Program Structures and Algorithms
Spring 2024

NAME: Rahul Pothirendi
NUID: 002889957
GITHUB LINK - https://github.com/Pothirendirahul/INFO6205.git

Report for INFO6205 Assignment 2

Task2 -

Solve 3-SUM using the *Quadrithmic*, *Quadratic*, and (bonus point) *quadraticWithCalipers* approaches, as shown in skeleton code in the repository. There are hints at the end of Lesson 2.5 Entropy.

There are also hints in the comments of the existing code. There are a number of unit tests which you should be able to run successfully.

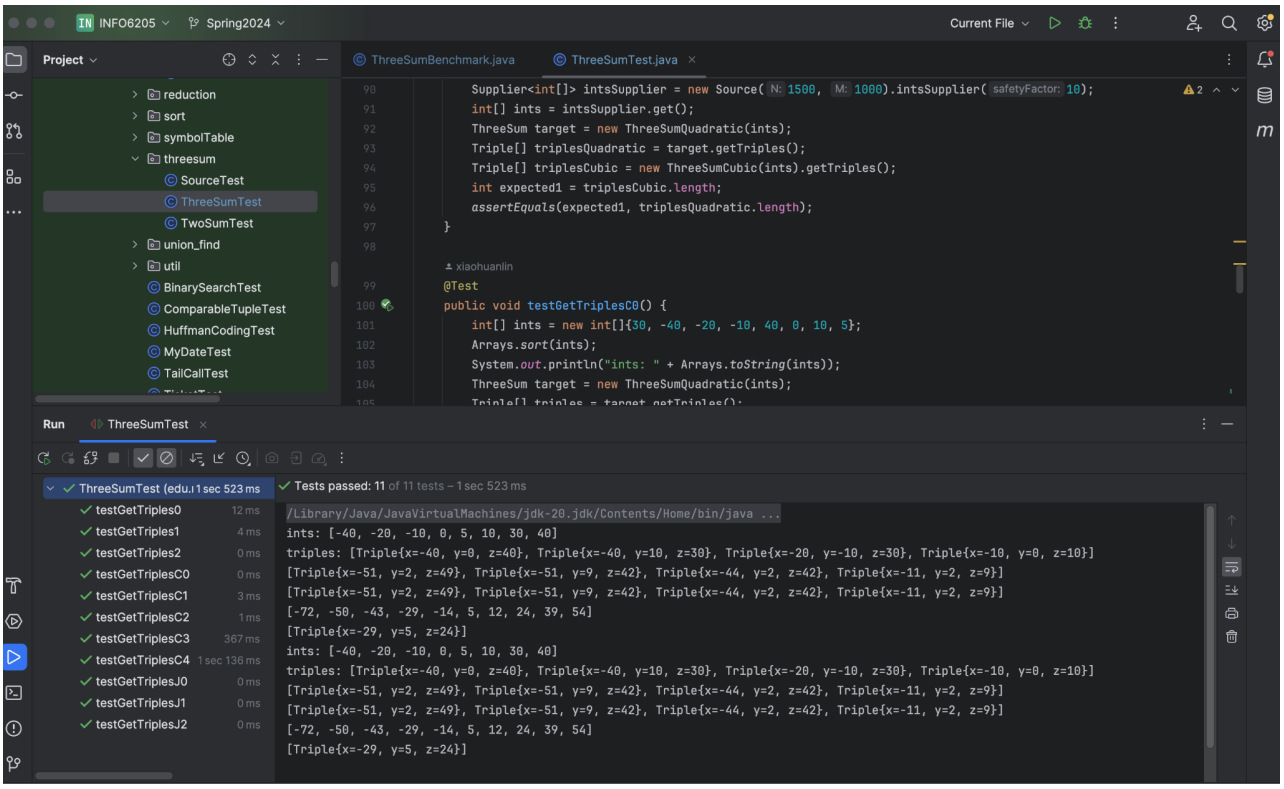
Submit (in your own repository--see instructions elsewhere--include the source code and the unit tests of course):

(a) evidence (screenshot) of your unit tests running (try to show the actual unit test code as well as the green strip);

(b) a spreadsheet showing your timing observations--using the doubling method for at least five values of N--for each of the algorithms (include cubic); Timing should be performed either with an actual stopwatch (e.g. your iPhone) or using the Stopwatch class in the repository.

