Three Sum

1. Intro

In this question, I was asked to solve the **Three Sum** problem by adopting various algorithms which have different time complexity. Before implementing them, we have to figure out how they work.

> - Quadrithmic
> - Quadratic
> - QuadraticWithCalipers

2. Explanation

2.1 Quadrithmic

We can know the time complexity of this method is:

$$n^2 log_2 n \tag{1}$$

When we see the time complexity (log), it is not hard for us to understand that we need to use **Binary Search** to solve this problem.

```
    public Triple[] getTriples() {
        List<Triple> triples = new ArrayList<>();
        for (int i = 0; i < length; i++)
            for (int j = i + 1; j < length; j++) {
                Triple triple = getTriple(i, j);
                if (triple != null) triples.add(triple);
            }
```

```
        Collections.sort(triples);
        return triples.stream().distinct().toArray(Triple[]::new);
    }

    public Triple getTriple(int i, int j) {
        int index = Arrays.binarySearch(a, -a[i] - a[j]);
        if (index >= 0 && index > j) return new Triple(a[i], a[j],
a[index]);
        else return null;
    }
```

> When we want to find a triple **(a, b, c)** which their sum = 0, we just need to enumerate **a, b**. Then, adopting **Binary Search** to find **c** of which value is equal to **(-a-b)**

2.2 Quadratic

We can know the time complexity of **Quadratic** method is:

$$n^2 \tag{2}$$

From the time complexity, we know that we can use only two nested loops to implement the method. From the comments, we can see that the parameter **J** passed in is the index starting from the middle. Which means we should start traversing from its two sides, since the array is sorted.

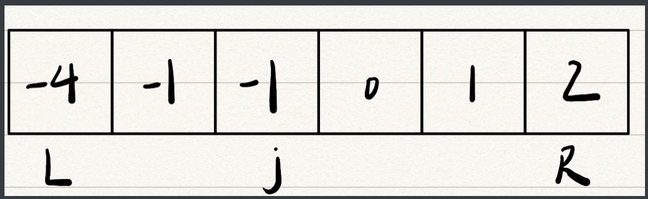

Figure 1 Traversing diagram

> sum = nums[L] + nums[j] + nums[R] = -3 < 0, so we need to increase the left index by one. When sum>0, we need to move the right index to make sure sum approaching to 0.

```
    /**
     * 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 left = 0, right = length - 1;
        while (left < j && right > j) {
            int sum = a[left] + a[j] + a[right];
            if (sum > 0) {
                right--;
            } else if (sum < 0) {
                left++;
            } else {
                triples.add(new Triple(left, j, right));
                left++;
                right--;
                //We can use these codes instead of using
"stream().distinct()" to remove duplicate triple
//                while (left < right && a[left] == a[left - 1]) left++;
//                while (left < right && a[right] == a[right + 1]) right--;
            }
        }
        // END
        return triples;
    }
```

2.3 QuadraticWithCalipers

The essence of this method is similar to the previous one (**Quadratic**). However, it uses the function interface in **lambda 8** to realize summation. Besides, its start index is not the middle index.

```
Function<Triple, Integer> function = Triple::sum;
```

```
    /**
     * 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 left = i + 1, right = a.length - 1;
        while (left < right) {
            Triple t = new Triple(a[i], a[left], a[right]);
            int sum = function.apply(t);
            if (sum > 0) {
                right--;
```

```
            } else if (sum < 0) {
                left++;
            } else {
                triples.add(t);
                left++;
                right--;
                //We can use these codes instead of using
"stream().distinct()" to remove duplicate triple
//                while (left < right && a[left] == a[left - 1]) left++;
//                while (left < right && a[right] == a[right + 1]) right--;
            }
        }
```

3. Observation

In order to deduce the relationship between **runtime** and **array length**, we must first implement test cases. In this case, in order to make the results more accurate, I run this method 100 times and then take the average value. Finally, using the **TimeLogger** to output the result.

```
    private void benchmarkThreeSum(final String description, final
Consumer<int[]> function, int n, final TimeLogger[] timeLoggers) {
        //Because it takes too long to cycle 100 times, the method will not
run ThreeSumCubic when n is greater than or equal to 2000
        if (description.equals("ThreeSumCubic") && n >= 2000) return;
        long time = 0L;
        int[] arr = supplier.get();
        //Cycle 100 times and take the average value
        for (int i = 0; i < 100; i++) {
            try (Stopwatch stopwatch = new Stopwatch()) {
                function.accept(arr);
```

```
                time += stopwatch.lap();
            }
        }
        for (TimeLogger timeLogger : timeLoggers) timeLogger.log(1.0 * time
    / 100, n);
        // END
    }
```

