

Deep Shah (002766755)  
**Program Structure & Algorithms**  
Spring 2023(Sec 03)

## **Assignment-2**

### **Task:**

There are two task to be performed.

Task list:

- Implement the 3-SUM code using the Quadrithmic, Quadratic, and QuadraticWithCalipers approaches for the experiment.
- Comparing the time complexities of 3-SUM for the above three approaches and adding Cubic.

### **Relationship Conclusion:**

There are mainly three approaches for solving 3-SUM problem.

1. Cubic
2. Quadratic
3. Quadramithic

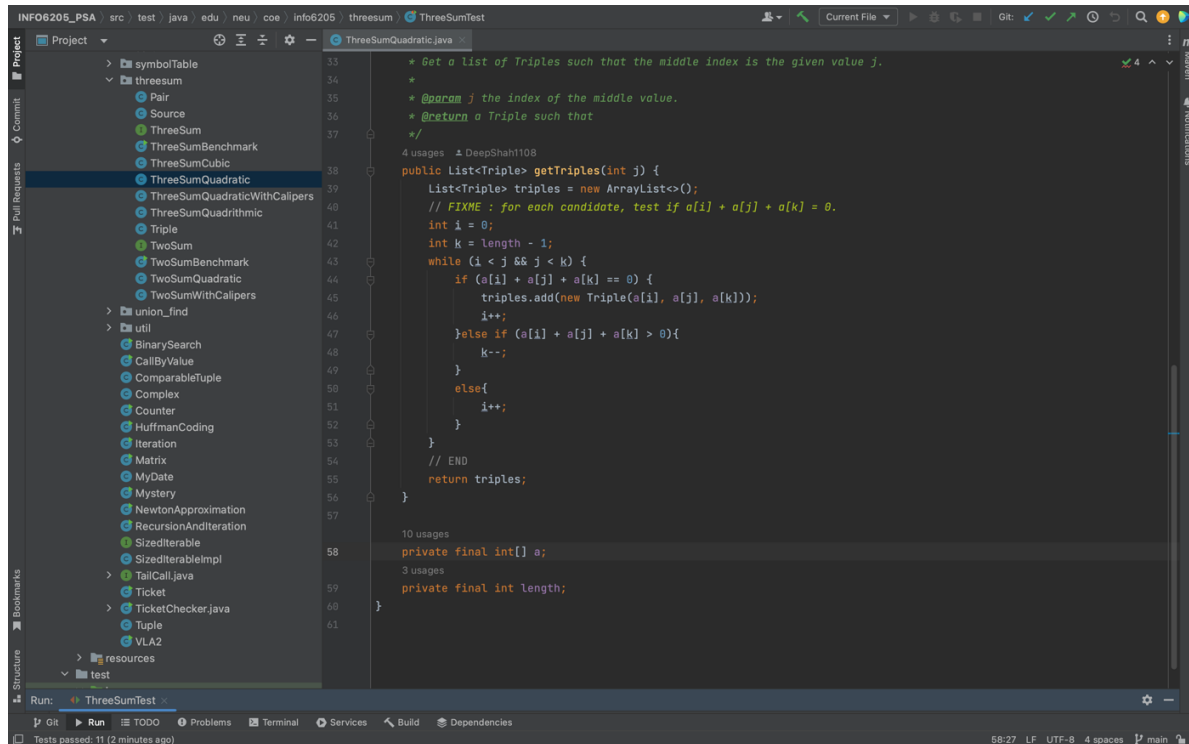
Looking at the time complexity for the above approaches. Cubic approach has the worst time complexity with  $O(N^3)$ . Quadrithmic approaches is a bit better than the Cubic but not the best having a time complexity of  $O(N^2 \log N)$ .

The best and the efficient approach is the Quadratic approach that has the time complexity of  $O(N^2)$ . Also, we tried the Quadratic approach with calipers that is basically a two-pointer approach. In this the time complexity is same as that of Quadratic (i.e.,  $O(N^2)$ ), but the run time is much faster than the quadratic. Hence, we get the result in much less time.

