Program Structures and Algorithms
Spring 2023(SEC – 8)

Assignment 2 (3-SUM)

NAME: Akash Bharadwaj Karthik
NUID: 002787011

Task:

Solve 3-SUM using the Quadrithmic, Quadratic and (bonus points) quadraticWithCalipers approaches, as shown in the skeleton code in the repository.

Relationship Conclusion:

Doubling the input size – 'N' – of the array results in the following run time relationships for each approach:

Cubic: $\log_2\left(\frac{T(2N)}{T(N)}\right) \approx 3$

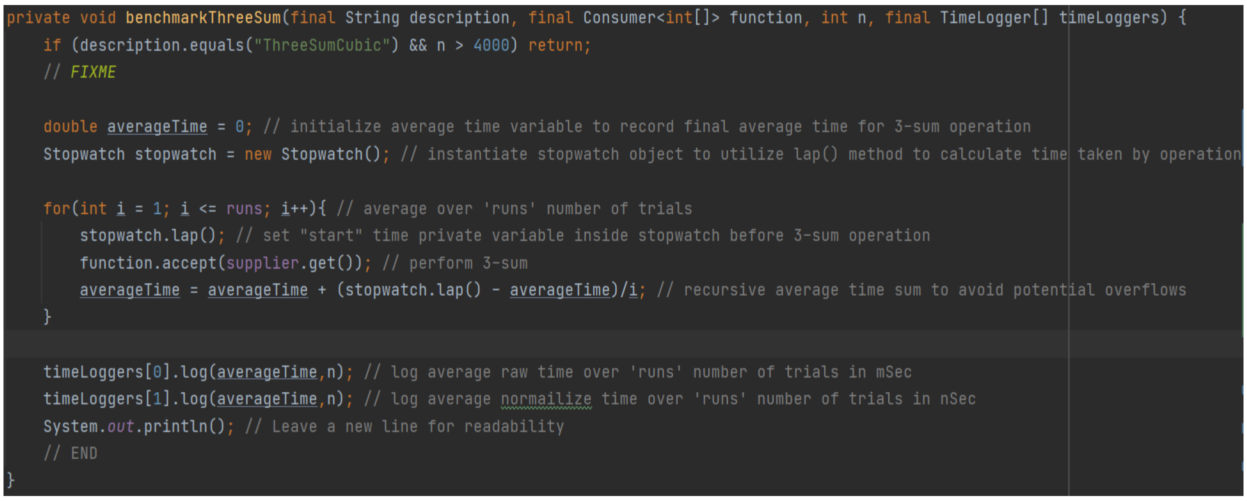
Quadrithmic: $\log_2\left(\frac{T(2N)}{T(N)}\right) \approx 2$ **(slightly greater)**

Quadratic: $\log_2\left(\frac{T(2N)}{T(N)}\right) \approx 2$

Evidence to support that conclusion:

Average running times are observed for Quadratic, Quadrithmic and Cubic approaches for values of N = 250,500,1000,2000,4000,8000,16000 over multiple runs, using the Stopwatch class in the repository.

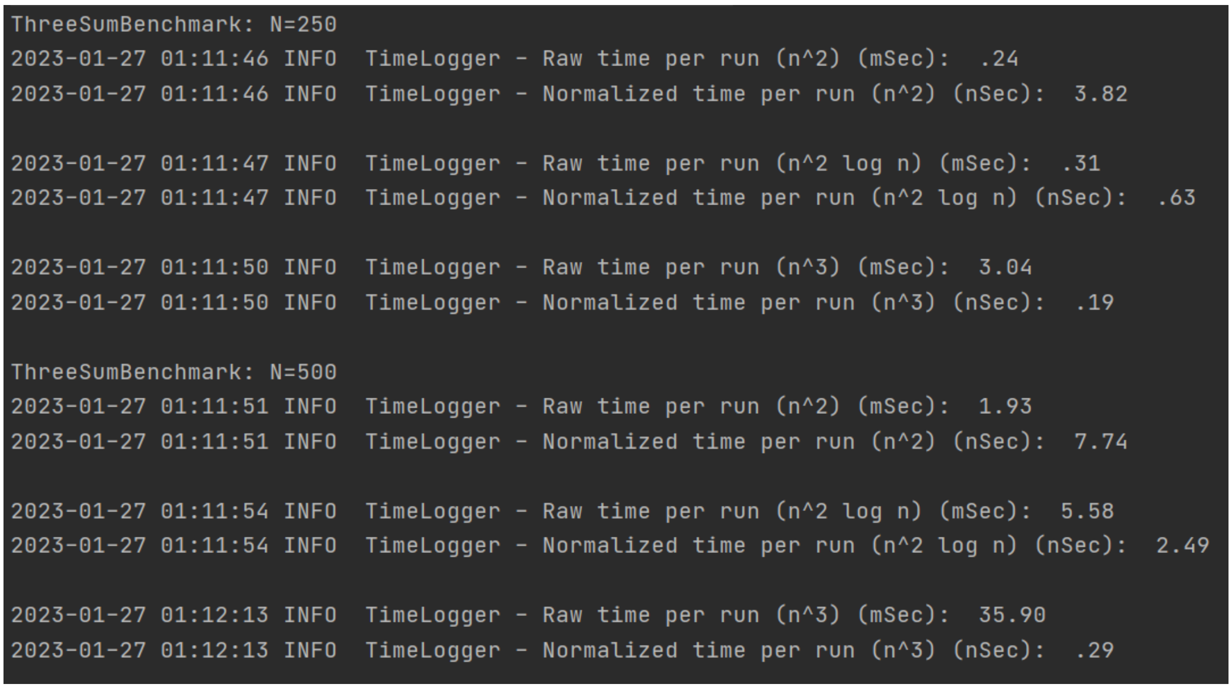
Code for benchmarking:



