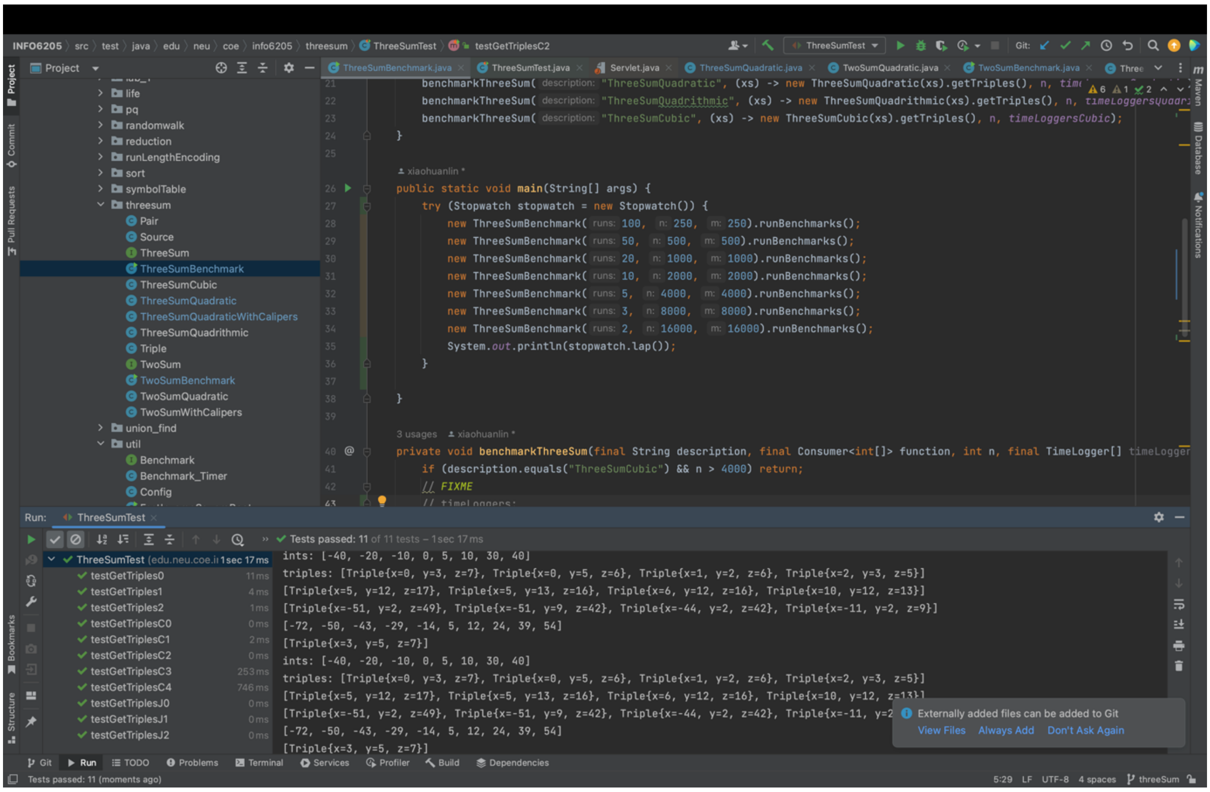
Program Structures and Algorithms
Spring 2023(SEC – 8)

NAME: Daiming Yang
NUID: 002771605

Task:

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.

a) Unit Test Screenshots:



b) Spreadsheet

ThreeSumBenchmark: N=250
ThreeSumQuadratic running: 100 runs cost 108.0ms
2023-01-28 19:00:35 INFO TimeLogger - Raw time per run (mSec): 1.08
ThreeSumQuadrithmic running: 100 runs cost 128.0ms

2023-01-28 19:00:36 INFO TimeLogger - Raw time per run (mSec): 1.28
ThreeSumCubic running: 100 runs cost 478.0ms
2023-01-28 19:00:36 INFO TimeLogger - Raw time per run (mSec): 4.78

ThreeSumBenchmark: N=500
ThreeSumQuadratic running: 50 runs cost 96.0ms
2023-01-28 19:00:36 INFO TimeLogger - Raw time per run (mSec): 1.92
ThreeSumQuadrithmic running: 50 runs cost 157.0ms
2023-01-28 19:00:36 INFO TimeLogger - Raw time per run (mSec): 3.14
ThreeSumCubic running: 50 runs cost 1504.0ms
2023-01-28 19:00:38 INFO TimeLogger - Raw time per run (mSec): 30.08