## Evidence to Support that conclusion:

After solving the 3-SUM problem, there were few observations made for Quadratic, Quadrithmic and QuadraticWithCaplipers and created a table for the following data.

For Quadratic:



The screenshot shows an IDE with a project named 'INFO6205\_PSA'. The left sidebar displays a file tree with various files, including 'ThreeSumQuadratic.java'. The main editor window shows the code for 'ThreeSumQuadratic.java'. The code is a Java class with a method 'getTriples(int j)' that returns a list of triples. The code is as follows:

```
33  + Get a list of Triples such that the middle index is the given value j.
34  +
35  + @param j the index of the middle value.
36  + @return a Triple such that
37  +
38  4 usages  A DeepShah1108
39  public List<Triple> getTriples(int j) {
40      List<Triple> triples = new ArrayList<>();
41      // FIXME : for each candidate, test if a[i] + a[j] + a[k] = 0.
42      int i = 0;
43      int k = length - 1;
44      while (i < j && j < k) {
45          if (a[i] + a[j] + a[k] == 0) {
46              triples.add(new Triple(a[i], a[j], a[k]));
47              i++;
48          } else if (a[i] + a[j] + a[k] > 0) {
49              k--;
50          } else {
51              i++;
52          }
53      }
54      // END
55      return triples;
56  }
57
58  10 usages
59  private final int[] a;
60  3 usages
61  private final int length;
62  }
```

N	log N	Quadratic Time (mSecs)	Quadratic log T
250	5.52146092	1.04	0.039220713
500	6.2146081	1.78	0.576613364
1000	6.90775528	4.6	1.526056303
2000	7.60090246	18.1	2.895911938
4000	8.29404964	81.8	4.404277244
8000	8.98719682	402.33	5.997272647
16000	9.680344	1950	7.575584652

For QuadraticWithCalipers:

```

47  * @return a List of Triples.
48  */
49  @
50  public static List<Triple> calipers(int[] a, int i, Function<Triple, Integer> function) {
51      List<Triple> triples = new ArrayList<>();
52      // FIXME : use function to qualify triples and to navigate otherwise.
53      int startIndex = 0 ;
54      int endIndex = a.length - 1;
55      while(startIndex < i && i < endIndex){
56          Triple triplet = new Triple(a[i], a[startIndex], a[endIndex]);
57          if(function.apply(triplet) == 0){
58              triples.add(triplet);
59              startIndex++;
60          }
61          else if(function.apply(triplet) > 0){
62              endIndex--;
63          }
64          else{
65              startIndex++;
66          }
67      }
68      // END
69      return triples;
70  }
71
72  2 usages
73  private final int[] a;
74  2 usages
75  private final int length;

```

N	log N	QuadraticWithCalipers Time (mSecs)	QuadraticWithCalipers log T
250	5.52146092	0.6	-0.510825624
500	6.2146081	1.16	0.148420005
1000	6.90775528	3.8	1.335001067
2000	7.60090246	18.3	2.90690106
4000	8.29404964	86	4.454347296
8000	8.98719682	335.67	5.816128534
16000	9.680344	2024.5	7.613078035

For Quadrithmic:

```

22 public ThreeSumQuadrithmic(int[] a) {
23     this.a = a;
24     length = a.length;
25 }
26
27 public Triple[] getTriples() {
28     List<Triple> triples = new ArrayList<>();
29     for (int i = 0; i < length; i++)
30         for (int j = i + 1; j < length; j++) {
31             Triple triple = getTriple(i, j);
32             if (triple != null) triples.add(triple);
33         }
34     Collections.sort(triples);
35     return triples.stream().distinct().toArray(Triple[]::new);
36 }
37
38 public Triple getTriple(int i, int j) {
39     int index = Arrays.binarySearch(a, key: -a[i] - a[j]);
40     if (index >= 0 && index > j) return new Triple(a[i], a[j], a[index]);
41     else return null;
42 }
43
44 private final int[] a;
45 private final int length;
46 }
47

