Program Structures and Algorithms
Spring 2024

NAME: Aneesh Arunjunai Saravanan
NUID: 002675639
GITHUB LINK: https://github.com/aneesharunjunai/INFO6205

Task: Assignment 2 – Three Sum

Solve 3-SUM using the Quadrithmic, Quadratic, and (bonus point) quadraticWithCalipers approaches, as shown in skeleton code in the repository. There are hints at the end of Lesson 2.5 Entropy.

There are also hints in the comments of the existing code. There are a number of unit tests which you should be able to run successfully.

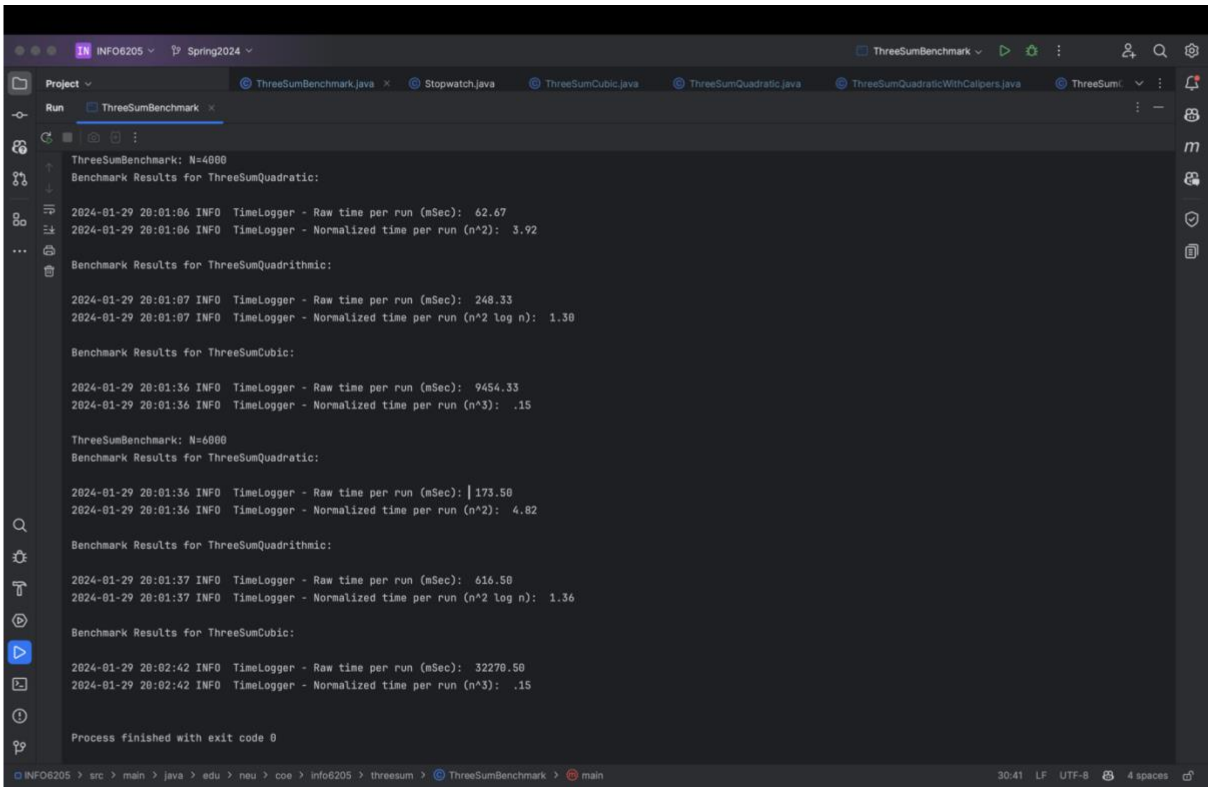
Submit (in your own repository--see instructions elsewhere--include the source code and the unit tests of course):

(a) Evidence (screenshot) of your unit tests running (try to show the actual unit test code as well as the green strip);