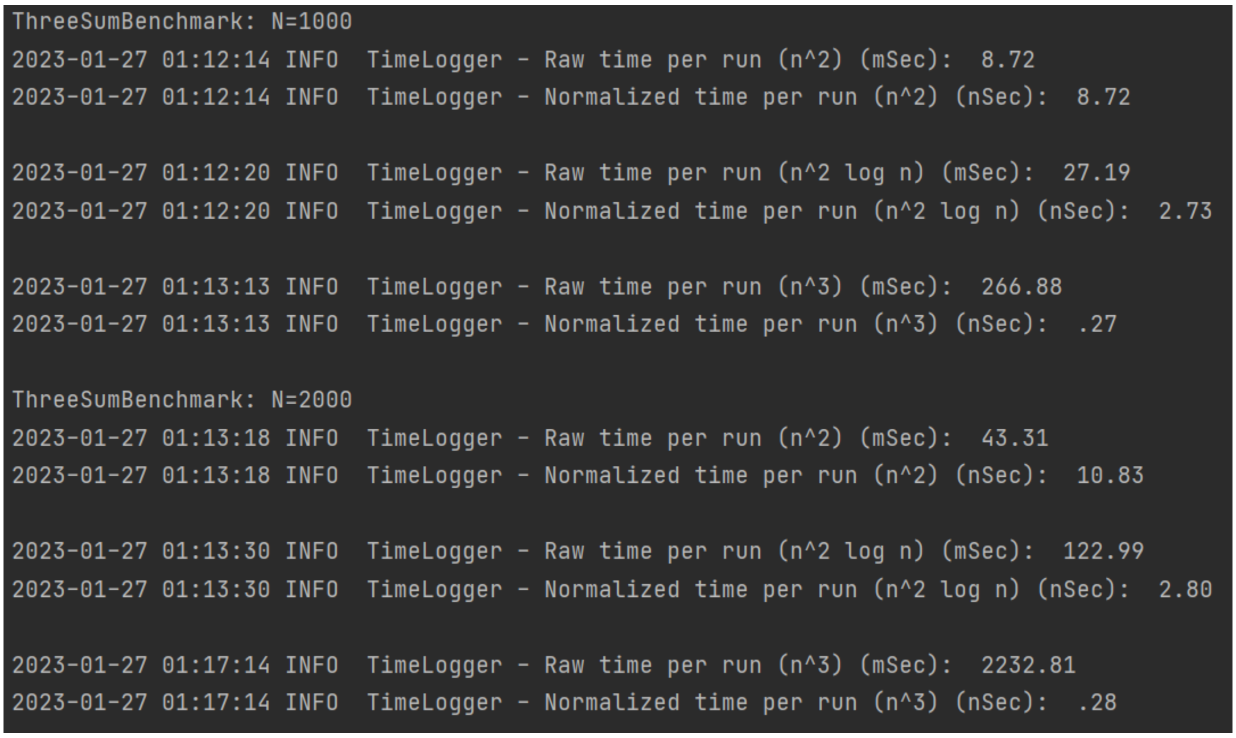
```java
private void benchmarkThreeSum(final String description, final Consumer<int[]> function, int n, final TimeLogger[] timeLoggers) {
    if (description.equals("ThreeSumCubic") && n > 4000) return;
    // FIXME

    double averageTime = 0; // initialize average time variable to record final average time for 3-sum operation
    Stopwatch stopwatch = new Stopwatch(); // instantiate stopwatch object to utilize lap() method to calculate time taken by operation

    for(int i = 1; i <= runs; i++){ // average over 'runs' number of trials
        stopwatch.lap(); // set "start" time private variable inside stopwatch before 3-sum operation
        function.accept(supplier.get()); // perform 3-sum
        averageTime = averageTime + (stopwatch.lap() - averageTime)/i; // recursive average time sum to avoid potential overflows
    }

    timeLoggers[0].log(averageTime,n); // log average raw time over 'runs' number of trials in mSec
    timeLoggers[1].log(averageTime,n); // log average normailize time over 'runs' number of trials in nSec
    System.out.println(); // Leave a new line for readability
    // END
}
```

Note: For the cubic approach, running times are only observed for N <= 4000 as the running times increases exponentially after this value.