```

N	log N	Quadrithmic Time (mSecs)	Quadrithmic log T
250	5.52146092	1.34	0.292669614
500	6.2146081	2.84	1.043804052
1000	6.90775528	12.95	2.561095788
2000	7.60090246	69.4	4.239886868
4000	8.29404964	321	5.771441123
8000	8.98719682	1374	7.225481473
16000	9.680344	5917	8.685584843

For Cubic:

```
15  /**  
16   * Construct a ThreeSumCubic on a.  
17   * @param a an array.  
18   */  
19  11 usages  DeepShah1108  
20  public ThreeSumCubic(int[] a) {  
21      this.a = a;  
22      length = a.length;  
23  }  
24  24 usages  DeepShah1108  
25  public Triple[] getTriples() {  
26      List<Triple> triples = new ArrayList<>();  
27      for (int i = 0; i < length; i++)  
28          for (int j = i + 1; j < length; j++) {  
29              for (int k = j + 1; k < length; k++) {  
30                  if (a[i] + a[j] + a[k] == 0)  
31                      triples.add(new Triple(a[i], a[j], a[k]));  
32              }  
33          }  
34      Collections.sort(triples);  
35      return triples.stream().distinct().toArray(Triple[]::new);  
36  }  
37  7 usages  
38  private final int[] a;  
39  4 usages  
40  private final int length;  
41  }  
42  }  
43  }  
44  }  
45  }  
46  }  
47  }  
48  }
```

N	log N	Cubic Time (mSecs)	Cubic log T
250	5.52146092	3.97	1.37876609
500	6.2146081	24.68	3.2059932
1000	6.90775528	188.55	5.23936322
2000	7.60090246	1416.5	7.25594432
4000	8.29404964	10670.4	9.27522883
8000	8.98719682		
16000	9.680344		

Running the Benchmark Test to calculate the time for all of the 3-SUM approach.

The screenshot shows an IDE with a project named 'INFO6205\_PSA'. The left sidebar shows a file tree with 'ThreeSumBenchmark.java' selected. The main editor shows the code for 'benchmarkThreeSum'. The bottom panel shows the output of the benchmark test, which includes raw and normalized time per run for different values of N (250, 500, 1000, 2000, 4000, 8000, 16000).

```
private void benchmarkThreeSum(final String description, final Consumer<int[]> function, int n, final TimeLogger[] timeLoggers) {
    if (description.equals("ThreeSumCubic") && n > 4000) return;
    // FIXME
    double duration = 0;
    for (int i = 0; i < runs; i++) {
        long startTime = System.currentTimeMillis();
        function.accept(supplier.get());
        long endTime = System.currentTimeMillis();
        duration += (endTime - startTime);
    }
    duration = duration / runs;
    for (TimeLogger timeLogger : timeLoggers) {
        timeLogger.log(duration, n);
    }
} // END
```