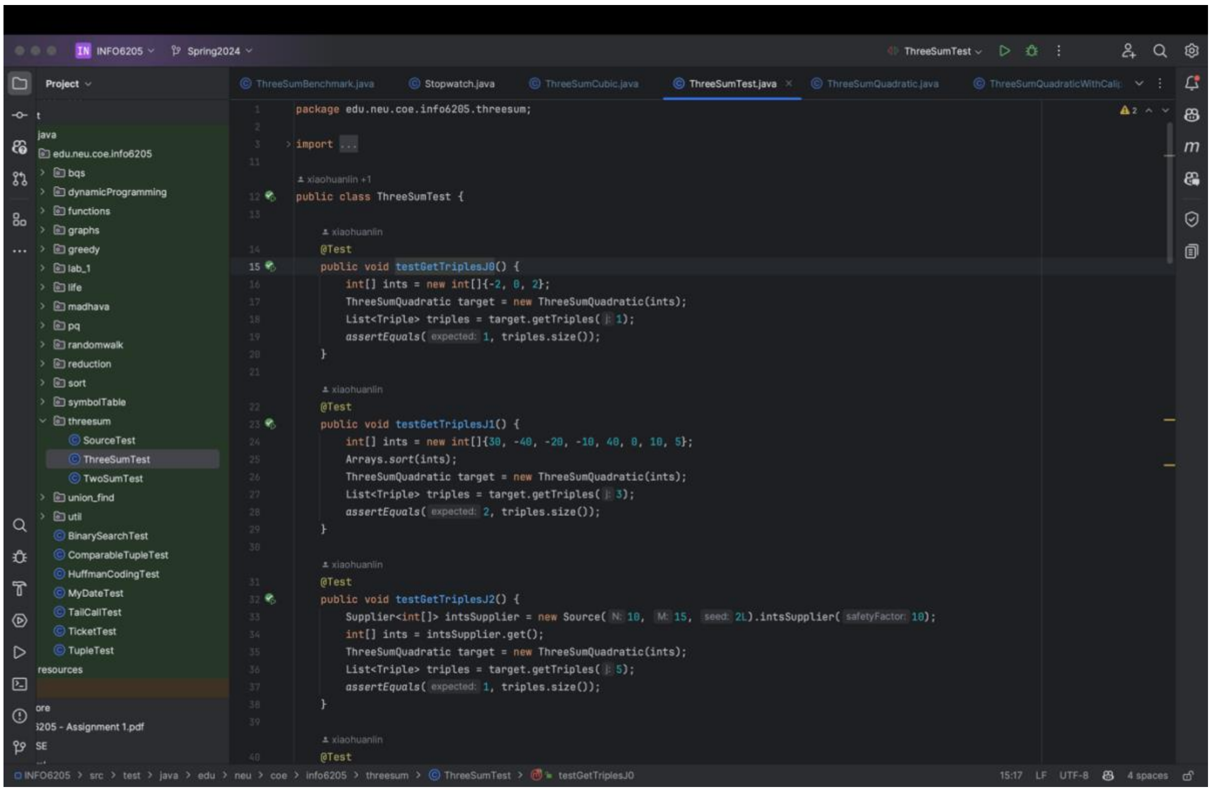
(b) A spreadsheet showing your timing observations--using the doubling method for at least five values of N--for each of the algorithms (include cubic); Timing should be performed either with an actual stopwatch (e.g. your iPhone) or using the Stopwatch class in the repository.

(c) Your brief explanation of why the quadratic method(s) work.

Console Output:

