 2, 2019 at 11:59pm

1.2 Submission

You must create a `.zip` file containing all of the files that you develop and work with and you must submit only the `.zip` file to blackboard before the deadline.

Your `.zip` file must be named using the following naming convention: `<yourNetId>-hw-03.zip`
For example, my NetId is `jrt`. If I submitted the homework, I would name my `.zip` file: `jrt-hw-03.zip`

Do not submit the compiled `.class` files. You must submit any `.java` files and other supporting files, *i.e.* data files like `posts` in this problem.

1.3 Grading Rubric

- 50% For successful compilation
- 10% For sufficient comments
- 10% For consistent coding style
- 20% For base implementation
- 10% For extension implementation

1.4 Unit Testing

As in the previous homework, a unit test file `SoNetUnitTest.java` has been provided in the framework. You should not need to make any changes to this file, but you may make changes to help your debugging efforts. However, your code will be evaluated against the unit test file originally provided in the framework regardless of changes that you make.

The unit tests provided with this assignment show the basis for how this assignment will be automatically tested. We will also look at the implementations to ensure that you are not simply gaming the unit tests. Your implementation grade will be subject to the number of unit tests that your program passes.

In subsequent assignments, if a unit test file is provided expect it only to be a skeleton and you will be responsible for implementing unit tests to validate your implementation. We will use our own unit test to evaluate your work, so you will not necessarily know exactly what testing metrics will be applied to your implentations. I encourage you to look closely at this and the previous assignment for models on how to go about unit testing.

1.5 Comments

Again, I have provided relevant comments to document the files (a file header), the function signatures (function block comments), and inline comments in much of the unit test file. These comments are a model that you should follow in the future when developing files or functions on your own. This level of comments will be expected in future assignments.

You must substitute your name in the file header where indicated and provide any inline comments to document code that you contribute to your solution.

In subsequent assignments, comments in the framework will begin to be omitted and you will be expected to document the code yourself. Again, I encourage you to look closely at how files, functions, and implementations have been documented so far and to follow these examples as a model for future work.

1.6 Plagarism

We will use a set of automated tools specifically designed to analyze code for plagarism. If you copy code from another source, classmate or website, there is a very high probability that these tools will flag your work as plagarized. You are permitted to discuss the problems at a high level; however, you must code your own solution. If you do not share code or outright borrow code from a website, you will have no problem with the plagarism filter.

2 Social Media

Social media describes a wide variety of applications where users provide ratings on ideas or services. Some examples include Facebook, Reddit, Yelp, and Uber. A developer often leverages user rating data in order to promote information to other users. Some sites use a positive and negative rating approach where users can express their support or dislike for particular information. The difference between positive and negative ratings is often referred to a ratio.

One major consideration for a social media site is the volume of traffic and the frequency with which ratings may change due to the evolving feedback from a large user base. Users are often attracted to social media sites to gather real-time information such as using a social media site as a news proxy and demand often increases when a socially significant event occurs, *e.g.* natural disaster, political process, or celebrity event.

When designing a social media platform that organizes data by ratio, a developer must consider how to rapidly respond to changing ratios. If ratio is the basis for organization of posts and if high volume can result in rapidly changing ratios, then the developer will need to implement a policy that dictates the frequency of post sorting. Recall that sorting is more expensive than searching and users demand fast responses, so it may be better to periodically index, *i.e.* sort, and provide users with a "snapshot" of the index table in order to minimize the amount of sorting time.

When dealing with sorting, there are two important considerations that must be balanced: how fast must the sort be and therefore how frequently the sort is performed, and how much memory is necessary to perform the sort. Which of these considerations is the most important requirement is a matter of analyzing the requirements of the product that you are developing. For a social media site on large servers where there is high frequency usage, speed trumps memory in requirements, but for other applications such as embedded code into a microcontroller where the data update is infrequent, memory may be a much more significant concern.

A framework for this program has been provided in `SocialNet.java` and a set of unit tests have been provided in `SoNetUnitTest.java`. There is also a data file `posts` and a file reader `PostReader.java` that you should

not modify but you are welcome to look at. In order to compile and test your work, compile `SoNetUnitTest.java` and run `SoNetUnitTest`.

2.1 Data Structures

The framework assumes a number of array based data structures which are consistently named or prefixed as `posts`, `view`, `post`, and `profile`. This section specifies the intent and usage of the data structures.

2.1.1 `posts`

`posts` is a master list of post information. This data structure represents a snapshot from a large data set of post data which may originate from a data base of some sort. As a snapshot, it reflects the current state of the post information for the website at one instant in time. Consider that this information may be changing tens of thousands per second and you might grasp the scale of the problem involved with designing an effective social media site.

