Brandon Allen

Code Challenge WriteUp

11/3/19

Part 1

**1. Explain your improvement process, and why you chose each step.**

The initial code was simple, but it did not do a great job of conveying its purpose just by reading through it. It looped through the lines of a file, reading them into a list after converting them to Int32 values. On each iteration it would do some math on the list and print the result to stdout. I would not have known it was calculating the interquartile mean for the set if not for the included instructions.

The first step I took to improve the readability of the code was create a class to describe the purpose and to break up the inner pieces of the loop. This was called DataSet. It represented the list of data points and allowed me to add functionality while minimizing side effects. For example, I could extend the List Add() method by creating an AddPoint() method to DataSet. If I wanted to optimize the add method to suite our data needs I could do it here. It also lets other developers know that if they want to tweak how data is added to the set, this would be where to do it.

The other method I added was GetIQMean(). This encapsulated calculating the interquartile mean of the current set of data. This helped make it obvious what calculation was going on. It also allows for a testable unit of code to validate the calculation is done correctly.

**2. What improvements did you make to ensure that future developers could quickly and easily understand this code?**

In addition to creating a class to wrap the functionality of the data calculation, I added unit tests to validate the correctness of the AddPoint() and GetIQMean() methods. For other developers, the unit tests provide a way to help describe the different scenarios possible in each unit and help uncover any assumptions.

Comments are another important part of extending the readability of a program. I added comments to describe classes, methods, and inner pieces of units to describe my approach to certain scenarios.

Part 2

**1. Explain how your optimization works and why you took this approach**

The first opportunity for optimization I saw was the step where data is inserted at the end of a list and then sorted at each iteration of the loop. When working up to 100,000 data points, this becomes a pretty significant bottleneck. Since the default comparer for Sort() has Big O of O(n^2) in the worst case, we're looking at Big 0 of (x\*n^2) where x is the number of data points were looping over, effectively O(n^3). To remedy this, instead of always inserted a new data point at the end, I created an AddPoint() method to insert the data in order, using a divide-and-conquer method. The Big O for the new AddPoint was near constant time.

Another approach I took was removing any calls to copy data into a new set. There was no reason we could not use the master data list to run calculations as long as we know which data range we were interested in.

**2. Would your approach continue to work if there were millions or billions of input values?**

Based on my understanding, because we're storing each data point as a 32-bit integer, we could probably handle a million, but holding a billion data points in memory would require at least 4GB. I would say that's not a reasonable requirement for any desktop application, another solution would need to be found. A possible approach would be to start throwing away data in the first and fourth quartiles, once we reached a set memory limit.

**3. Would your approach still be efficient if you needed to store the intermediate state between each IQM calculation in a data store? If not, how would you change it to meet this requirement?**

As it currently is, my solution would not support storing the intermediate state. I would need to expand my program to provide the list of data points belonging to the interquartile. Currently I don’t provide a way to capture this list of data points, I only provide the sample size and the IQM.