Program Structures and Algorithms
Spring 2023 (SEC – 8)

NAME: Rishi Desai
NUID: 002751030

Task: Solving the 3-sum problem using the Quadratic, Quadrithmic, and Quadratic with calipers.

Link to report:
https://github.com/RishiDesai17/INFO6205/blob/Spring2023/assignments/assignment-2/Assignment%202.pdf

Link to code:
https://github.com/RishiDesai17/INFO6205/tree/Spring2023/src/main/java/edu/neu/coe/info6205/threesum

Relationship Conclusion:
As observed by the logarithm ratios shown in the table given in the next section, we can understand the values towards which the logarithm ratio converges towards in the 3-sum algorithms of different complexities.

Quadratic (n^2): The logarithm ratio converges towards being approximately around 2.
Quadrithmic ($n^2\log(n)$): The logarithm ratio converges towards being a number greater than 2 but lesser than 3.
Cubic (n^3): The logarithm ratio converges towards being approximately around 3.

Evidence to support that conclusion:

N		Quadratic	log ratio	Quadrithmic	log ratio	Cubic	log ratio
250	Raw time (millisec)	1.79		1.4		9.47	
	Normalized time (nanosec)	28.64		2.81		0.61	
500	Raw time (millisec)	5.2	1.53855204	6.24	2.1561192	60.96	2.68642657
	Normalized time (nanosec)	20.8		2.78		0.49	
1000	Raw time (millisec)	12.4	1.25375659	22.85	1.87257623	420.75	2.78702846

	Normalized time (nanosec)	12.4		2.29		0.42	
2000	Raw time (millisec)	54.4	2.13326653	128.6	2.49262457	2975.1	2.82190298
	Normalized time (nanosec)	13.6		2.93		0.37	
4000	Raw time (millisec)	524.4	3.26898913	919	2.83717422	20500.8	2.78467014
	Normalized time (nanosec)	32.77		4.8		0.32	
8000	Raw time (millisec)	1906.33	1.86205829	3335.67	1.8598398	-	-
	Normalized time (nanosec)	29.79		4.02		-	
16000	Raw time (millisec)	7707	2.01537151	15662	2.23121998	-	-
	Normalized time (nanosec)	30.11		4.38		-	

Why the quadratic method(s) work:

The quadratic and quadratic with calipers methods expect the array to be sorted. This means that the two pointers that are being used to find the other 2 elements which upon adding to the current element will sum to zero, can be easily traversed. If the sum is less than zero, it means the left pointer (middle pointer in calipers method), needs to be moved forward, since the array is sorted in ascending order. Moving that pointer in the forward direction will only increase the sum, which will bring us closer to zero. Similarly, if the sum is greater than zero, the right pointer needs to be moved in the backward direction, since this will reduce the sum and bring us closer to zero. If at any point the sum is found to be zero, we have found an eligible triplet. In that case, we move both the left and right pointers so that we can find more such pairs. We can make a small optimization that we can directly skip over repetitive set of elements. I have handled this using the do-while loop which means it will run at least once and keep going until an element different from the current one is obtained.

Code:

ThreeSumQuadratic.java

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

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

/**
 * Implementation of ThreeSum which follows the approach of dividing the solution-space into
 * N sub-spaces where each sub-space corresponds to a fixed value for the middle index of the three values.
 * Each sub-space is then solved by expanding the scope of the other two indices outwards from the starting point.
 * Since each sub-space can be solved in O(N) time, the overall complexity is O(N^2).
 * <p>
 * NOTE: The array provided in the constructor MUST be ordered.
 */
public class ThreeSumQuadratic implements ThreeSum {
    /**
     * Construct a ThreeSumQuadratic on a.
     * @param a a sorted array.
     */
    public ThreeSumQuadratic(int[] a) {
        this.a = a;
        length = a.length;
    }

    public Triple[] getTriples() {
        List<Triple> triples = new ArrayList<>();
        for (int i = 1; i < length - 1; i++) triples.addAll(getTriples(i));
        Collections.sort(triples);
        return triples.stream().distinct().toArray(Triple[]::new);
    }

    /**
     * Get a list of Triples such that the middle index is the given value j.
     *
     * @param j the index of the middle value.
     * @return a Triple such that
     */
    public List<Triple> getTriples(int j) {
        List<Triple> triples = new ArrayList<>();

        int leftIndex = 0;
        int midIndex = j;
        int rightIndex = a.length - 1;
```