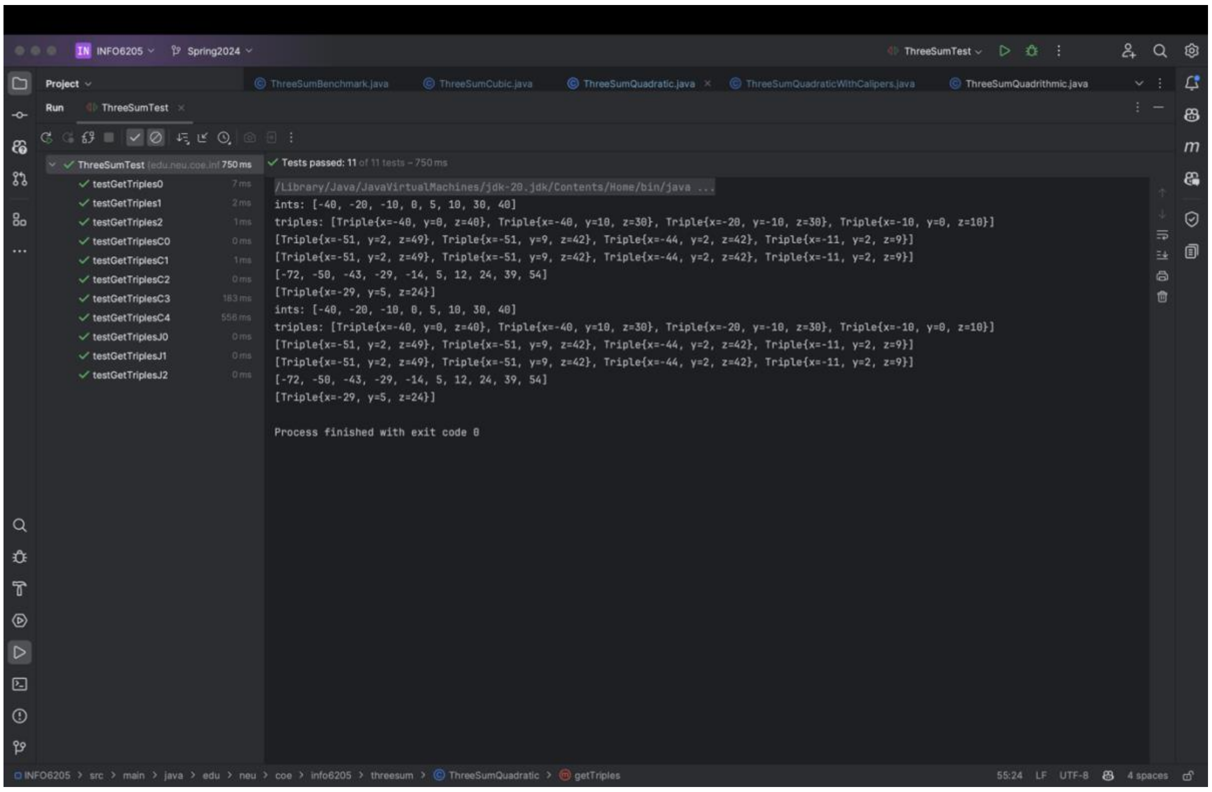


```
ThreeSumBenchmark: N=4000
Benchmark Results for ThreeSumQuadratic:

2024-01-29 20:01:06 INFO  TimeLogger - Raw time per run (mSec):  62.67
2024-01-29 20:01:06 INFO  TimeLogger - Normalized time per run (n^2):  3.92

Benchmark Results for ThreeSumQuadrithmic:

2024-01-29 20:01:07 INFO  TimeLogger - Raw time per run (mSec):  248.33
2024-01-29 20:01:07 INFO  TimeLogger - Normalized time per run (n^2 log n):  1.30

Benchmark Results for ThreeSumCubic:

2024-01-29 20:01:36 INFO  TimeLogger - Raw time per run (mSec):  9454.33
2024-01-29 20:01:36 INFO  TimeLogger - Normalized time per run (n^3):  .15

ThreeSumBenchmark: N=6000
Benchmark Results for ThreeSumQuadratic:

2024-01-29 20:01:36 INFO  TimeLogger - Raw time per run (mSec):  173.50
2024-01-29 20:01:36 INFO  TimeLogger - Normalized time per run (n^2):  4.82

Benchmark Results for ThreeSumQuadrithmic:

2024-01-29 20:01:37 INFO  TimeLogger - Raw time per run (mSec):  616.50
2024-01-29 20:01:37 INFO  TimeLogger - Normalized time per run (n^2 log n):  1.36

Benchmark Results for ThreeSumCubic:

2024-01-29 20:02:42 INFO  TimeLogger - Raw time per run (mSec):  32270.50
2024-01-29 20:02:42 INFO  TimeLogger - Normalized time per run (n^3):  .15


Process finished with exit code 0
```



