

Program Structures and Algorithms
Spring 2023(SEC 03)

NAME: Vipul Rajderkar
NUID: 002700991

Task:

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

Relationship Conclusion:

After solving 3-Sum problem using three different approaches, it can be conferred that Quadratic Approach (quadratic as well as quadratic with calipers) performs better than rest 2 viz. Quadrithmic and cubic.

Time complexities of all approaches are as below:

Quadratic: $O(n^2)$

Quadrithmic: $O(n^2 \log n)$

Cubic: $O(n^3)$

Evidence to support that conclusion:

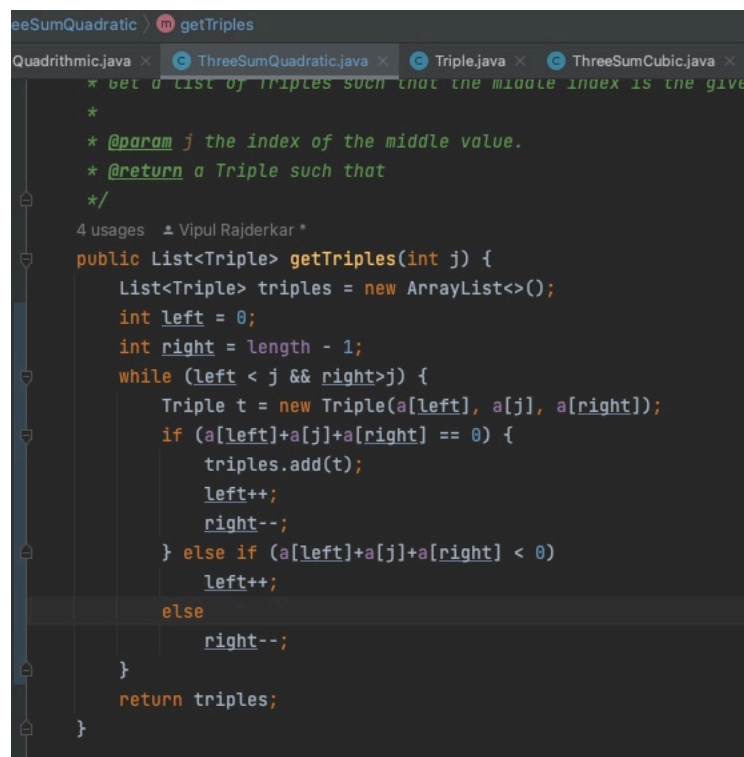
Approach 1: 3-Sum Quadratic

Step 1: Select a number as a middle number (say x) from an array (complexity: $O(n)$)

Step 2: Use 2 pointers pointing number less than and greater than middle number (x)

Step 3: While both pointers don't reach middle number's index -> check the sum and increment corresponding pointers depending on whether the sum is less than or equal to 0.

Code Snippet:



```
eeSumQuadratic > getTriples
Quadrithmic.java x ThreeSumQuadratic.java x Triple.java x ThreeSumCubic.java x
/* get a list of triples such that the middle index is the give
 *
 * @param j the index of the middle value.
 * @return a Triple such that
 */
4 usages Vipul Rajderkar *
public List<Triple> getTriples(int j) {
    List<Triple> triples = new ArrayList<>();
    int left = 0;
    int right = length - 1;
    while (left < j && right > j) {
        Triple t = new Triple(a[left], a[j], a[right]);
        if (a[left] + a[j] + a[right] == 0) {
            triples.add(t);
            left++;
            right--;
        } else if (a[left] + a[j] + a[right] < 0)
            left++;
        else
            right--;
    }
    return triples;
}
```