```
        while (leftIndex < midIndex && midIndex < rightIndex) {
            Triple triple = new Triple(a[leftIndex], a[midIndex], a[rightIndex]);
            int sum = triple.sum();

            if (sum == 0) {
                    triples.add(triple);

                    do {
                            leftIndex += 1;
                    }
                    while(leftIndex < midIndex && a[leftIndex - 1] == a[leftIndex]);

                    do {
                            rightIndex -= 1;
                    }
                    while(rightIndex > midIndex && a[rightIndex] == a[rightIndex + 1]);
            }
            else if (sum < 0) {
                    do {
                            leftIndex += 1;
                    }
                    while(leftIndex < midIndex && a[leftIndex - 1] == a[leftIndex]);
            }
            else {
                    do {
                            rightIndex -= 1;
                    }
                    while(rightIndex > midIndex && a[rightIndex] == a[rightIndex + 1]);
            }
        }

        return triples;
    }

    private final int[] a;
    private final int length;
}
```

ThreeSumQuadraticWithCalipers.java

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

import java.util.ArrayList;
import java.util.Collections;
```

```
import java.util.List;
import java.util.function.Function;

/**
 * Implementation of ThreeSum which follows the approach of dividing the solution-space into
 * N sub-spaces where each sub-space corresponds to a fixed value for the middle index of the three values.
 * Each sub-space is then solved by expanding the scope of the other two indices outwards from the starting point.
 * Since each sub-space can be solved in O(N) time, the overall complexity is O(N^2).
 * <p>
 * The array provided in the constructor MUST be ordered.
 */
public class ThreeSumQuadraticWithCalipers implements ThreeSum {
    /**
     * Construct a ThreeSumQuadratic on a.
     *
     * @param a a sorted array.
     */
    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);
    }

    /**
     * Get a set of candidate Triples such that the first index is the given value i.
     * Any candidate triple is added to the result if it yields zero when passed into function.
     *
     * @param a        a sorted array of ints.
     * @param i        the index of the first element of resulting triples.
```

```
    * @param function a function which takes a triple and returns a value which will be compared with zero.
    * @return a List of Triples.
    */
   public static List<Triple> calipers(int[] a, int i, Function<Triple, Integer> function) {
      List<Triple> triples = new ArrayList<>();

      int leftIndex = i;
      int midIndex = i + 1;
      int rightIndex = a.length - 1;

      while (midIndex < rightIndex) {
         Triple triple = new Triple(a[leftIndex], a[midIndex], a[rightIndex]);
         int sum = triple.sum();

         if (sum == 0) {
                  triples.add(triple);

                  do {
                           midIndex += 1;
                  }
                  while(midIndex < rightIndex && a[midIndex - 1] == a[midIndex]);

                  do {
                           rightIndex -= 1;
                  }
                  while(rightIndex > midIndex && a[rightIndex] == a[rightIndex + 1]);
         }
         else if (sum < 0) {
                  do {
                           midIndex += 1;
                  }
                  while(midIndex < rightIndex && a[midIndex - 1] == a[midIndex]);
         }
         else {
                  do {
                           rightIndex -= 1;
                  }
                  while(rightIndex > midIndex && a[rightIndex] == a[rightIndex + 1]);
         }
      }

      return triples;
   }
```

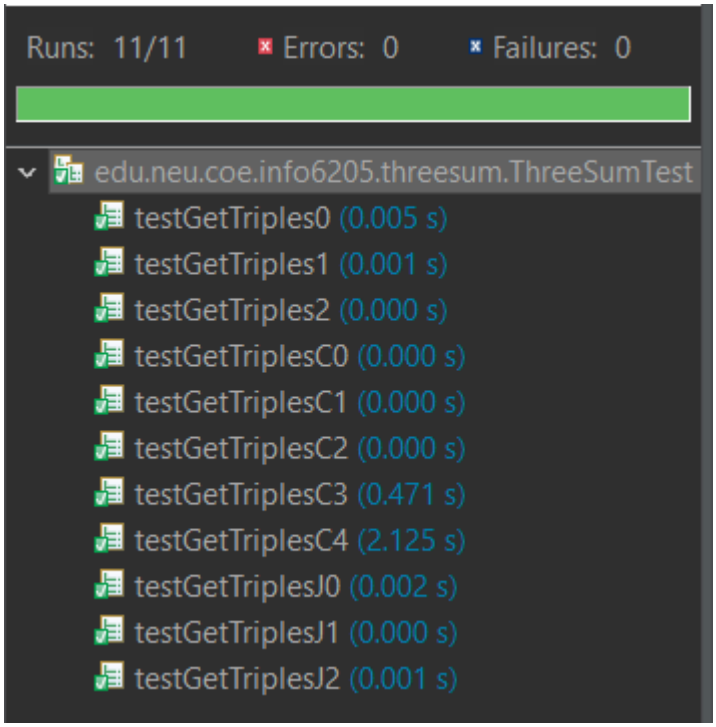
```
    private final int[] a;
    private final int length;
}
```

