

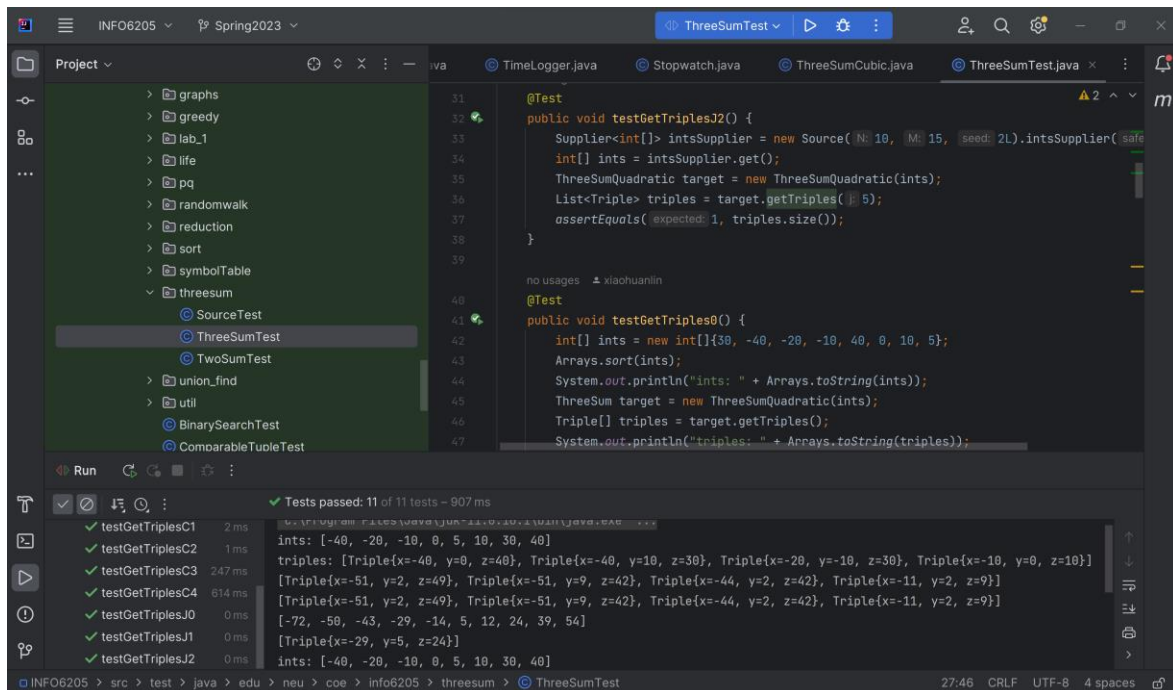
Program Structures and Algorithms

Spring 2023(SEC – 8)

NAME: Nandikonda Srikanth
NUID: 002737724

Task: Assignment 2 (Random Walk)

a) Evidence of Unit test cases



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

@Test
public void testGetTriples0() {
    int[] ints = new int[]{30, -40, -20, -10, 40, 0, 10, 5};
    Arrays.sort(ints);
    System.out.println("ints: " + Arrays.toString(ints));
    ThreeSum target = new ThreeSumQuadratic(ints);
    Triple[] triples = target.getTriples();
    System.out.println("triples: " + Arrays.toString(triples));
}
```

Tests passed: 11 of 11 tests - 907 ms

testGetTriplesC1 2 ms
testGetTriplesC2 1 ms
testGetTriplesC3 247 ms
testGetTriplesC4 614 ms
testGetTriplesJ0 0 ms
testGetTriplesJ1 0 ms
testGetTriplesJ2 0 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}]
[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]

b) Spreadsheet showing the timing observation

Time/Method	Quadratic	Log ratio	Quadrithmic	Log ratio	Cubic	Log Ratio
250(Raw Time)	1.85		1.62		8.82	
Normalized	29.6		3.25		0.56	
500(Raw Time)	3.04	0.716546053	7.6	2.230005605	88.41	3.325359
Normalized	12.16		3.39		0.71	
1000(Raw Time)	10.37	1.770272665	35.21	2.211913904	618.08	2.805512
Normalized	10.37		3.53		0.62	
2000(Raw Time)	52.16	2.330527975	166.3	2.239731036	4630.02	2.905153
Normalized	13.04		3.79		0.64	
4000(Raw Time)	237.2	2.185088235	688	2.048620396	41379	3.159808
Normalized	14.82		3.59		0.65	
8000(Raw time)	1413.9	2.575504172	3725.4	2.436914867		
Normalized	22.09		4.49			
16000(Raw Time)	6450.8	2.189798001	16226.7	2.122902388		
Normalized	25.2		4.54			

Values in the above spread sheet is obtained by running the main method in ThreeSumBenchmark class.