ThreeSumBenchmark: N=1000
ThreeSumQuadratic running: 20 runs cost 123.0ms
2023-01-28 19:00:38 INFO TimeLogger - Raw time per run (mSec): 6.15
ThreeSumQuadrithmic running: 20 runs cost 320.0ms
2023-01-28 19:00:38 INFO TimeLogger - Raw time per run (mSec): 16.00
ThreeSumCubic running: 20 runs cost 4681.0ms
2023-01-28 19:00:43 INFO TimeLogger - Raw time per run (mSec): 234.05

ThreeSumBenchmark: N=2000
ThreeSumQuadratic running: 10 runs cost 207.0ms
2023-01-28 19:00:43 INFO TimeLogger - Raw time per run (mSec): 20.70
ThreeSumQuadrithmic running: 10 runs cost 917.0ms
2023-01-28 19:00:44 INFO TimeLogger - Raw time per run (mSec): 91.70
ThreeSumCubic running: 10 runs cost 18206.0ms
2023-01-28 19:01:02 INFO TimeLogger - Raw time per run (mSec): 1820.60

ThreeSumBenchmark: N=4000
ThreeSumQuadratic running: 5 runs cost 409.0ms
2023-01-28 19:01:03 INFO TimeLogger - Raw time per run (mSec): 81.80
ThreeSumQuadrithmic running: 5 runs cost 1978.0ms
2023-01-28 19:01:05 INFO TimeLogger - Raw time per run (mSec): 395.60
ThreeSumCubic running: 5 runs cost 73456.0ms
2023-01-28 19:02:18 INFO TimeLogger - Raw time per run (mSec): 14691.20

ThreeSumBenchmark: N=8000
ThreeSumQuadratic running: 3 runs cost 1095.0ms
2023-01-28 19:02:19 INFO TimeLogger - Raw time per run (mSec): 365.00
ThreeSumQuadrithmic running: 3 runs cost 5224.0ms
2023-01-28 19:02:24 INFO TimeLogger - Raw time per run (mSec): 1741.33

ThreeSumBenchmark: N=16000
ThreeSumQuadratic running: 2 runs cost 2733.0ms
2023-01-28 19:02:27 INFO TimeLogger - Raw time per run (mSec): 1366.50
ThreeSumQuadrithmic running: 2 runs cost 14688.0ms
2023-01-28 19:02:42 INFO TimeLogger - Raw time per run (mSec): 7344.00

c) why the quadratic method(s) work

High Level:

To find 3 numbers in an sorted array, forming a triple which the sum of the three numbers is zero, first pick a number in the array is needed, then pick the second and the third number separately from the left side and the right side of the first number, checking if their sum is zero. In the explanation, the array is assumed as a sorted incrementing array.

When picking the first number, traverse the whole array is needed, the time complexity for this step is $O(n)$. The second and the third number are picked at the same time, in a traversal using two pointers. The time complexity for this step is $O(n)$. So the total time complexity for this algorithm is $O(n^2)$

When picking the second and the third number, there are three cases to be discussed.

a. The sum is greater than zero. The left pointer to pick the second number should be moved to the left (smaller side) to decrease the sum until reaching the bound.
b. The sum is smaller than zero. Similar with the a. case, the right pointer to pick the third number should be moved to the right (larger side) to increase the sum until reaching the bound.
c. The sum equals to zero. Store the result, and move the left pointer and right pointer at the same time, since there still could have some valid pairs from the left side and the right side of the first number.

```java
public List<Triple> getTriples(int j) {
    List<Triple> triples = new ArrayList<>();
    if (j == 0 || j == length - 1) {
        return triples;
    }
    int i = j - 1;
    int k = j + 1;
    while (-1 < i && k < length) {       //reach the bound
        int threeSum = a[i] + a[j] + a[k]; //sum
        if (threeSum > 0) { //case a
            i--;
        } else if (threeSum < 0) { //case b
            k++;
        } else {          //case c, the sum equals to zero
            triples.add(new Triple(i, j, k));
            i--;
            k++;
        }
    }
    return triples;
}
```