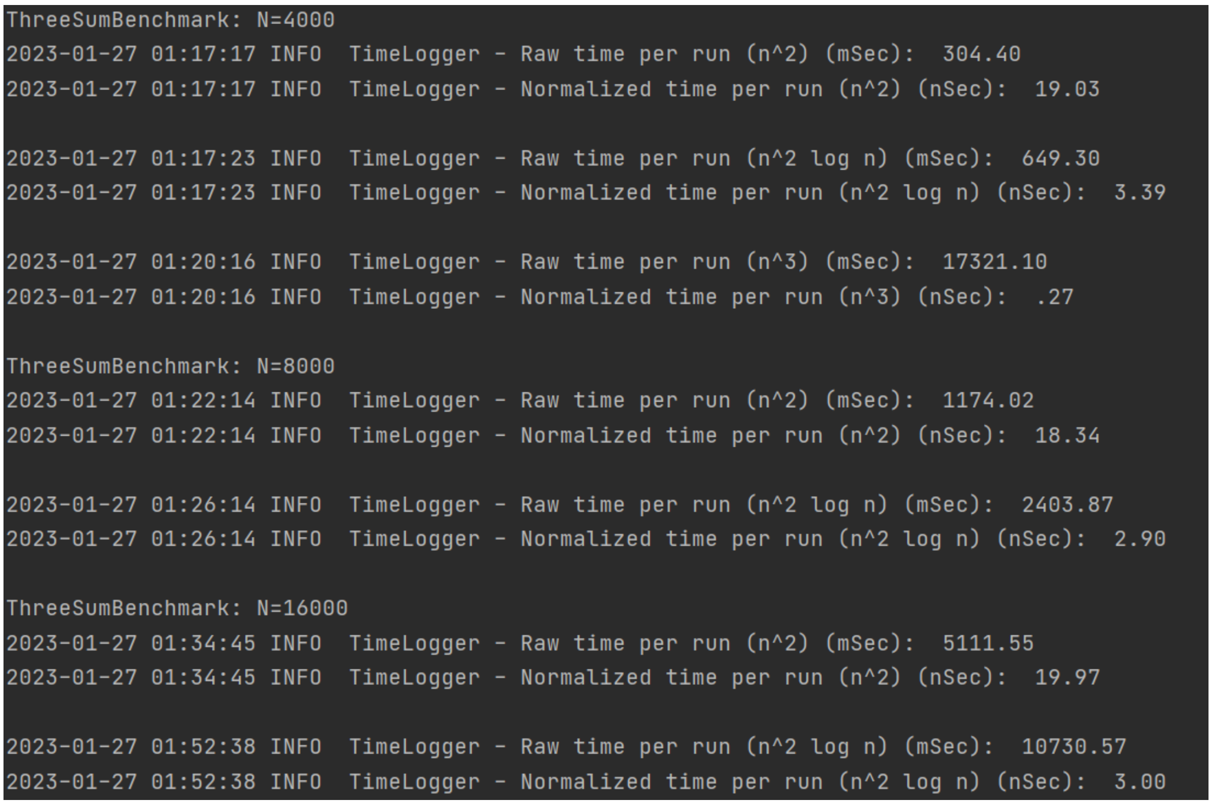
IDE simulation output:



```
ThreeSumBenchmark: N=250
2023-01-27 01:11:46 INFO  TimeLogger - Raw time per run (n^2) (mSec):  .24
2023-01-27 01:11:46 INFO  TimeLogger - Normalized time per run (n^2) (nSec):  3.82

2023-01-27 01:11:47 INFO  TimeLogger - Raw time per run (n^2 log n) (mSec):  .31
2023-01-27 01:11:47 INFO  TimeLogger - Normalized time per run (n^2 log n) (nSec):  .63

2023-01-27 01:11:50 INFO  TimeLogger - Raw time per run (n^3) (mSec):  3.04
2023-01-27 01:11:50 INFO  TimeLogger - Normalized time per run (n^3) (nSec):  .19

ThreeSumBenchmark: N=500
2023-01-27 01:11:51 INFO  TimeLogger - Raw time per run (n^2) (mSec):  1.93
2023-01-27 01:11:51 INFO  TimeLogger - Normalized time per run (n^2) (nSec):  7.74

2023-01-27 01:11:54 INFO  TimeLogger - Raw time per run (n^2 log n) (mSec):  5.58
2023-01-27 01:11:54 INFO  TimeLogger - Normalized time per run (n^2 log n) (nSec):  2.49

2023-01-27 01:12:13 INFO  TimeLogger - Raw time per run (n^3) (mSec):  35.90
2023-01-27 01:12:13 INFO  TimeLogger - Normalized time per run (n^3) (nSec):  .29
```