Approach 2: 3-Sum Quadratic with Calipers

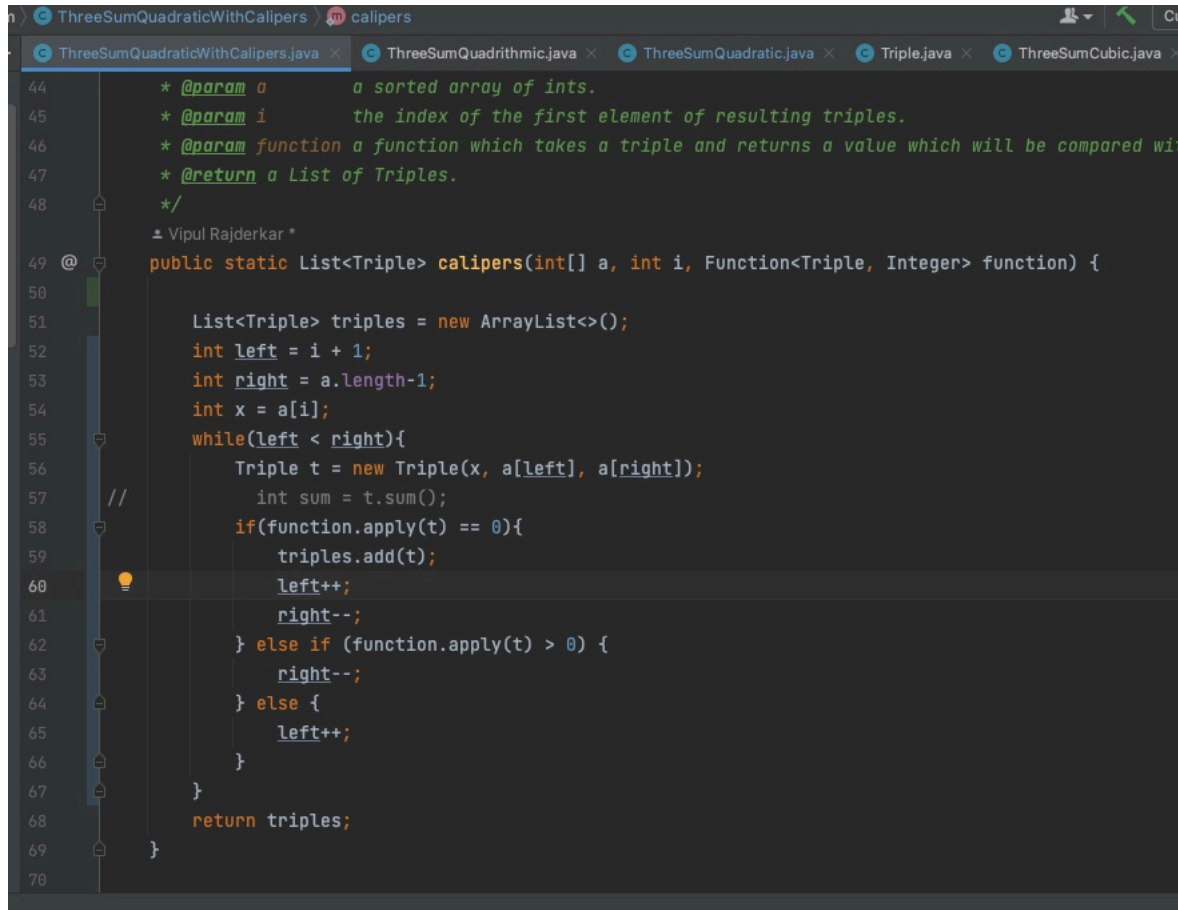
The computation is less heavy than approach 1 because of the invariant that remaining 2 numbers are going to be more than first number.

Step 1: Select a number as a first number (say x) from a sorted array (complexity: $O(n)$)

Step 2: Use 2 pointers pointing numbers just greater than x and the largest number.

Step 3: While both pointers don't reach middle number's index \rightarrow check the sum and increment/decrement corresponding pointers depending on whether the sum is less than or equal to 0.

Code Snippet:



```
44      * @param a      a sorted array of ints.
45      * @param i      the index of the first element of resulting triples.
46      * @param function a function which takes a triple and returns a value which will be compared with 0.
47      * @return a List of Triples.
48      */
49      @ Vipul Rajderkar *
50      public static List<Triple> calipers(int[] a, int i, Function<Triple, Integer> function) {
51
52          List<Triple> triples = new ArrayList<>();
53          int left = i + 1;
54          int right = a.length-1;
55          int x = a[i];
56          while(left < right){
57              Triple t = new Triple(x, a[left], a[right]);
58              // int sum = t.sum();
59              if(function.apply(t) == 0){
60                  triples.add(t);
61                  left++;
62                  right--;
63              } else if (function.apply(t) > 0) {
64                  right--;
65              } else {
66                  left++;
67              }
68          }
69          return triples;
70      }
```

Approach 3: 3-Sum Quadrithmic

Step 1: Select a pair of number x and y (Complexity: $O(n^2)$)

Step 2: Find the third element($z = -(x+y)$) using BinarySearch (Complexity: $O(\log n)$)