Run: ThreeSumBenchmark

ThreeSumBenchmark: N=250

2023-01-28 11:08:04 INFO TimeLogger - Raw time per run (mSec): .83

2023-01-28 11:08:04 INFO TimeLogger - Normalized time per run (n^2): 13.28

2023-01-28 11:08:04 INFO TimeLogger - Raw time per run (mSec): 1.05

2023-01-28 11:08:04 INFO TimeLogger - Normalized time per run (n^2 log n): 2.11

2023-01-28 11:08:05 INFO TimeLogger - Raw time per run (mSec): 3.66

2023-01-28 11:08:05 INFO TimeLogger - Normalized time per run (n^3): .23

2023-01-28 11:08:05 INFO TimeLogger - Raw time per run (mSec): .50

2023-01-28 11:08:05 INFO TimeLogger - Normalized time per run (n^2): 8.00

ThreeSumBenchmark: N=500

2023-01-28 11:08:05 INFO TimeLogger - Raw time per run (mSec): 1.60

2023-01-28 11:08:05 INFO TimeLogger - Normalized time per run (n^2): 6.40

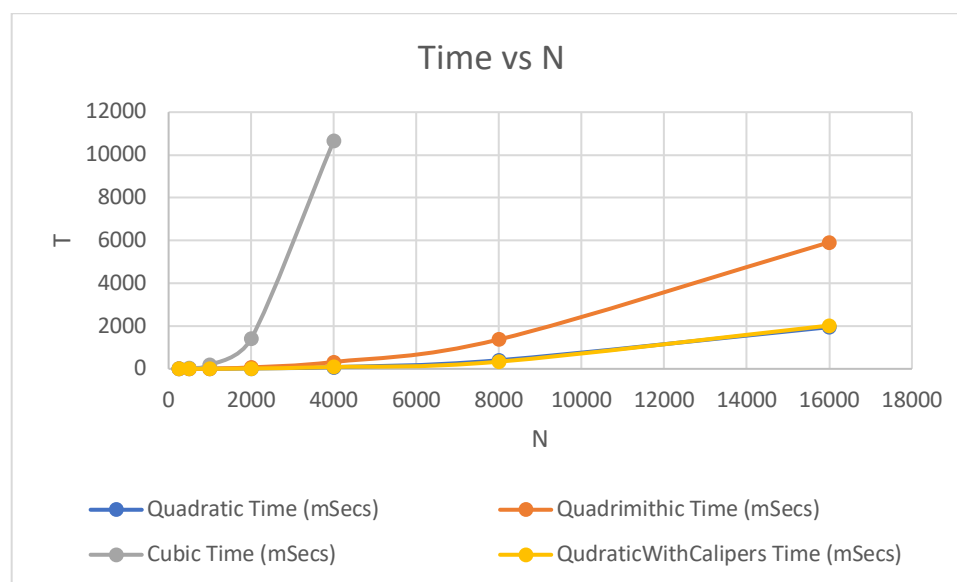
2023-01-28 11:08:05 INFO TimeLogger - Raw time per run (mSec): 2.50

2023-01-28 11:08:05 INFO TimeLogger - Normalized time per run (n^2 log n): 1.12

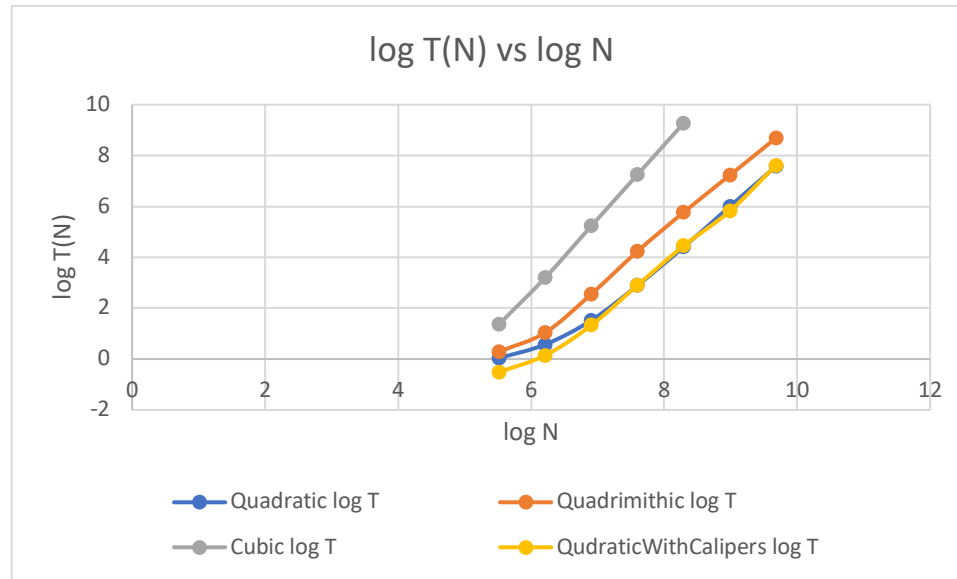
## Graphical Representation:

As per the observations made, created the detailed excel sheet providing graphical representation of the data.