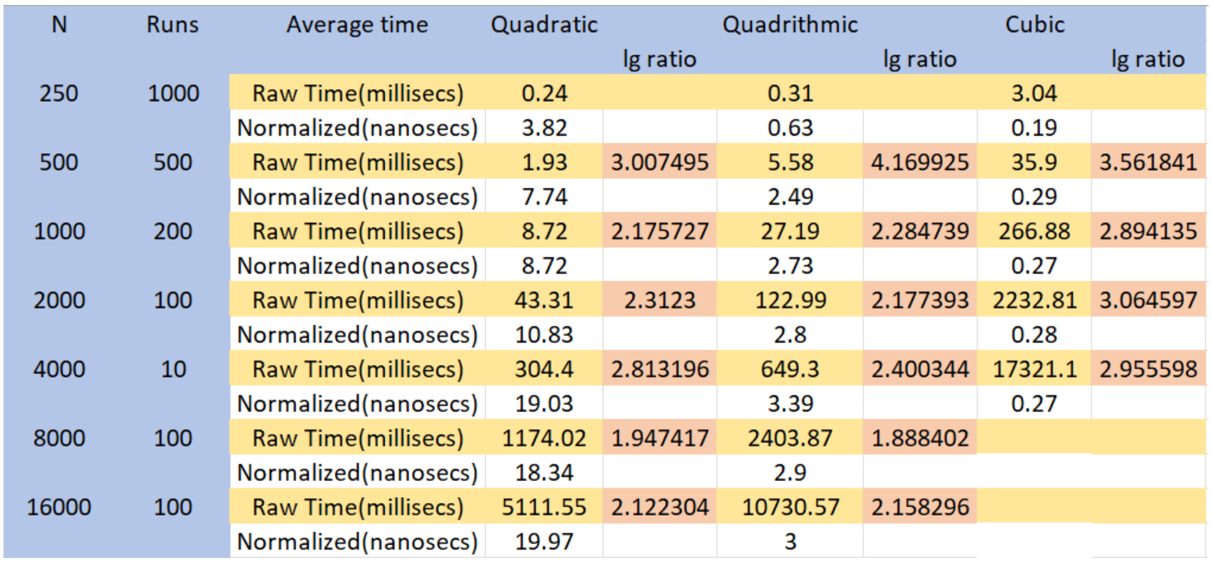
```
ThreeSumBenchmark: N=1000
2023-01-27 01:12:14 INFO  TimeLogger - Raw time per run (n^2) (mSec):  8.72
2023-01-27 01:12:14 INFO  TimeLogger - Normalized time per run (n^2) (nSec):  8.72

2023-01-27 01:12:20 INFO  TimeLogger - Raw time per run (n^2 log n) (mSec):  27.19
2023-01-27 01:12:20 INFO  TimeLogger - Normalized time per run (n^2 log n) (nSec):  2.73

2023-01-27 01:13:13 INFO  TimeLogger - Raw time per run (n^3) (mSec):  266.88
2023-01-27 01:13:13 INFO  TimeLogger - Normalized time per run (n^3) (nSec):  .27

ThreeSumBenchmark: N=2000
2023-01-27 01:13:18 INFO  TimeLogger - Raw time per run (n^2) (mSec):  43.31
2023-01-27 01:13:18 INFO  TimeLogger - Normalized time per run (n^2) (nSec):  10.83

2023-01-27 01:13:30 INFO  TimeLogger - Raw time per run (n^2 log n) (mSec):  122.99
2023-01-27 01:13:30 INFO  TimeLogger - Normalized time per run (n^2 log n) (nSec):  2.80

2023-01-27 01:17:14 INFO  TimeLogger - Raw time per run (n^3) (mSec):  2232.81
2023-01-27 01:17:14 INFO  TimeLogger - Normalized time per run (n^3) (nSec):  .28
```



```
ThreeSumBenchmark: N=4000
2023-01-27 01:17:17 INFO  TimeLogger - Raw time per run (n^2) (mSec):  304.40
2023-01-27 01:17:17 INFO  TimeLogger - Normalized time per run (n^2) (nSec):  19.03

2023-01-27 01:17:23 INFO  TimeLogger - Raw time per run (n^2 log n) (mSec):  649.30
2023-01-27 01:17:23 INFO  TimeLogger - Normalized time per run (n^2 log n) (nSec):  3.39

2023-01-27 01:20:16 INFO  TimeLogger - Raw time per run (n^3) (mSec):  17321.10
2023-01-27 01:20:16 INFO  TimeLogger - Normalized time per run (n^3) (nSec):  .27

ThreeSumBenchmark: N=8000
2023-01-27 01:22:14 INFO  TimeLogger - Raw time per run (n^2) (mSec):  1174.02
2023-01-27 01:22:14 INFO  TimeLogger - Normalized time per run (n^2) (nSec):  18.34

2023-01-27 01:26:14 INFO  TimeLogger - Raw time per run (n^2 log n) (mSec):  2403.87
2023-01-27 01:26:14 INFO  TimeLogger - Normalized time per run (n^2 log n) (nSec):  2.90

ThreeSumBenchmark: N=16000
2023-01-27 01:34:45 INFO  TimeLogger - Raw time per run (n^2) (mSec):  5111.55
2023-01-27 01:34:45 INFO  TimeLogger - Normalized time per run (n^2) (nSec):  19.97

2023-01-27 01:52:38 INFO  TimeLogger - Raw time per run (n^2 log n) (mSec):  10730.57
2023-01-27 01:52:38 INFO  TimeLogger - Normalized time per run (n^2 log n) (nSec):  3.00
```

