CS2020: Data Structures and Algorithms (Accelerated)

Problem Set 2

Due: January 29th, 11:59pm

Overview. Sometimes you have an idea for an algorithm. It seems like a really good idea. It *should* result in much better performance. Unfortunately, it is hard to know for sure whether it really does work. What do you do? In Problem 1, we consider the problem of "List Maintenance." It seems like a very natural idea to move items that are frequently accessed to the front, leaving infrequently accessed items at the back of the list. Your job is to simulate this process and see how well it works.

In Problem 2, we have a large database to process. The goal is to extract some useful information out of the database without having to read the entire big database into memory. The fewer the number of data accesses, the better.

Collaboration Policy. You are encouraged to work with other students on solving these problems. However, you **must** write up your solution **by yourself**. We may, randomly, ask you questions about your solution, and if you cannot answer them, we will assume you have cheated. In addition, when you write up your solution, you **must** list the names of every collaborator, that is, every other person that you talked to about the problem (even if you only discussed it briefly). Any deviation from this policy will be considered cheating, and will be punished severely, including referral to the NUS Board of Discipline.

Problem 1. (**List Maintenance**)
The problem with lists is that they are either expensive to access or expensive to modify. For example, consider two possible designs for a list that is stored in an array:

1. *Sorted:* The list is kept in sorted order. Searching a list of n elements takes time $O(\log n)$ using binary search. Inserting a new element takes time $O(n)$, as you have to make room for the new element in the array.

2. *Unsorted:* The list is not sorted. Searching a list of n elements takes time $O(n)$ using a linear search of the array. Inserting a new element takes time $O(1)$, as you simply add it to the end of the array.

Here is one idea for improving the performance of a list: when you insert an item in the list, add it to the end of the list; when you access an item in the list, move it to the front of the list—and move every item in the list back by one slot.

How much does this strategy help? If the same element is accessed repeatedly, then this strategy should help a lot. Many real-world systems have exactly this type of *locality*, i.e., the same elements are used over and over again, and hence you would expect this strategy to help.

For this problem, we have implemented a simple list class `FixedLengthList`, and a simulator class `ListSimulator` that simulates the list. The following are features of the simulator and the provided list implementation:

- The `FixedLengthList` class stores the list in an array of fixed length. We assume that the list is designed only to store non-negative integers.

- The `final` (constant) variable `LISTSIZE` defines the size of the list to simulate, while the `final` variable `NUMQUERIES` defines the number of times the list will be searched during the simulation.

- For the purpose of the simulation, the list consists of exactly the integers:

$$0, 1, 2, 3, \ldots, \texttt{LISTSIZE} - 1$$

- Each element in the list is accessed according to a fixed distribution. For the purpose of simulation, we assume that if $i < j$, then i is accessed with larger probability than j. For example, the integer 0 is accessed with the highest probability, and the integer `LISTSIZE` $- 1$ is accessed with the lowest probability.

- The distribution is stored in the variable `m_ListProb`, and is generated by the method `generateListProbs`. The probability of accessing integer j is stored in `m_ListProb[j]`. For example, the probability of accessing 2 is stored in `m_ListProb[2]`.

- The simulator can be configured to run three different experiments, depending on the initial ordering of the items in the list. For the `DECREASING` experiment, items are initially added to the list in order of decreasing probability: the most accessed elements are at the beginning of the list. For the `INCREASING` experiment, items are initially added to the list in order of increasing probability. For the `RANDOM` experiment, items are initially added to the list in a random order. Intuitively, the `DECREASING` order is the best order for the list.

- The `main` method demonstrates how to use the simulator to run the three experiments for the `FixedLengthList` class. Notice in particular how the `StopWatch` class is used to measure time.

Problem 1.a. The class `FixedLengthList` implements `add` by adding an element to the end of the list, and `search` by performing a linear search through the list.

Write a new class `MoveToFrontList` that extends `FixedLengthList` and overrides `search`. The new version of `search` should:

- Perform a linear search for the specified item.
- If the item is found, then move that item to the front of the list (i.e., to slot 0 in the array) by shifting every intervening item back one slot.
- Return `true` if the item is found and `false` if it is not.

Your new class *must* extend `FixedLengthList`, i.e., be a child class of the fixed-length list. (Otherwise, the simulator will not simulate it!) Use the `MoveToFrontListTest` JUnit test to test your solution.

Problem 1.b. Implement one other list maintenance heuristic of your choosing. That is, extend the `FixedLengthList` such that it implements search in a different manner (perhaps using some alternate rule as to how to move items when they have been searched for). Call your new class `MyFastList`. (Who can come up with the fastest algorithm?) Explain clearly how your list algorithm works. (Remember that the list class is not allowed to know anything about the distribution being used by the simulator.)

Problem 1.c. Run several experiments comparing the performance of the FixedLengthList, the MoveToFrontList, and MyFastList. (Again, look at how the simulator uses the StopWatch class to measure time.) Which one performs best? What is your conclusion on the efficacy of the *Move-to-Front* heuristic?