ThreeSumBenchmark.java

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

import edu.neu.coe.info6205.util.Benchmark_Timer;
import edu.neu.coe.info6205.util.Stopwatch;
import edu.neu.coe.info6205.util.TimeLogger;
import edu.neu.coe.info6205.util.Utilities;

import java.util.function.Consumer;
import java.util.function.Supplier;
import java.util.function.UnaryOperator;

public class ThreeSumBenchmark {
    public ThreeSumBenchmark(int runs, int n, int m) {
        this.runs = runs;
        this.supplier = new Source(n, m).intsSupplier(10);
        this.n = n;
    }

    public void runBenchmarks() {
        System.out.println("ThreeSumBenchmark: N=" + n);
        benchmarkThreeSum("ThreeSumQuadratic", (xs) -> new
ThreeSumQuadratic(xs).getTriples(), n, timeLoggersQuadratic);
        benchmarkThreeSum("ThreeSumQuadrithmic", (xs) -> new
ThreeSumQuadrithmic(xs).getTriples(), n, timeLoggersQuadrithmic);
        benchmarkThreeSum("ThreeSumCubic", (xs) -> new ThreeSumCubic(xs).getTriples(),
n, timeLoggersCubic);
    }

    public static void main(String[] args) {
        new ThreeSumBenchmark(100, 250, 250).runBenchmarks();
        new ThreeSumBenchmark(50, 500, 500).runBenchmarks();
        new ThreeSumBenchmark(20, 1000, 1000).runBenchmarks();
        new ThreeSumBenchmark(10, 2000, 2000).runBenchmarks();
        new ThreeSumBenchmark(5, 4000, 4000).runBenchmarks();
        new ThreeSumBenchmark(3, 8000, 8000).runBenchmarks();
        new ThreeSumBenchmark(2, 16000, 16000).runBenchmarks();
    }
```

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

        int[] ints = this.supplier.get();
        long totalTimeTaken = 0;

        for (int i = 0; i < this.runs; i++) {
                Stopwatch stopwatch = new Stopwatch();
        function.accept(ints);
        long timeTaken = stopwatch.lap();
        totalTimeTaken += timeTaken;
        }

      System.out.println(description + ":");
      timeLoggers[0].log(totalTimeTaken * 1.0 / this.runs, n);
      timeLoggers[1].log(totalTimeTaken * 1.0 / this.runs, n);
      System.out.println();
    }

    private final static TimeLogger[] timeLoggersCubic = {
        new TimeLogger("Raw time per run (mSec): ", (time, n) -> time),
        new TimeLogger("Normalized time per run (n^3): ", (time, n) -> time / n / n / n * 1e6)
    };
    private final static TimeLogger[] timeLoggersQuadrithmic = {
        new TimeLogger("Raw time per run (mSec): ", (time, n) -> time),
        new TimeLogger("Normalized time per run (n^2 log n): ", (time, n) -> time / n / n /
Utilities.lg(n) * 1e6)
    };
    private final static TimeLogger[] timeLoggersQuadratic = {
        new TimeLogger("Raw time per run (mSec): ", (time, n) -> time),
        new TimeLogger("Normalized time per run (n^2): ", (time, n) -> time / n / n * 1e6)
    };

    private final int runs;
    private final Supplier<int[]> supplier;
    private final int n;
}
```

Unit Test Screenshots and Code:




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

import org.junit.Ignore;
import org.junit.Test;

import java.util.Arrays;
import java.util.List;
import java.util.function.Supplier;

import static org.junit.Assert.assertEquals;

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(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(2, triples.size());
}

@Test
public void testGetTriplesJ2() {
    Supplier<int[]> intsSupplier = new Source(10, 15, 2L).intsSupplier(10);
    int[] ints = intsSupplier.get();
    ThreeSumQuadratic target = new ThreeSumQuadratic(ints);
    List<Triple> triples = target.getTriples(5);
    assertEquals(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));
    assertEquals(4, triples.length);
    assertEquals(4, new ThreeSumCubic(ints).getTriples().length);
}