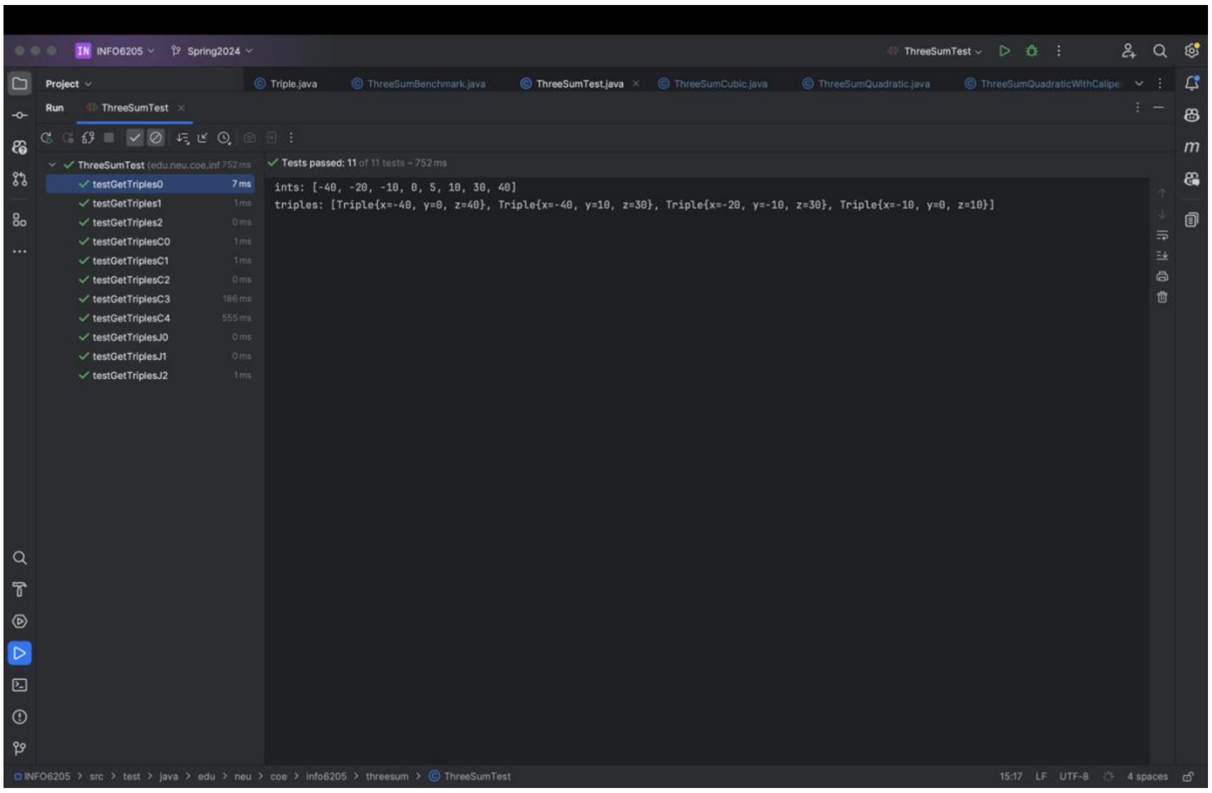
```java
package edu.neu.coe.info6205.threesum;

import ...

public class ThreeSumTest {

    @Test
    public void testGetTriplesJ0() {
        int[] ints = new int[]{-2, 0, 2};
        ThreeSumQuadratic target = new ThreeSumQuadratic(ints);
        List<Triple> triples = target.getTriples( 1);
        assertEquals( expected: 1, triples.size());
    }

    @Test
    public void testGetTriplesJ1() {
        int[] ints = new int[]{30, -40, -20, -10, 40, 0, 10, 5};
        Arrays.sort(ints);
        ThreeSumQuadratic target = new ThreeSumQuadratic(ints);
        List<Triple> triples = target.getTriples( 3);
        assertEquals( expected: 2, triples.size());
    }

    @Test
    public void testGetTriplesJ2() {
        Supplier<int[]> intsSupplier = new Source( N: 10,  M: 15,  seed: 2L).intsSupplier( safetyFactor: 10);
        int[] ints = intsSupplier.get();
        ThreeSumQuadratic target = new ThreeSumQuadratic(ints);
        List<Triple> triples = target.getTriples( 5);
        assertEquals( expected: 1, triples.size());
    }

    @Test
```