Problem 1.d. (Optional.) One major problem with the current strategy is that searching for an item *not* in the list always takes $O(n)$ time for a list of size n. That is, to discover that an item is not in the list requires exhaustively scanning the entire list. Can you solve this problem as follows: whenever a client searches for item k and it is not found, insert $-k$ in the list and move it to the front. Now, whenever a search for k finds $-k$, it can stop searching—and move that item to the front of the list. Notice that you also have to update the `add` routine to make this work (and it will not work for $k = 0$).

Try this idea out and see how much improvement you get. What are the advantages and disadvantages of this strategy? Using the negation of a key as a signal is, in fact, a bad idea from a software engineering perspective. What is the right way of implementing this strategy?

Problem 2. Herbert the Robotic Clown

As we all know, the Acme Corporation is a world leader in making people smile. (Motto: *For fifty years, the leader in creative mayhem!*.) The Acme Corporation is well-known for its wide selection of anvils and birdseed, along with specialty items such as dehydrated boulders and earthquake pills. However, in recent days, Acme stock has been depressed, and their inventors have been hard at work coming up with new ways to make people happy.

Their most recent invention is a robotic clown named *Herbert*. He is a fully automated clown, well equipped with all of your favorite Acme gizmos, and he can be hired to perform at your party! While Herbert is an excellent clown, he is not so good at math. (He was programmed with a very funny joke about how $1 + 1 = 3$, and ever since he has had great difficulties.) Your job, in this problem, is to help Herbert with the financial aspects of his job.

Problem 2.a. Herbert is paid by the minute. Sometimes, he is hired for a very short performance (perhaps, only 1 minute long), and sometimes he is hired for a long engagement that may last for weeks. Herbert's salary, however, is very complicated: he is paid different amounts for different lengths of time. (An additional complication is that Herbert is payed in SmileBucks, which Herbert then may exchange for spare parts and electricity. Currency conversion and bartering techniques are outside the scope of this problem set.)

Herbert has given you an employment log of his to analyze. The log is very long, containing data from the last several months. Each line of the log represents one minute of work. Hence a job that takes ten minutes is represented by ten lines in the log.

Each line in the log consists of two components: the name of the employer, and the total amount of SmileBucks (SB) that Herbert has been paid for the job, up to and including that minute. For example, consider the following two jobs where Humperdink hires Herbert for 5 minutes and Hortensia hires Herbert for 2 minutes:

```
Humperdink:7
Humperdink:10
Humperdink:12
Humperdink:14
Humperdink:16
Hortensia:10
Hortensia:11
```

Here, we see that Herbert was paid 7 SB by Humperdink for the first minute. He was paid an additional 3 SB for the second minute, yielding a total of 10 SB. In total, for the five minutes, he was paid 16 SB. Herbert then was paid 11 SB by Hortensia. You can assume that each employer only hires Herbert once, i.e., all the names of the employers are unique.

You have been provided with a file `HerbertLog.java` which contains the `HerbertLog` class. This class has a constructor, which takes one argument: the filename containing the log. It provides you with two methods:

- `numMinutes()`: returns the total number of minutes in the log file.
- `get(int i)`: return the record associated with minute i.

Notice that the record returned by the `get` method is an object of type `Record` (see the provided `Record.java` class).

Your job is to add a new method to the `HerbertLog` class that calculates the total amount of money made by Herbert over the duration of that log. That is, create a new method: `int calculateSalary()` that returns the total salary of Herbert.

Notice that a simple solution involves reading every single line of the log. However, the log might be very long (even if it contains relatively few jobs). Devise an algorithm which retrieves as few records as possible. (That is, call the `get(int i)` method the minimum number of times.) Partial credit will be given for any working solution, even if inefficient. More credit will be given for better (more efficient) solutions.

In order to test your solutions, we have released various logs containing Herbert's history. Report the number of `get` operations you require for each of the sample files. Use the `HerbertLog.java` JUnit test to check your answers.

Problem 2.b. (Bonus problem.) Herbert has one flaw: he is lazy. He does not want to work so many long, long jobs. He would prefer that none of his jobs be more than ten minutes. Or perhaps, 20 minutes? What should be the maximum length job that Herbert performs? He needs to make at least enough SmileBucks every month to pay for his spare parts!

Your job is to create a new method: `int calculateMinimumWork(int goalIncome)`. This method returns the *minimum* number of minutes m such that if Herbert works no more than m minutes for each employer, then he makes at least the specified incoming. Consider the following example:

```
Amelia:10
Amelia:20
Amelia:30
Bert:40
Chloe:1
Chloe:2
Chloe:3
```

If the desired income is 60, then the method should return 2: if Herbert works for at most 2 minutes for each employer, then he will make 62 SB. If he only worked 1 minute for each employer, then he would make only 51 SB, which is not enough. (Notice that the most he can make in this case, by working for up to 3 minutes per employer, is 73.)

Again, minimize the number of times that you query the log. Partial credit will be given for slow solutions (in this case, as long as they do not read every line in the log), but faster solutions that read the log fewer times will get more credit.

In order to test your solutions, we have released various logs containing Herbert's history. Report the number of `get` operations you require for each of the sample files. Use the `HerbertLogBonus.java` JUnit test to check your answers.