Code Snippet:

```
ThreeSumQuadrithmic
ThreeSumQuadraticWithCalipers.java x ThreeSumQuadrithmic.java x ThreeSumQuadratic.java x Triple.java x
22 public ThreeSumQuadrithmic(int[] a) {
23     this.a = a;
24     length = a.length;
25 }
26
27 24 usages Vipul Rajderkar
28 public Triple[] getTriples() {
29     List<Triple> triples = new ArrayList<>();
30     for (int i = 0; i < length; i++)
31         for (int j = i + 1; j < length; j++) {
32             Triple triple = getTriple(i, j);
33             if (triple != null) triples.add(triple);
34         }
35     Collections.sort(triples);
36     return triples.stream().distinct().toArray(Triple[]::new);
37 }
38 1 usage Vipul Rajderkar
39 public Triple getTriple(int i, int j) {
40     int index = Arrays.binarySearch(a, key: -a[i] - a[j]);
41     if (index >= 0 && index > j) return new Triple(a[i], a[j], a[index]);
42     else return null;
43 }
```

Benchmarking Code Snippet and Output:

```
PSA-INF06205 - ThreeSumBenchmark.java
private void benchmarkThreeSum(final String description, final Consumer<int> function, int n, final TimeLogger[] timeLoggers) {
    //if (description.equals("ThreeSumCubic") && n > 4000) return;
    double duration = 0;
    for (int i = 0; i < n; i++) {
        long startTime = System.currentTimeMillis();
        function.accept(function.get());
    }
}

Run: ThreeSumBenchmark
/Users/vipulrajderkar/Library/Java/JavaVirtualMachines/openjdk-19.0.2/Contents/Home/bin/java ...
ThreeSumBenchmark: N=250
2023-01-31 21:08:49 INFO TimeLogger - (Quadratic) Raw time per run (mSec): 1.84
2023-01-31 21:08:49 INFO TimeLogger - (Quadratic) Normalized time per run (n^2): 29.44
2023-01-31 21:08:49 INFO TimeLogger - (Quadratic with Calipers) Raw time per run (mSec): 1.89
2023-01-31 21:08:49 INFO TimeLogger - (Quadratic with Calipers) Normalized time per run (n^2): 17.44
2023-01-31 21:08:50 INFO TimeLogger - (Quadrithmic) Raw time per run (mSec): 1.08
2023-01-31 21:08:50 INFO TimeLogger - (Quadrithmic) Normalized time per run (n^2 log n): 2.17
2023-01-31 21:08:50 INFO TimeLogger - (Cubic) Raw time per run (mSec): 5.97
2023-01-31 21:08:50 INFO TimeLogger - (Cubic) Normalized time per run (n^3): .38
ThreeSumBenchmark: N=500
2023-01-31 21:08:50 INFO TimeLogger - (Quadratic) Raw time per run (mSec): 3.14
2023-01-31 21:08:50 INFO TimeLogger - (Quadratic) Normalized time per run (n^2): 12.56
2023-01-31 21:08:50 INFO TimeLogger - (Quadratic with Calipers) Raw time per run (mSec): 1.34
2023-01-31 21:08:50 INFO TimeLogger - (Quadratic with Calipers) Normalized time per run (n^2): 4.56
2023-01-31 21:08:51 INFO TimeLogger - (Quadrithmic) Raw time per run (mSec): 2.88
2023-01-31 21:08:51 INFO TimeLogger - (Quadrithmic) Normalized time per run (n^2 log n): 1.28
2023-01-31 21:08:51 INFO TimeLogger - (Cubic) Raw time per run (mSec): 40.68
2023-01-31 21:08:51 INFO TimeLogger - (Cubic) Normalized time per run (n^3): .33
ThreeSumBenchmark: N=1000
2023-01-31 21:08:53 INFO TimeLogger - (Quadratic) Raw time per run (mSec): 6.80
2023-01-31 21:08:53 INFO TimeLogger - (Quadratic) Normalized time per run (n^2): 6.80
2023-01-31 21:08:53 INFO TimeLogger - (Quadratic with Calipers) Raw time per run (mSec): 4.25
2023-01-31 21:08:53 INFO TimeLogger - (Quadratic with Calipers) Normalized time per run (n^2): 4.25
2023-01-31 21:08:53 INFO TimeLogger - (Quadrithmic) Raw time per run (mSec): 14.25
2023-01-31 21:08:53 INFO TimeLogger - (Quadrithmic) Normalized time per run (n^2 log n): 1.43
2023-01-31 21:08:59 INFO TimeLogger - (Cubic) Raw time per run (mSec): 517.98
2023-01-31 21:08:59 INFO TimeLogger - (Cubic) Normalized time per run (n^3): .32
```

```
PSA-INF06205 src main java edu neu coe info205 threesum ThreeSumBenchmark runBenchmarks
36 private void benchmarkThreeSum(final String description, final Consumer<int[]> function, int n, final TimeLogger[] timeLoggers) {
37     //if (description.equals("ThreesumCubic") && n > 4000) return;
38
39     double duration = 0;
40     for(int i=0; i< runs; i++){
41         long startTime = System.currentTimeMillis();
42         function.accept(supplier.get());
43     }
44     duration = duration / runs;
45     System.out.println(description+" Total time taken "+ duration+ " for n "+n);
46     for(TimeLogger timeLogger : timeLoggers){
47         timeLogger.log(duration, n);
48     }
49 }
50
51 @Usage
52 private final static TimeLogger[] timeLoggersCubic = {
53     new TimeLogger("@@Cubic Raw time per run (mSec): ", (time, n) -> time),
54     new TimeLogger("@@Cubic Normalized time per run (m^3): ", (time, n) -> time / n / n * 1ec)
55 };
56
57 @Usage
```