The output from the IDE simulations is displayed in the spreadsheet below:

Spreadsheet Representation:



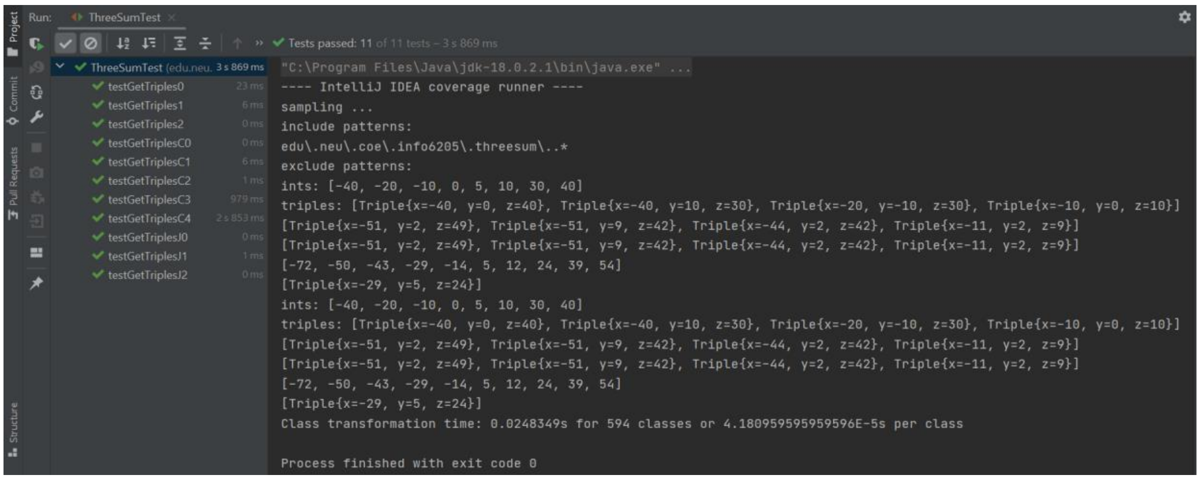
N	Runs	Average time	Quadratic		Quadrithmic		Cubic	
				lg ratio		lg ratio		lg ratio
250	1000	Raw Time(millisecs)	0.24		0.31		3.04	
		Normalized(nanosecs)	3.82		0.63		0.19	
500	500	Raw Time(millisecs)	1.93	3.007495	5.58	4.169925	35.9	3.561841
		Normalized(nanosecs)	7.74		2.49		0.29	
1000	200	Raw Time(millisecs)	8.72	2.175727	27.19	2.284739	266.88	2.894135
		Normalized(nanosecs)	8.72		2.73		0.27	
2000	100	Raw Time(millisecs)	43.31	2.3123	122.99	2.177393	2232.81	3.064597
		Normalized(nanosecs)	10.83		2.8		0.28	
4000	10	Raw Time(millisecs)	304.4	2.813196	649.3	2.400344	17321.1	2.955598
		Normalized(nanosecs)	19.03		3.39		0.27	
8000	100	Raw Time(millisecs)	1174.02	1.947417	2403.87	1.888402		
		Normalized(nanosecs)	18.34		2.9			
16000	100	Raw Time(millisecs)	5111.55	2.122304	10730.57	2.158296		
		Normalized(nanosecs)	19.97		3			

It can be observed from the spreadsheet that:

- lg ratio for the Quadratic approach converges towards 2 for high values of N
- lg ratio for the Quadrithmic approach converges slightly over 2 for high values of N
- lg ratio for the Cubic approach converges towards 3 for high values of N

Unit Test Results and Code Coverage over newly implemented code body:

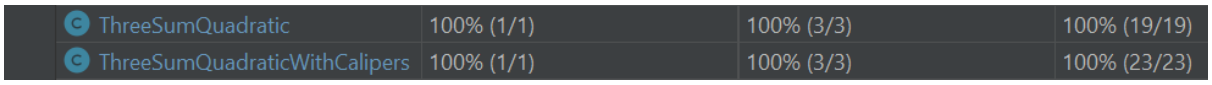
All Tests passing:



```
Tests passed: 11 of 11 tests – 3 s 869 ms

ThreeSumTest (edu.neu.   3 s 869 ms
    testGetTriples0          23 ms
    testGetTriples1           6 ms
    testGetTriples2           0 ms
    testGetTriplesC0          0 ms
    testGetTriplesC1          6 ms
    testGetTriplesC2          1 ms
    testGetTriplesC3        979 ms
    testGetTriplesC4   2 s 853 ms
    testGetTriplesJ0          0 ms
    testGetTriplesJ1          1 ms
    testGetTriplesJ2          0 ms
```

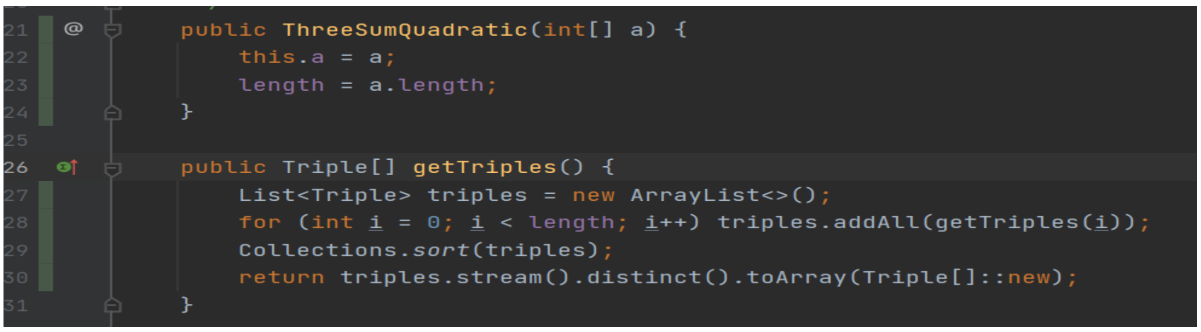
```
"C:\Program Files\Java\jdk-18.0.2.1\bin\java.exe" ...
---- IntelliJ IDEA coverage runner ----
sampling ...
include patterns:
edu\.neu\.coe\.info6205\.threesum\..*
exclude patterns:
ints: [-40, -20, -10, 0, 5, 10, 30, 40]
triples: [Triple{x=-40, y=0, z=40}, Triple{x=-40, y=10, z=30}, Triple{x=-20, 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}]
Class transformation time: 0.0248349s for 594 classes or 4.180959595959596E-5s per class

Process finished with exit code 0
```

Coverage for newly implemented methods for this assignment (quadratic and quadratic with calipers):

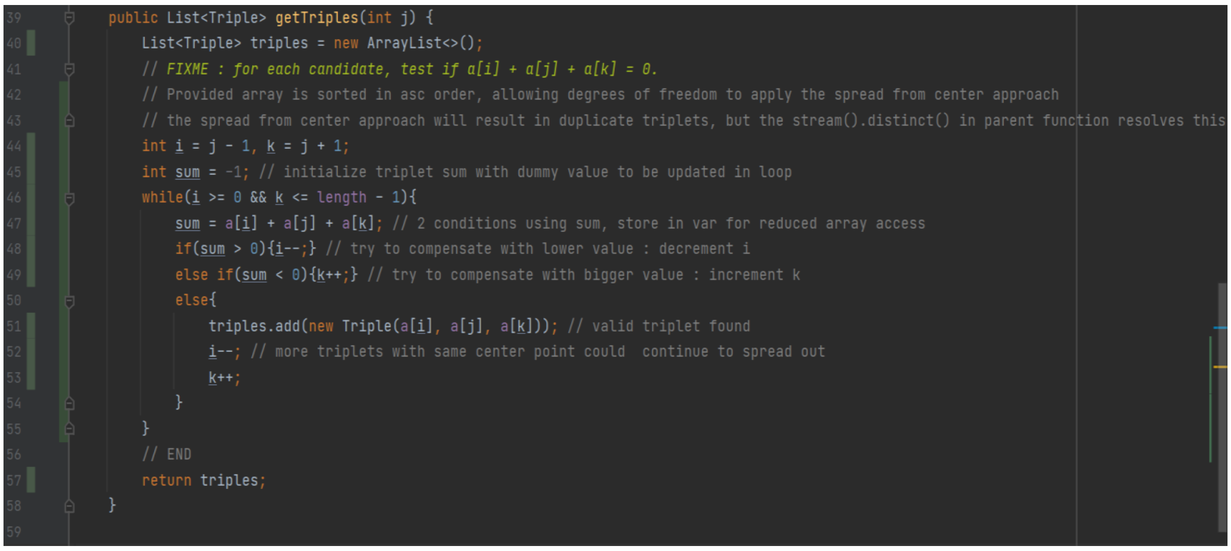


ThreeSumQuadratic	100% (1/1)	100% (3/3)	100% (19/19)
ThreeSumQuadraticWithCalipers	100% (1/1)	100% (3/3)	100% (23/23)