Unit Test Screenshots:



```
ThreeSumTest (edu.neu.coe.inf 750 ms
  testGetTriples0    7 ms
  testGetTriples1    2 ms
  testGetTriples2    1 ms
  testGetTriplesC0   0 ms
  testGetTriplesC1   1 ms
  testGetTriplesC2   0 ms
  testGetTriplesC3   183 ms
  testGetTriplesC4   556 ms
  testGetTriplesJ0   0 ms
  testGetTriplesJ1   0 ms
  testGetTriplesJ2   0 ms
```

```
Tests passed: 11 of 11 tests – 750 ms
/Library/Java/JavaVirtualMachines/jdk-20.jdk/Contents/Home/bin/java ...
ints: [-40, -20, -10, 0, 5, 10, 30, 40]
triples: [Triple{x=-40, y=0, z=40}, Triple{x=-40, y=10, z=30}, Triple{x=-20, y=-10, z=30}, Triple{x=-10, y=0, z=10}]
[Triple{x=-51, y=2, z=49}, Triple{x=-51, y=9, z=42}, Triple{x=-44, y=2, z=42}, Triple{x=-11, y=2, z=9}]
[Triple{x=-51, y=2, z=49}, Triple{x=-51, y=9, z=42}, Triple{x=-44, y=2, z=42}, Triple{x=-11, y=2, z=9}]
[-72, -50, -43, -29, -14, 5, 12, 24, 39, 54]
[Triple{x=-29, y=5, z=24}]
ints: [-40, -20, -10, 0, 5, 10, 30, 40]
triples: [Triple{x=-40, y=0, z=40}, Triple{x=-40, y=10, z=30}, Triple{x=-20, y=-10, z=30}, Triple{x=-10, y=0, z=10}]
[Triple{x=-51, y=2, z=49}, Triple{x=-51, y=9, z=42}, Triple{x=-44, y=2, z=42}, Triple{x=-11, y=2, z=9}]
[Triple{x=-51, y=2, z=49}, Triple{x=-51, y=9, z=42}, Triple{x=-44, y=2, z=42}, Triple{x=-11, y=2, z=9}]
[-72, -50, -43, -29, -14, 5, 12, 24, 39, 54]
[Triple{x=-29, y=5, z=24}]

Process finished with exit code 0
```



```
ThreeSumTest (edu.neu.coe.inf 752 ms
  testGetTriples0    7 ms
  testGetTriples1    1 ms
  testGetTriples2    0 ms
  testGetTriplesC0   1 ms
  testGetTriplesC1   1 ms
  testGetTriplesC2   0 ms
  testGetTriplesC3   186 ms
  testGetTriplesC4   555 ms
  testGetTriplesJ0   0 ms
  testGetTriplesJ1   0 ms
  testGetTriplesJ2   1 ms
```

```
Tests passed: 11 of 11 tests – 752 ms
ints: [-40, -20, -10, 0, 5, 10, 30, 40]
triples: [Triple{x=-40, y=0, z=40}, Triple{x=-40, y=10, z=30}, Triple{x=-20, y=-10, z=30}, Triple{x=-10, y=0, z=10}]
```

Why the quadratic methods work:

ThreeSumQuadratic:

The ThreeSumQuadratic algorithm adopts a straightforward approach to identify triplets within an array that sum to zero. It employs three pointers, denoted as i, j, and k. The outer loop revolves around the i pointer, while the nested loops (j and k) iterate over the remaining elements in the array. For each combination of i, j, and k, the algorithm checks if the sum of array[i] + array[j] + array[k] equals zero. The time complexity of this algorithm is O(N^2), where N represents the size of the array. Despite its simplicity, this method may encounter inefficiencies, particularly for larger arrays, due to its quadratic time complexity.

ThreeSumQuadrithmic:

Building upon the quadratic approach, the ThreeSumQuadrithmic algorithm aims to enhance efficiency by mitigating duplicate triplet combinations. It initiates the process by sorting the array, a step that facilitates the identification and removal of duplicate combinations. The algorithm deploys three pointers—i, low, and high—to traverse the sorted array. For each i, it initializes low and high and checks if the sum of array[i] + array[low] + array[high] equals zero. In cases where the sum is zero, it adjusts low and high accordingly, avoiding duplicate combinations. Although the algorithm maintains a quadratic time complexity of O(N^2), the sorting step contributes to a reduction in constant factors, leading to improved overall performance, especially when dealing with larger datasets.

Summary:

In summary, both quadratic methods share a fundamental strategy of exhaustively exploring all possible combinations of three numbers within an array to identify those whose sum is zero. The ThreeSumQuadrithmic algorithm optimizes efficiency by incorporating sorting to eliminate duplicates, resulting in enhanced performance while still operating within a quadratic time complexity.