4. Evidence

4.1 Unit Tests

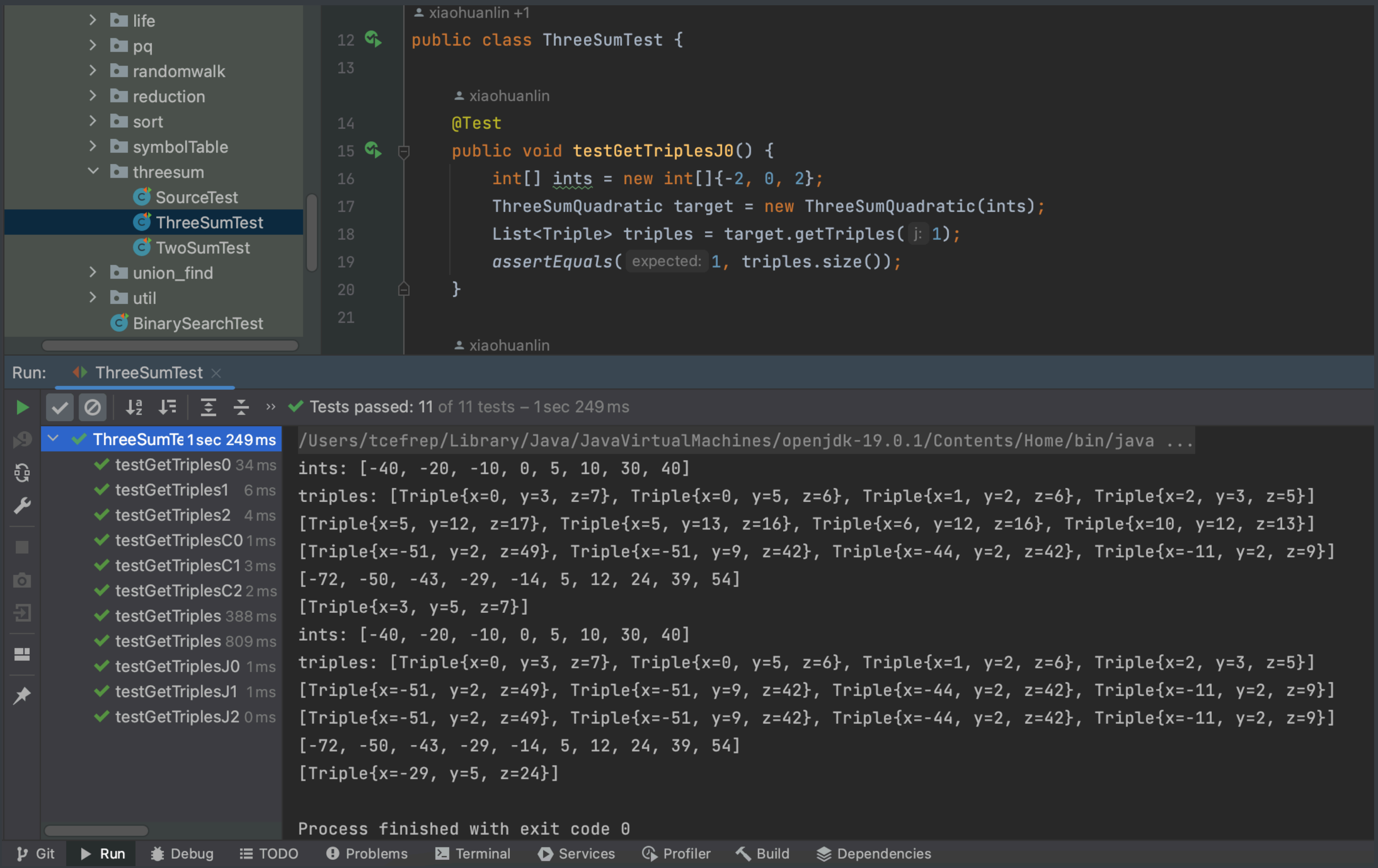

Figure 2 Screenshot of operation results

Passed all test cases.