Run: ThreeSumBenchmark - N=2880

Time	Logger	Algorithm	Raw time per run (mSec)	Normalized time per run (n^2)
2023-01-31 21:09:00	INFO	TimeLogger	(Quadratic) Raw time per run (mSec): 28.20	(Quadratic) Normalized time per run (n^2): 7.85
2023-01-31 21:09:00	INFO	TimeLogger	(Quadratic with Calipers) Raw time per run (mSec): 19.60	(Quadratic with Calipers) Normalized time per run (n^2): 4.75
2023-01-31 21:09:01	INFO	TimeLogger	(Quadrithmic) Raw time per run (mSec): 71.40	(Quadrithmic) Normalized time per run (n^2 log n): 1.63
2023-01-31 21:09:26	INFO	TimeLogger	(Cubic) Raw time per run (mSec): 2518.70	(Cubic) Normalized time per run (n^3): .31
2023-01-31 21:09:26	INFO	TimeLogger	(Quadratic) Raw time per run (mSec): 149.20	(Quadratic) Normalized time per run (n^2): 9.32
2023-01-31 21:09:27	INFO	TimeLogger	(Quadratic with Calipers) Raw time per run (mSec): 91.80	(Quadratic with Calipers) Normalized time per run (n^2): 5.74
2023-01-31 21:09:29	INFO	TimeLogger	(Quadrithmic) Raw time per run (mSec): 332.40	(Quadrithmic) Normalized time per run (n^2 log n): 1.74
2023-01-31 21:11:09	INFO	TimeLogger	(Cubic) Raw time per run (mSec): 20007.60	(Cubic) Normalized time per run (n^3): .31
2023-01-31 21:11:09	INFO	TimeLogger	(Quadratic) Raw time per run (mSec): 686.67	(Quadratic) Normalized time per run (n^2): 10.75
2023-01-31 21:11:12	INFO	TimeLogger	(Quadratic with Calipers) Raw time per run (mSec): 527.67	(Quadratic with Calipers) Normalized time per run (n^2): 8.24
2023-01-31 21:11:17	INFO	TimeLogger	(Quadrithmic) Raw time per run (mSec): 1464.33	(Quadrithmic) Normalized time per run (n^2 log n): 1.77
2023-01-31 21:19:22	INFO	TimeLogger	(Cubic) Raw time per run (mSec): 161619.00	(Cubic) Normalized time per run (n^3): .32

Build completed successfully in 3 sec, 601 ms - today 8:08 PM

```
PSA-INF06205 src main java edu neu coe info205 threesum ThreeSumBenchmark runBenchmarks
36 private void benchmarkThreeSum(final String description, final Consumer<int[]> function, int n, final TimeLogger[] timeLoggers) {
37     //if (description.equals("ThreesumCubic") && n > 4000) return;
38
39     double duration = 0;
40     for(int i=0; i< runs; i++){
41         long startTime = System.currentTimeMillis();
42         function.accept(supplier.get());
43         long endTime = System.currentTimeMillis();
44         duration += (endTime - startTime);
45     }
46     duration = duration / runs;
47     System.out.println(description+" Total time taken "+ duration+ " for n "+n);
48     for(TimeLogger timeLogger : timeLoggers){
49         timeLogger.log(duration, n);
50     }
51 }
52
53 @Usage
54 private final static TimeLogger[] timeLoggersCubic = {
55     new TimeLogger("@@Cubic Raw time per run (mSec): ", (time, n) -> time),
56     new TimeLogger("@@Cubic Normalized time per run (m^3): ", (time, n) -> time / n / n * 1ec)
57 };
58
59 @Usage
```