(c) your brief explanation of why the quadratic method(s) work.

Screenshot for running unit tests



After completing the missing code in the ThreeSumBenchMark and other classes we are finally able to print out the time. For recording the time in we have used Stopwatch class
Terminal Output of the

ThreeSumBenchmark: N=250
ThreeSumQuadratic - Raw avg time per run (ms): 2.901
ThreeSumQuadrithmic - Raw avg time per run (ms): 2.636
ThreeSumCubic - Raw avg time per run (ms): 5.668

ThreeSumBenchmark: N=500
ThreeSumQuadratic - Raw avg time per run (ms): 2.546
ThreeSumQuadrithmic - Raw avg time per run (ms): 3.938
ThreeSumCubic - Raw avg time per run (ms): 35.094

ThreeSumBenchmark: N=1000
ThreeSumQuadratic - Raw avg time per run (ms): 7.691
ThreeSumQuadrithmic - Raw avg time per run (ms): 20.201
ThreeSumCubic - Raw avg time per run (ms): 268.386

ThreeSumBenchmark: N=2000
ThreeSumQuadratic - Raw avg time per run (ms): 30.110
ThreeSumQuadrithmic - Raw avg time per run (ms): 101.067
ThreeSumCubic - Raw avg time per run (ms): 2101.518

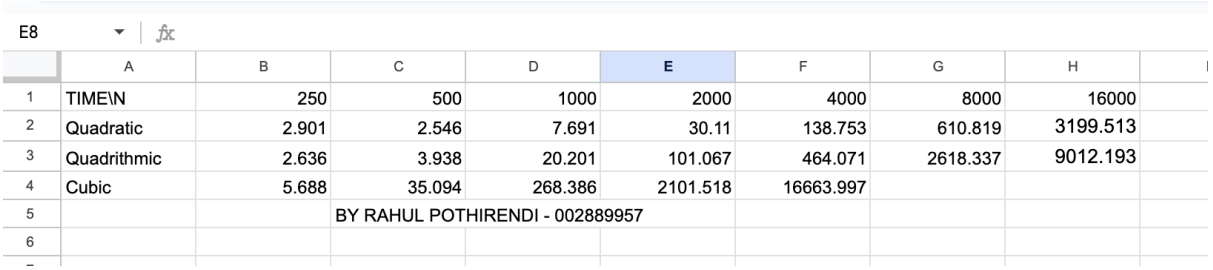
ThreeSumBenchmark: N=4000
ThreeSumQuadratic - Raw avg time per run (ms): 138.753
ThreeSumQuadrithmic - Raw avg time per run (ms): 464.071
ThreeSumCubic - Raw avg time per run (ms): 16663.997

ThreeSumBenchmark: N=8000
ThreeSumQuadratic - Raw avg time per run (ms): 610.819
ThreeSumQuadrithmic - Raw avg time per run (ms): 2168.337

ThreeSumBenchmark: N=16000
ThreeSumQuadratic - Raw avg time per run (ms): 3199.513
ThreeSumQuadrithmic - Raw avg time per run (ms): 9012.193