Quadratic Code with Green strips:

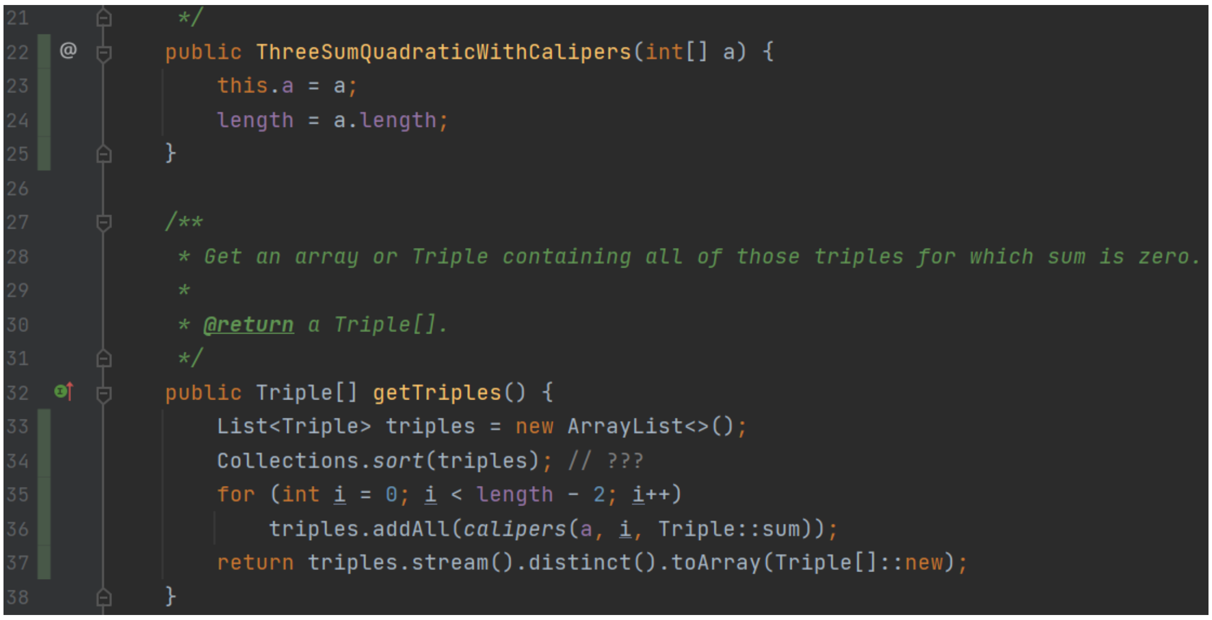


```java
    public ThreeSumQuadratic(int[] a) {
        this.a = a;
        length = a.length;
    }

    public Triple[] getTriples() {
        List<Triple> triples = new ArrayList<>();
        for (int i = 0; i < length; i++) triples.addAll(getTriples(i));
        Collections.sort(triples);
        return triples.stream().distinct().toArray(Triple[]::new);
    }
```

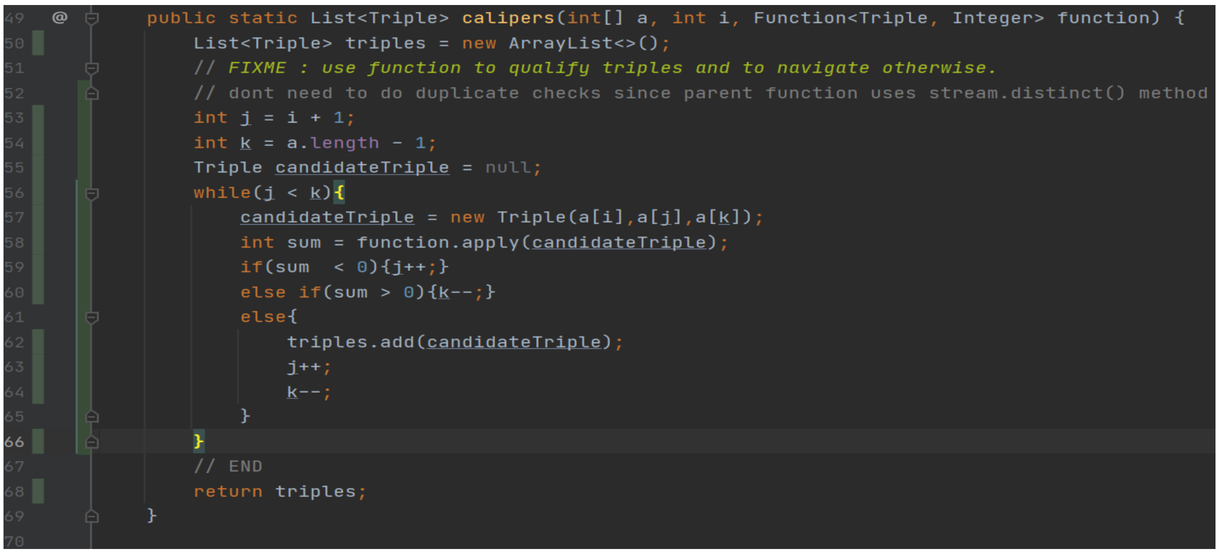


```java
    public List<Triple> getTriples(int j) {
        List<Triple> triples = new ArrayList<>();
        // FIXME : for each candidate, test if a[i] + a[j] + a[k] = 0.
        // Provided array is sorted in asc order, allowing degrees of freedom to apply the spread from center approach
        // the spread from center approach will result in duplicate triplets, but the stream().distinct() in parent function resolves this
        int i = j - 1, k = j + 1;
        int sum = -1; // initialize triplet sum with dummy value to be updated in loop
        while(i >= 0 && k <= length - 1){
            sum = a[i] + a[j] + a[k]; // 2 conditions using sum, store in var for reduced array access
            if(sum > 0){i--;} // try to compensate with lower value : decrement i
            else if(sum < 0){k++;} // try to compensate with bigger value : increment k
            else{
                triples.add(new Triple(a[i], a[j], a[k])); // valid triplet found
                i--; // more triplets with same center point could  continue to spread out
                k++;
            }
        }
        // END
        return triples;
    }
```