Run: ThreeSumBenchmark - N=16000

Time	Logger	Algorithm	Raw time per run (mSec)	Normalized time per run (n^2)
2023-01-31 21:19:27	INFO	TimeLogger	(Quadratic) Raw time per run (mSec): 2676.50	(Quadratic) Normalized time per run (n^2): 10.46
2023-01-31 21:19:32	INFO	TimeLogger	(Quadratic with Calipers) Raw time per run (mSec): 2243.50	(Quadratic with Calipers) Normalized time per run (n^2): 8.84
2023-01-31 21:19:45	INFO	TimeLogger	(Quadrithmic) Raw time per run (mSec): 6888.00	(Quadrithmic) Normalized time per run (n^2 log n): 1.93
2023-01-31 22:02:37	INFO	TimeLogger	(Cubic) Raw time per run (mSec): 1208370.00	(Cubic) Normalized time per run (n^3): .31

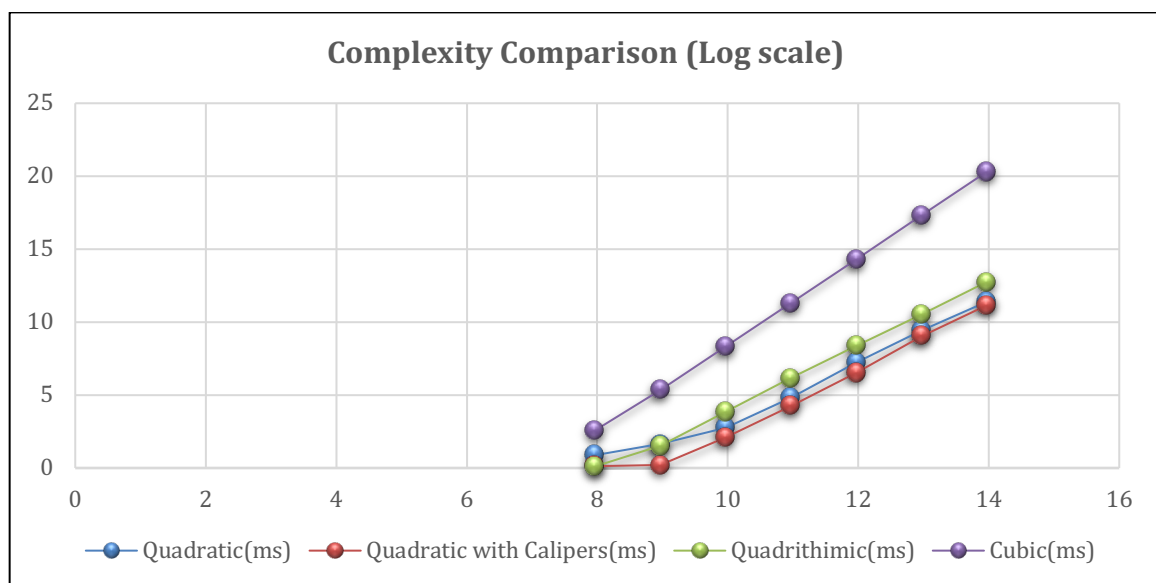
Process finished with exit code 0

Build completed successfully in 3 sec, 601 ms - today 8:08 PM

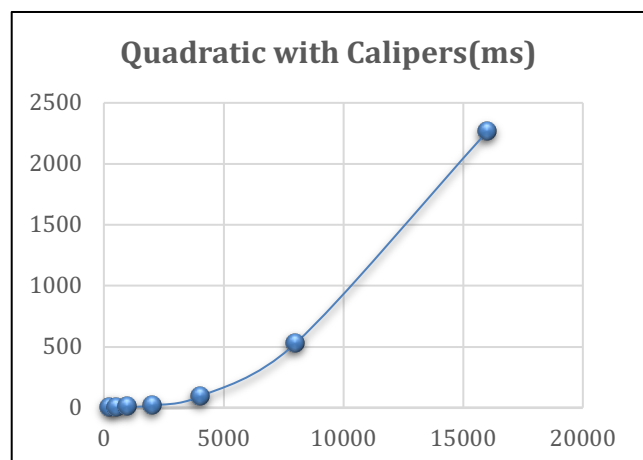