### 1. Graph between N(Size) and Time



## 2. Graph between Natural log(N) and Natural log(T(N))



## Unit Tests Result:

```
INFO6205_PSA src test java edu neu coe info6205 symbolTable
Project
  pq
  randomwalk
  reduction
  sort
  symbolTable
    SourceTest
    ThreeSumTest
    TwoSumTest
  union_find
  util
  BinarySearchTest
  ComparableTupleTest
  HuffmanCodingTest
  MyDateTest
  TailCallTest
  TicketTest
  TupleTest
Pull Requests
Commit
Run: ThreeSumTest x
  Tests passed: 11 of 11 tests - 899ms
  ThreeSumTest (edu.neu.coe.info6205) 899ms
    testGetTriples0 14ms
    testGetTriples1 10ms
    testGetTriples2 1ms
    testGetTriplesC0 0ms
    testGetTriplesC1 1ms
    testGetTriplesC2 0ms
    testGetTriplesC3 254ms
    testGetTriplesC4 619ms
    testGetTriplesJ0 0ms
    testGetTriplesJ1 0ms
    testGetTriplesJ2 0ms
  Process finished with exit code 0
```

```

40  @test
41  public void testGetTriples0() {
42      int[] ints = new int[]{30, -40, -20, -10, 40, 0, 10, 5};
43      Arrays.sort(ints);
44      System.out.println("ints: " + Arrays.toString(ints));
45      ThreeSum target = new ThreeSumQuadratic(ints);
46      Triple[] triples = target.getTriples();
47      System.out.println("triples: " + Arrays.toString(triples));
48      assertEquals("expected: 4, triples.length", 4, triples.length);
49      assertEquals("expected: 4, new ThreeSumCubic(ints).getTriples().length", 4, new ThreeSumCubic(ints).getTriples().length);
50  }
51
52  no usages  DeepShah1108
53  @Test
54  public void testGetTriples1() {
55      Supplier<int[]> intsSupplier = new Source( N: 20, M: 20, seed: 1L).intsSupplier( safetyFactor: 10);
56      int[] ints = intsSupplier.get();

```

```

/Library/Java/JavaVirtualMachines/jdk-19.jdk/Contents/Home/bin/java ...
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}]
[Tuple{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}]
[Tuple{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]
[Tuple{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}]
[Tuple{x=2, y=-51, z=49}, Triple{x=2, y=-44, z=42}, Triple{x=2, y=-11, z=9}, Triple{x=9, y=-51, z=42}]
[Tuple{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]
[Tuple{x=5, y=-29, z=24}]

Process finished with exit code 0

```

Tests passed: 11 (moments ago)
 12:14 LF UTF-8 4 spaces main

## Explanation of why Quadratics work:

Quadratic approach is the best as it is most efficient approach with a time complexity of  $O(N^2)$ . In this approach we use a two-pointer approach, such that we iterate through the array from both the start as well as from the end which takes a less time to finish the task.

To better understand it, let us consider an example.

Taking the array as  $\{-40, -20, -10, 0, 5, 10, 30, 40\}$

So in this we have the middle element of the triplet of the 3-SUM.

Suppose the index of the middle element is 5. ( $j=5$ )

1. Initializing the start Index ( $i$ ) as index of first element of the array and end Index ( $k$ ) as index of last element of the array.
2. Iterating through the array we calculate:  
 $A[i] + A[j] + A[k] = 0$   
If this true, then the value for the pair  $A[i], A[j], A[k]$  will be added to the triplet
3. If  $A[i] + A[j] + A[k]$  is greater than 0, we decrement the value of end Index ( $k$ )
4. If both steps 2 and 3 are false, we then increment the value of start Index ( $i$ ).

In this we iterate the above steps until the array is fully processed.

Hence Quadratic approach is the best and the most efficient approach.