The structure of `posts` consists of rows (1st dimension) of `post` records (2nd dimension). Each post record consists of three fields of integers (see `post` below).

2.1.2 `view`

A `view` is a referential (shallow) copy of the master list of `posts`. A `view` is a shallow copy because it will allow changes to records in the master set of `posts` to be reflected in the `view`.

The structure of a `view` is the same as `posts`.

2.1.3 `post`

A `post` is a record (or row) in a set of post data.

The structure of a `post` consists of an array of three integer values. The 1st field (index 0) is the post id which is an identifier unique to each post. The 2nd field (index 1) is the number of "ups" or the number of positive responses to a post. The 3rd field (index 2) is the number of "downs" or the number of negative responses to a post.

2.1.4 `profile`

A `profile` is an array that is used to track the performance of a sort.

The structure of a `post` consists of an array of three integer values. The 1st field (index 0) accumulates the number of allocations on the stack that have been made during the sort. The 2nd field (index 1) accumulates the number of comparisons that are made during a sort. The 3rd field (index 2) accumulates the number of swaps that are made during a sort.

It will be difficult to get a fully accurate measure for profile data and we really only want to get an idea of what the hidden costs are. You will not count any of the `profile` operations or allocations, but you will count these actions for the sort itself. When the profile data is printed for a full implementation, you should see orders of magnitude more allocs for recursive over iterative and orders of magnitude more compares and swaps for iterative over recursive which should help illustrate the hidden cost of these implementations

When profiling programs, we use a different approach; however, there is an important lesson about pass-by-value and changing data accessed by reference type hidden in this approach especially when implementing the recursive sorting.

2.2 Base

The requirements in this section constitute a 90% solution. You must implement the following functions using the following function signatures:

```
public static int[][] createView(int[][] posts)
public static int differential(int[] post)
public static boolean lessThan(int[] post1, int[] post2)
public static void swapPosts(int[][] view, int i, int j)
public static int[] iterativeSort(int[][] view)
```

2.2.1 Base Functions

This section gives a general explanation of the requirements for each function. More information may be available in the function headers and in how the functions are unit tested.

```
int[][] createView(int[][] posts)
```

`createView` performs a shallow copy or referential copy of a set of `posts` into what is called a `view` and returns the `view`. The intent is that if post information in the master set of `posts` changes, we want the information to be known to the `view`. If we performed a deep copy, we would have to have a separate process to either recopy `view` or to execute a large number of searches to update an existing `view`. Note the contrast between this requirement and the requirements for Steganography.

```
int differential(int[] post)
```

`differential` computes the virtual `differential` for a single `post` record and returns the `differential`. The `differential` is simply the difference between `ups` and `downs`. `differential` is virtual because it is not stored in a `post` record.

```
boolean lessThan(int[] post1, int[] post2)
```

`lessThan` performs a less than comparison between two `post` records where `post1` acts as the left operand and `post2` acts as the right operand. The basis of comparison is the `differential` between the two operands. You must use the `differential` method in this implementation.

```
void swapPosts(int[][] view, int i, int j)
```

`swapPosts` performs a shallow swap between two `post` records (at indices `i` and `j`) in a `view`. This is a referential swap and not a deep data swap. The intent is that the original post data in the master set of `posts` should be completely unaffected by this swapping operation.

```
int[] iterativeSort(int[][] view)
```

`iterativeSort` performs a sorting operation on a `view` (a referential view of the master set of `posts`) using one of the iterative sorts that we discussed: selection, bubble, or insertion sort. `iterativeSort` returns an array of `profile` information as detailed in the Data Structures section above. You may implement addition functions to support this sorting operation; however, take care that the `profile` information is accurately maintained and accounted for over any subsequent funtion calls.

2.3 Extension

The requirements in this section represents the remaining 10% of your potential grade. You will receive no credit for the extension if your base implementation is incorrect. Focus first on getting the base program fully correct, then consider the extension requirements. You must implement the following functions using the following function signatures:

```
public static int[] recursiveSort(int[][] view)
```

2.3.1 Extension Functions

```
int[] recursiveSort(int[][] view)
```

`recursiveSort` performs a sorting operation on a `view` (a referential view of the master set of `posts`) using recursive quicksort. `recursiveSort` returns an array of `profile` information as detailed in the Data Structures section above. You may implement addition functions to support this sorting operation; however, take care that the `profile` information is accurately maintained and accounted for over any subsequent function calls.