QuadraticWithCalipers Code with Green strips:



```
     */
    public ThreeSumQuadraticWithCalipers(int[] a) {
        this.a = a;
        length = a.length;
    }

    /**
     * Get an array or Triple containing all of those triples for which sum is zero.
     *
     * @return a Triple[].
     */
    public Triple[] getTriples() {
        List<Triple> triples = new ArrayList<>();
        Collections.sort(triples); // ???
        for (int i = 0; i < length - 2; i++)
            triples.addAll(calipers(a, i, Triple::sum));
        return triples.stream().distinct().toArray(Triple[]::new);
    }
```



```
    public static List<Triple> calipers(int[] a, int i, Function<Triple, Integer> function) {
        List<Triple> triples = new ArrayList<>();
        // FIXME : use function to qualify triples and to navigate otherwise.
        // dont need to do duplicate checks since parent function uses stream.distinct() method
        int j = i + 1;
        int k = a.length - 1;
        Triple candidateTriple = null;
        while(j < k){
            candidateTriple = new Triple(a[i],a[j],a[k]);
            int sum = function.apply(candidateTriple);
            if(sum  < 0){j++;}
            else if(sum > 0){k--;}
            else{
                triples.add(candidateTriple);
                j++;
                k--;
            }
        }
        // END
        return triples;
    }
```

<u>Explanation for working of quadratic method:</u>

Quadratic method (Spread from centre approach):

Every element in the array is initially considered as the middle element of a potential triple candidate, with its immediate left neighbour as the left triplet and its immediate right neighbour as the right triplet.
Since the array is sorted, there is an invariant in place wherein decrementing left index for a new left triplet reduces the overall sum, and incrementing the right index for a new right triplet increases overall triplet sum.

While the target sum is not equal to the current triplet sum and while the left and right triplet indices are within bounds of the array, the left triplet index is decremented if the triplet sum exceeds the target sum, or the right triplet index is incremented if the triplet sum is lesser than target sum to compensate for the difference.
If the candidate triplet sum equals the target sum, the current triplet is included in the answer, and the left triplet index is decremented and right triplet index in incremented to account for more solutions with the same middle triplet.

Since this process potentially scans the entire array (O(n)) for every element considered as the middle triplet (O(n)), the overall complexity = O(n^2)
(Note: Duplicates are not accounted for in this algorithm, but are handled outside of it)

Quadratic method (calliper approach):
This is similar to the previous approach, but each element in the array is considered as the left element of a potential triplet candidate, and a similar invariant is created in the algorithm wherein two pointers can be incremented or decremented to compensate towards the target sum. Placing the right pointer at the maximum value which is at index (length – 1) and placing the middle pointer at the least value after left triplet which is at index (left + 1) creates this invariant.

While the target sum is not equal to the current triplet sum and while middle index < right index, the right triplet index is decremented if the triplet sum exceeds the target sum, or the middle triplet index is incremented if the triplet sum is lesser than target sum to compensate for the difference.
If the candidate triplet sum equals the target sum, the current triplet is included in the answer, and the right triplet index is decremented and middle triplet index in incremented to account for more solutions with the same left triplet.

Since this process potentially scans the entire array (O(n)) for every element considered as the left triplet (O(n)), the overall complexity = O (n^2)

(Note: Duplicates are not accounted for in this algorithm, but are handled outside of it)