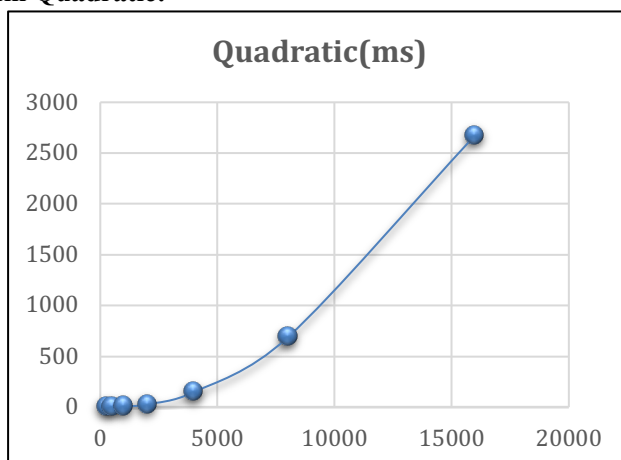
Graphical Representation:

Timing Table:

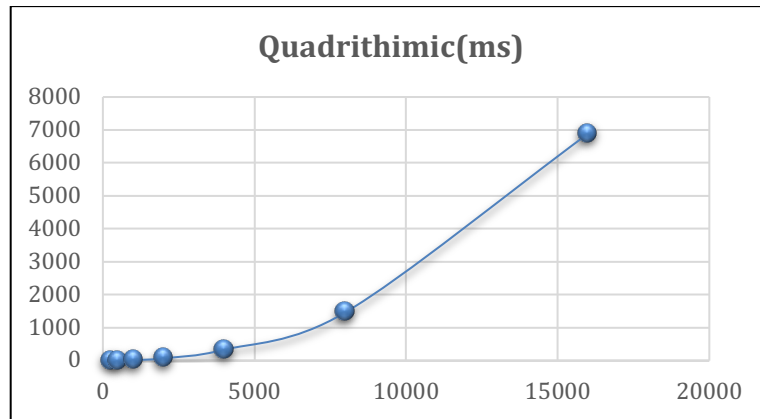
N	Quadratic(ms)	Quadratic with Calipers(ms)	Quadrithimic(ms)	Cubic(ms)
250	1.84	1.09	1.08	5.97
500	3.14	1.14	2.88	40.68
1000	6.8	4.25	14.25	317.9
2000	28.2	19	71.4	2518.7
4000	149.2	91.8	332.6	20007.6
8000	686.67	527.67	1466.33	161619
16000	2676.5	2263.5	6888	1285873



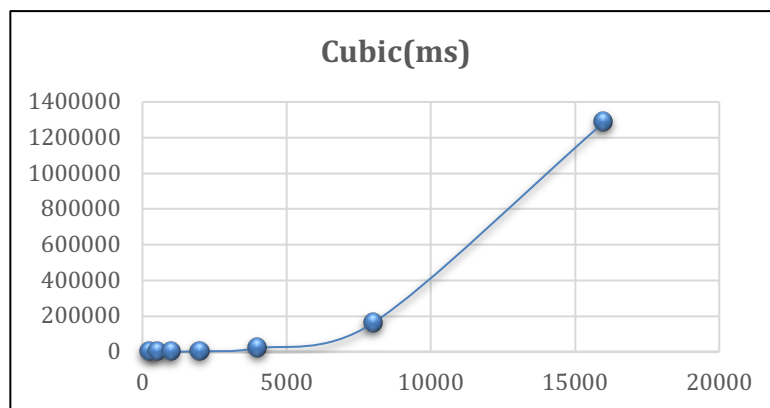
3-Sum Quadratic:



3-Sum Quadrithimic:



Cubic:



Unit Test Screenshots:

[illegible]

Explanation of why Quadratics work:

Quadratic approach performs definitely better than the rest as it is most efficient approach with a time complexity of $O(N^2)$. Since we use a two-pointer approach, the computation time is reduced as depending on the difference between expected and actual sum only required pointers can be moved.