4.2 Benchmark Three Sum

```
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();
}
```

ThreeSumBenchmark

```
/Users/tcefrep/Library/Java/JavaVirtualMachines/openjdk-19.0.1/Contents/Home/bin/java ...
ThreeSumBenchmark: N=250
2023-01-28 11:47:17 INFO  TimeLogger - Raw time per run (mSec):  .41
2023-01-28 11:47:17 INFO  TimeLogger - Normalized time per run (n^2):  6.56
2023-01-28 11:47:17 INFO  TimeLogger - Raw time per run (mSec):  .46
2023-01-28 11:47:17 INFO  TimeLogger - Normalized time per run (n^2 log n):  .92
2023-01-28 11:47:17 INFO  TimeLogger - Raw time per run (mSec):  6.87
2023-01-28 11:47:17 INFO  TimeLogger - Normalized time per run (n^3):  .44
ThreeSumBenchmark: N=500
2023-01-28 11:47:18 INFO  TimeLogger - Raw time per run (mSec):  1.39
2023-01-28 11:47:18 INFO  TimeLogger - Normalized time per run (n^2):  5.56
2023-01-28 11:47:18 INFO  TimeLogger - Raw time per run (mSec):  3.02
2023-01-28 11:47:18 INFO  TimeLogger - Normalized time per run (n^2 log n):  1.35
2023-01-28 11:47:23 INFO  TimeLogger - Raw time per run (mSec):  54.31
2023-01-28 11:47:23 INFO  TimeLogger - Normalized time per run (n^3):  .43
ThreeSumBenchmark: N=1000
2023-01-28 11:47:24 INFO  TimeLogger - Raw time per run (mSec):  3.76
2023-01-28 11:47:24 INFO  TimeLogger - Normalized time per run (n^2):  3.76
2023-01-28 11:47:26 INFO  TimeLogger - Raw time per run (mSec):  16.12
2023-01-28 11:47:26 INFO  TimeLogger - Normalized time per run (n^2 log n):  1.62
2023-01-28 11:48:09 INFO  TimeLogger - Raw time per run (mSec):  433.84
2023-01-28 11:48:09 INFO  TimeLogger - Normalized time per run (n^3):  .43
ThreeSumBenchmark: N=2000
2023-01-28 11:48:11 INFO  TimeLogger - Raw time per run (mSec):  16.90
2023-01-28 11:48:11 INFO  TimeLogger - Normalized time per run (n^2):  4.22
2023-01-28 11:48:20 INFO  TimeLogger - Raw time per run (mSec):  92.03
2023-01-28 11:48:20 INFO  TimeLogger - Normalized time per run (n^2 log n):  2.10
ThreeSumBenchmark: N=4000
2023-01-28 11:48:28 INFO  TimeLogger - Raw time per run (mSec):  80.38
2023-01-28 11:48:28 INFO  TimeLogger - Normalized time per run (n^2):  5.02
2023-01-28 11:49:14 INFO  TimeLogger - Raw time per run (mSec):  457.32
2023-01-28 11:49:14 INFO  TimeLogger - Normalized time per run (n^2 log n):  2.39
```

```
ThreeSumBenchmark: N=4000
2023-01-28 11:48:28 INFO  TimeLogger - Raw time per run (mSec):  80.38
2023-01-28 11:48:28 INFO  TimeLogger - Normalized time per run (n^2):  5.02
2023-01-28 11:49:14 INFO  TimeLogger - Raw time per run (mSec):  457.32
2023-01-28 11:49:14 INFO  TimeLogger - Normalized time per run (n^2 log n):  2.39
ThreeSumBenchmark: N=8000
2023-01-28 11:49:51 INFO  TimeLogger - Raw time per run (mSec):  366.41
2023-01-28 11:49:51 INFO  TimeLogger - Normalized time per run (n^2):  5.73
2023-01-28 11:53:18 INFO  TimeLogger - Raw time per run (mSec):  2070.18
2023-01-28 11:53:18 INFO  TimeLogger - Normalized time per run (n^2 log n):  2.49
```