```
Run
"C:\Program Files\Java\jdk-11.0.16.1\bin\java.exe" ...
ThreeSumBenchmark: N=250
2023-01-28 18:02:27 INFO TimeLogger - Raw time per run (mSec) Quadratic: 1.85
2023-01-28 18:02:27 INFO TimeLogger - Normalized time per run (n^2) Quadratic: 29.60
2023-01-28 18:02:27 INFO TimeLogger - Raw time per run (mSec) Quadrithmic: 1.62
2023-01-28 18:02:27 INFO TimeLogger - Normalized time per run (n^2 log n) Quadrithmic: 3.25
2023-01-28 18:02:28 INFO TimeLogger - Raw time per run (mSec) for Cubic: 8.82
2023-01-28 18:02:28 INFO TimeLogger - Normalized time per run (n^3) Cubic: .56
ThreeSumBenchmark: N=500
2023-01-28 18:02:28 INFO TimeLogger - Raw time per run (mSec) Quadratic: 3.04
2023-01-28 18:02:28 INFO TimeLogger - Normalized time per run (n^2) Quadratic: 12.16
2023-01-28 18:02:29 INFO TimeLogger - Raw time per run (mSec) Quadrithmic: 7.60
2023-01-28 18:02:29 INFO TimeLogger - Normalized time per run (n^2 log n) Quadrithmic: 3.39
2023-01-28 18:02:38 INFO TimeLogger - Raw time per run (mSec) for Cubic: 88.41
2023-01-28 18:02:38 INFO TimeLogger - Normalized time per run (n^3) Cubic: .71
ThreeSumBenchmark: N=1000
2023-01-28 18:02:39 INFO TimeLogger - Raw time per run (mSec) Quadratic: 10.37
2023-01-28 18:02:39 INFO TimeLogger - Normalized time per run (n^2) Quadratic: 10.37
2023-01-28 18:02:42 INFO TimeLogger - Raw time per run (mSec) Quadrithmic: 35.21
2023-01-28 18:02:42 INFO TimeLogger - Normalized time per run (n^2 log n) Quadrithmic: 3.53
2023-01-28 18:03:44 INFO TimeLogger - Raw time per run (mSec) for Cubic: 618.08
2023-01-28 18:03:44 INFO TimeLogger - Normalized time per run (n^3) Cubic: .62
ThreeSumBenchmark: N=2000
2023-01-28 18:03:47 INFO TimeLogger - Raw time per run (mSec) Quadratic: 52.16
2023-01-28 18:03:47 INFO TimeLogger - Normalized time per run (n^2) Quadratic: 13.04
2023-01-28 18:03:55 INFO TimeLogger - Raw time per run (mSec) Quadrithmic: 166.30
2023-01-28 18:03:55 INFO TimeLogger - Normalized time per run (n^2 log n) Quadrithmic: 3.79
2023-01-28 18:08:11 INFO TimeLogger - Raw time per run (mSec) for Cubic: 5126.24
2023-01-28 18:08:11 INFO TimeLogger - Normalized time per run (n^3) Cubic: .64
ThreeSumBenchmark: N=4000
2023-01-28 18:08:13 INFO TimeLogger - Raw time per run (mSec) Quadratic: 237.20
2023-01-28 18:08:13 INFO TimeLogger - Normalized time per run (n^2) Quadratic: 14.82
2023-01-28 18:08:16 INFO TimeLogger - Raw time per run (mSec) Quadrithmic: 688.00
2023-01-28 18:08:16 INFO TimeLogger - Normalized time per run (n^2 log n) Quadrithmic: 3.59
2023-01-28 18:11:43 INFO TimeLogger - Raw time per run (mSec) for Cubic: 41379.00
2023-01-28 18:11:43 INFO TimeLogger - Normalized time per run (n^3) Cubic: .65
ThreeSumBenchmark: N=8000
2023-01-28 18:11:57 INFO TimeLogger - Raw time per run (mSec) Quadratic: 1413.90
2023-01-28 18:11:57 INFO TimeLogger - Normalized time per run (n^2) Quadratic: 22.09
2023-01-28 18:12:34 INFO TimeLogger - Raw time per run (mSec) Quadrithmic: 3725.40
2023-01-28 18:12:34 INFO TimeLogger - Normalized time per run (n^2 log n) Quadrithmic: 4.49
ThreeSumBenchmark: N=16000
2023-01-28 18:13:39 INFO TimeLogger - Raw time per run (mSec) Quadratic: 6450.80
2023-01-28 18:13:39 INFO TimeLogger - Normalized time per run (n^2) Quadratic: 25.20
2023-01-28 18:16:21 INFO TimeLogger - Raw time per run (mSec) Quadrithmic: 16226.70
2023-01-28 18:16:21 INFO TimeLogger - Normalized time per run (n^2 log n) Quadrithmic: 4.54

Process finished with exit code 0
```

Timing is performed using the System time (currentMilliseconds method) and log ratio is calculated for the various outputs. Log ratio in the case of Quadratic and Quadrathamic is approximately equal to 2 and in the case of Cubic value is close to 3.

c) **Quadratic Method working.**

The Quadratic Method for finding triplets in an array that add up to zero is based on the observation that the input array is sorted. By taking each element in the array as the middle element (j) and looking for adjacent elements (i and k) such that $a[i] + a[k] = -a[j]$, the algorithm can efficiently find all the triplets that add up to zero. The key insight behind this method is that, by dividing the array into two subgroups, one with elements on the left of j that are always less than $a[j]$, and the other with elements on the right of j that are always greater than $a[j]$, the algorithm can quickly narrow down the possible values of i and k by either increasing or decreasing their values depending on the sum of $a[i]$ and $a[k]$ in relation to $-a[j]$. If we find i and k values such that sum of $a[i]$ and $a[k]$ is equal to $-a[j]$, we add these three values to the list as these three values form a triplet. If the sum is greater than $-a[j]$, we have to try and reduce the sum so we decrement the value of the i , as the values to the left of $a[i]$ are always less than $a[i]$ and then calculate the sum again. If the sum is less than $-a[j]$, we have to try and increase the sum so we increment the value of the k , as the values to the right of $a[k]$ are always greater than $a[k]$ and then calculate the sum again. We follow this approach till we don't have any elements to the left of $a[i]$ and to the right of $a[k]$. This allows the algorithm to avoid the need for a nested loop to find the values of i and k , which would have a quadratic time complexity, and instead achieve a linear time complexity by only iterating through the array once to find i and k .