@Test
public void testGetTriples1() {
    Supplier<int[]> intsSupplier = new Source(20, 20, 1L).intsSupplier(10);
    int[] ints = intsSupplier.get();
    ThreeSum target = new ThreeSumQuadratic(ints);
    Triple[] triples = target.getTriples();
    assertEquals(4, triples.length);
    System.out.println(Arrays.toString(triples));
    Triple[] triples2 = new ThreeSumCubic(ints).getTriples();
    System.out.println(Arrays.toString(triples2));
    assertEquals(4, triples2.length);
}

@Test
public void testGetTriples2() {
    Supplier<int[]> intsSupplier = new Source(10, 15, 3L).intsSupplier(10);
    int[] ints = intsSupplier.get();
    ThreeSum target = new ThreeSumQuadratic(ints);
```

```
    System.out.println(Arrays.toString(ints));
    Triple[] triples = target.getTriples();
    System.out.println(Arrays.toString(triples));
    assertEquals(1, triples.length);
    assertEquals(1, new ThreeSumCubic(ints).getTriples().length);
}

@Ignore // Slow
public void testGetTriples3() {
    Supplier<int[]> intsSupplier = new Source(1000, 1000).intsSupplier(10);
    int[] ints = intsSupplier.get();
    ThreeSum target = new ThreeSumQuadratic(ints);
    Triple[] triplesQuadratic = target.getTriples();
    Triple[] triplesCubic = new ThreeSumCubic(ints).getTriples();
    int expected1 = triplesCubic.length;
    assertEquals(expected1, triplesQuadratic.length);
}

@Ignore // Slow
public void testGetTriples4() {
    Supplier<int[]> intsSupplier = new Source(1500, 1000).intsSupplier(10);
    int[] ints = intsSupplier.get();
    ThreeSum target = new ThreeSumQuadratic(ints);
    Triple[] triplesQuadratic = target.getTriples();
    Triple[] triplesCubic = new ThreeSumCubic(ints).getTriples();
    int expected1 = triplesCubic.length;
    assertEquals(expected1, triplesQuadratic.length);
}

@Test
public void testGetTriplesC0() {
    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));
    assertEquals(4, triples.length);
    assertEquals(4, new ThreeSumCubic(ints).getTriples().length);
}

@Test
public void testGetTriplesC1() {
    Supplier<int[]> intsSupplier = new Source(20, 20, 1L).intsSupplier(10);
```

```
        int[] ints = intsSupplier.get();
        ThreeSum target = new ThreeSumQuadraticWithCalipers(ints);
        Triple[] triples = target.getTriples();
        assertEquals(4, triples.length);
        System.out.println(Arrays.toString(triples));
        Triple[] triples2 = new ThreeSumCubic(ints).getTriples();
        System.out.println(Arrays.toString(triples2));
        assertEquals(4, triples2.length);
    }

    @Test
    public void testGetTriplesC2() {
        Supplier<int[]> intsSupplier = new Source(10, 15, 3L).intsSupplier(10);
        int[] ints = intsSupplier.get();
        ThreeSum target = new ThreeSumQuadraticWithCalipers(ints);
        System.out.println(Arrays.toString(ints));
        Triple[] triples = target.getTriples();
        System.out.println(Arrays.toString(triples));
        assertEquals(1, triples.length);
        assertEquals(1, new ThreeSumCubic(ints).getTriples().length);
    }

    @Test
    public void testGetTriplesC3() {
        Supplier<int[]> intsSupplier = new Source(1000, 1000).intsSupplier(10);
        int[] ints = intsSupplier.get();
        ThreeSum target = new ThreeSumQuadraticWithCalipers(ints);
        Triple[] triplesQuadratic = target.getTriples();
        Triple[] triplesCubic = new ThreeSumCubic(ints).getTriples();
        assertEquals(triplesCubic.length, triplesQuadratic.length);
    }

    @Test
    public void testGetTriplesC4() {
        Supplier<int[]> intsSupplier = new Source(1500, 1000).intsSupplier(10);
        int[] ints = intsSupplier.get();
        ThreeSum target = new ThreeSumQuadraticWithCalipers(ints);
        Triple[] triplesQuadratic = target.getTriples();
        Triple[] triplesCubic = new ThreeSumCubic(ints).getTriples();
        assertEquals(triplesCubic.length, triplesQuadratic.length);
    }

}
```