Assignment 2

Due: Oct 5, 2023 at 11:59PM

1 Assignment 2 Specification

1.1 Implementation

The skeleton code at `https://classroom.github.com/a/Q2gAR5Sv` is the start of a program to add floating point numbers in a way that reduces rounding error. Since floats take up a fixed amount of memory, adding two floats with very different magnitudes can lead to an approximate rather than exact solution – some or all of the smaller value is rounded off. This means that adding many floats together can lead to different amounts of rounding error depending on the order they are added together. To minimize rounding error, see the algorithm below.

Complete the program by implementing a faster version of **min2Scan** that implements a heap for its insert/extract operations. To complete the assignment, replace the **TODO** method stub with the below algorithm. You can (and should) write private helper methods in the **A2** class that are called by **heapAdd**. All inputs will be non-negative.

The skeleton code has three sample algorithms:

seqAdd Performs a simple sequential sum of the numbers in the order in which they were given. This can generate a lot of rounding error.

sortAdd Sorts the input from smallest to largest and then sums them sequentially. This generates less rounding error than **seqAdd**, but doesn't minimize it.

min2Scan Scans the set of input numbers for the smallest two numbers, adds them, and puts the result back in the set. Repeats this until all the numbers are summed. This minimizes rounding error.

Your algorithm, **heapAdd**, should use a heap to implement the technique that **min2Scan** uses:

```
heapAdd(A)
1: build heap from A
2: while heap.size > 1 do
3:     extract min from heap
4:     extract min from heap
5:     sum two mins
6:     insert sum into heap
7: return heap[0]
```

Using a heap will speed up the extract and insert operations while still minimizing rounding error.

When you change the size of the heap, consider using a separate variable to keep track of its current size, rather than relying on the array's size. **Do not** resize or copy the array for each insert and delete operation – that takes too many operations. Also, **do not** use any of Java's built-in Collections (e.g. PriorityQueue, ArrayList) in this assignment.

1.2 Testing

Full JUnit tests are provided in the **edu.wit.cs.comp2350.tests** package. You can run these tests to see if your code is performing correctly for some samples. Your assignment grade will be based entirely on these tests. In future assignments I will not provide all of the grading tests so you will have to be thoughtful with testing your code.

The value that **heapAdd** calculates should match the tests *exactly.* An answer that looks close means that you've added all the values but not in the correct order. If tests are timing out, that means that you have a lot of extra operations that are slowing down your algorithm – most likely from unnecessary memory allocation.

A **ChartMaker** class is also included, which will create a chart of the runtimes of each algorithm with different input sizes. You can use this chart to verify that the time complexities of the algorithms are what you expect. Additionally, an **ErrorChartMaker** class in included, which creates a chart showing the roundoff error of each adding algorithm.

2 Grading

`testSmall`: 65% with heap operations implemented

Other tests: 35%