Figure 3 Screenshot of operation results

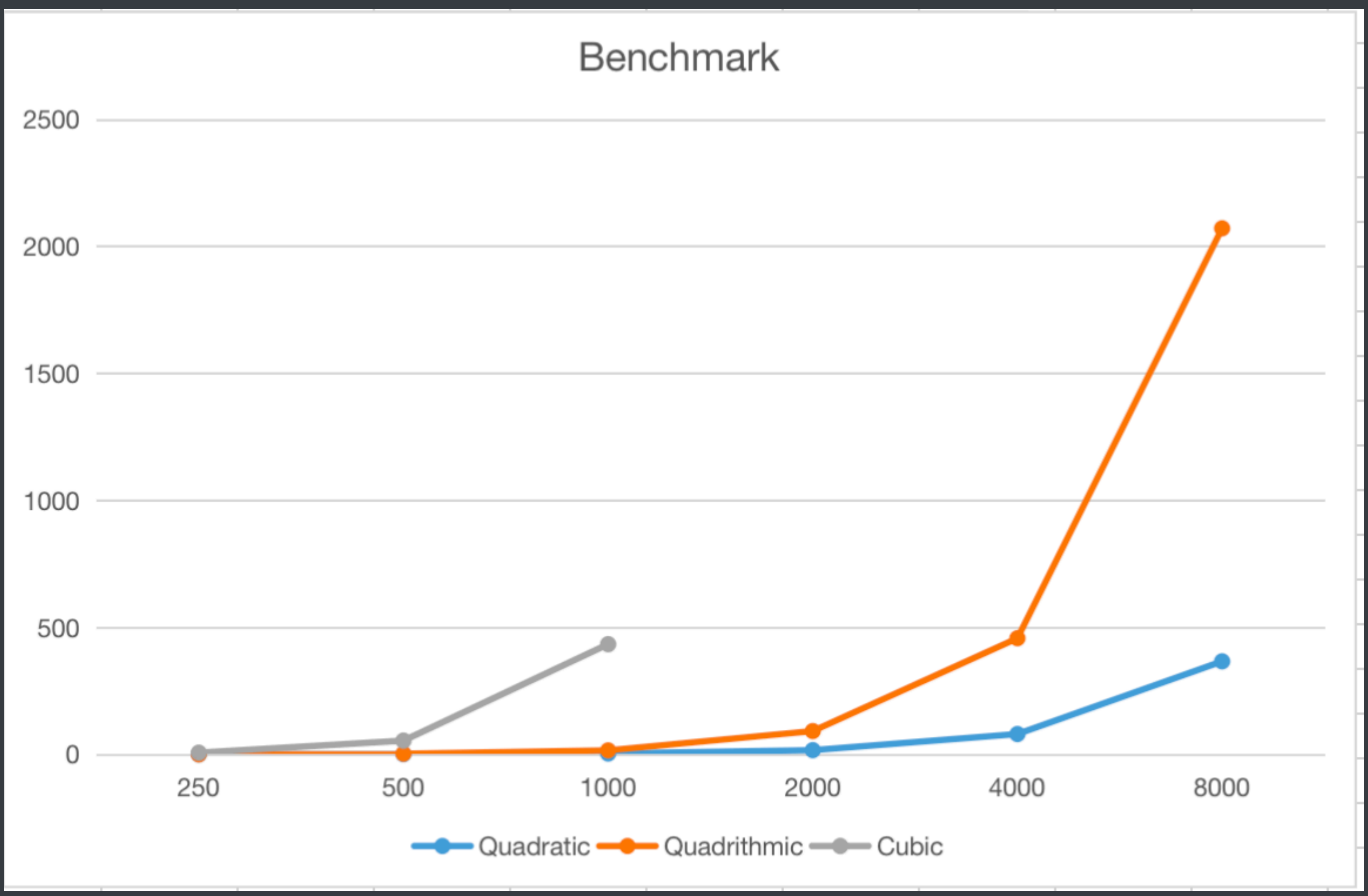

Figure 4 Line statistical chart

5. Conclusion

From the test results, the time spent on each run of the method is closely related to the time complexity of the method.

Every time we doubling the **N** (arr.length), the time spent is almost **X (Time complexity of the method)** times longer than the original time. It can be seen that when **N** is small, the time spent in running methods does not vary greatly. But when **N** gradually increases, the difference of the time spent between the three methods is very large.

Therefore, when we implement a method, we should first consider its time complexity and whether the method can be optimized. For example:

> - In the three sum method. If the value of leftmost pointer is larger than 0 or the value of rightmost pointer is smaller than 0. We can jump out of the current loop.
> - We can add this judgment to the traversal to reduce the traversal of the result set. (stream().distinct() needs to traverse the result set)
>
> ```
>                 while (left < right && a[left] == a[left - 1])
> left++;
>                 while (left < right && a[right] == a[right + 1])
> right--;
> ```