Process finished with exit code 0

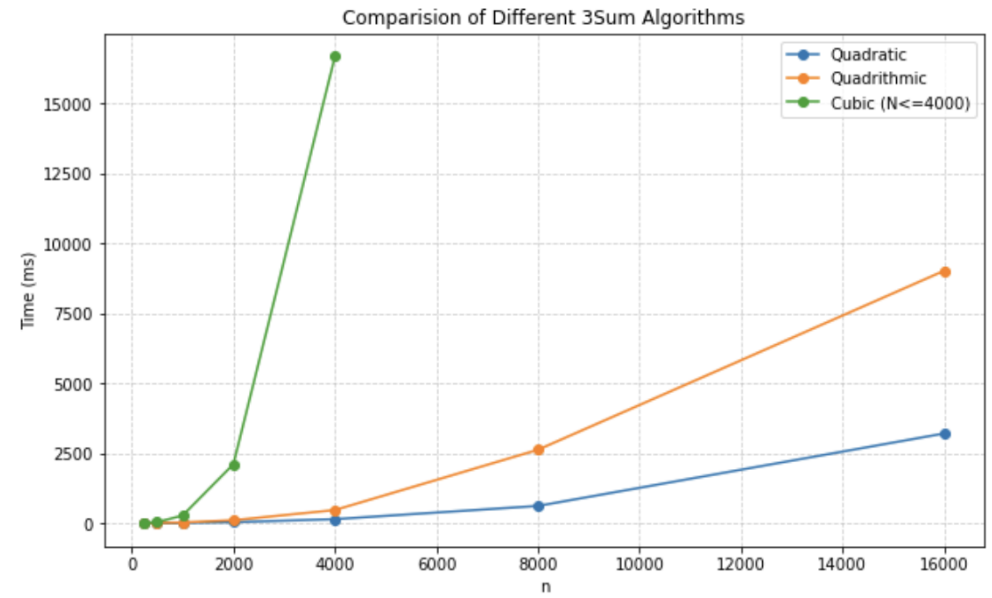
SPREADSHEET



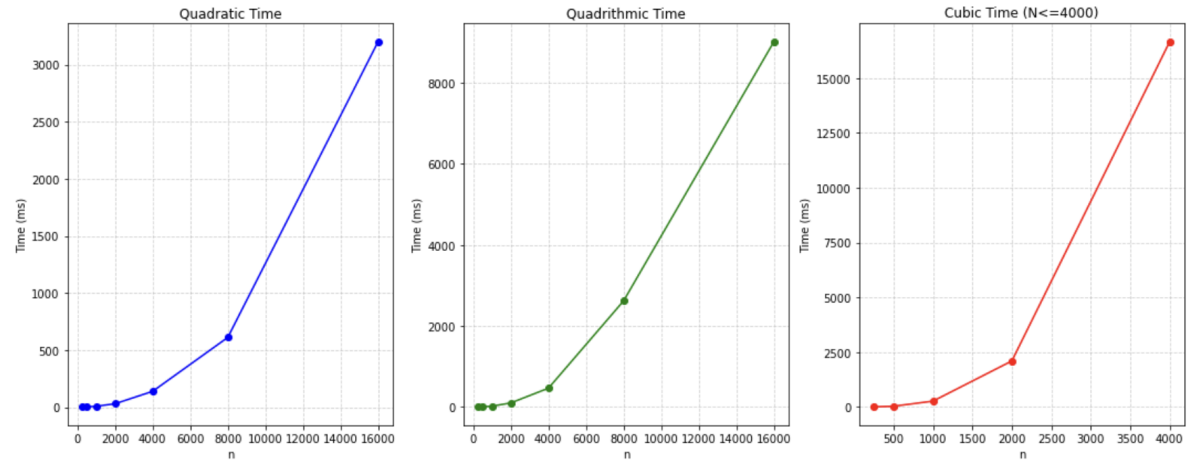
E8

	A	B	C	D	E	F	G	H
1	TIME\N	250	500	1000	2000	4000	8000	16000
2	Quadratic	2.901	2.546	7.691	30.11	138.753	610.819	3199.513
3	Quadrithmic	2.636	3.938	20.201	101.067	464.071	2618.337	9012.193
4	Cubic	5.688	35.094	268.386	2101.518	16663.997		
5			BY RAHUL POTHIRENDI - 002889957					
6								

GRAPHICAL OUTPUT

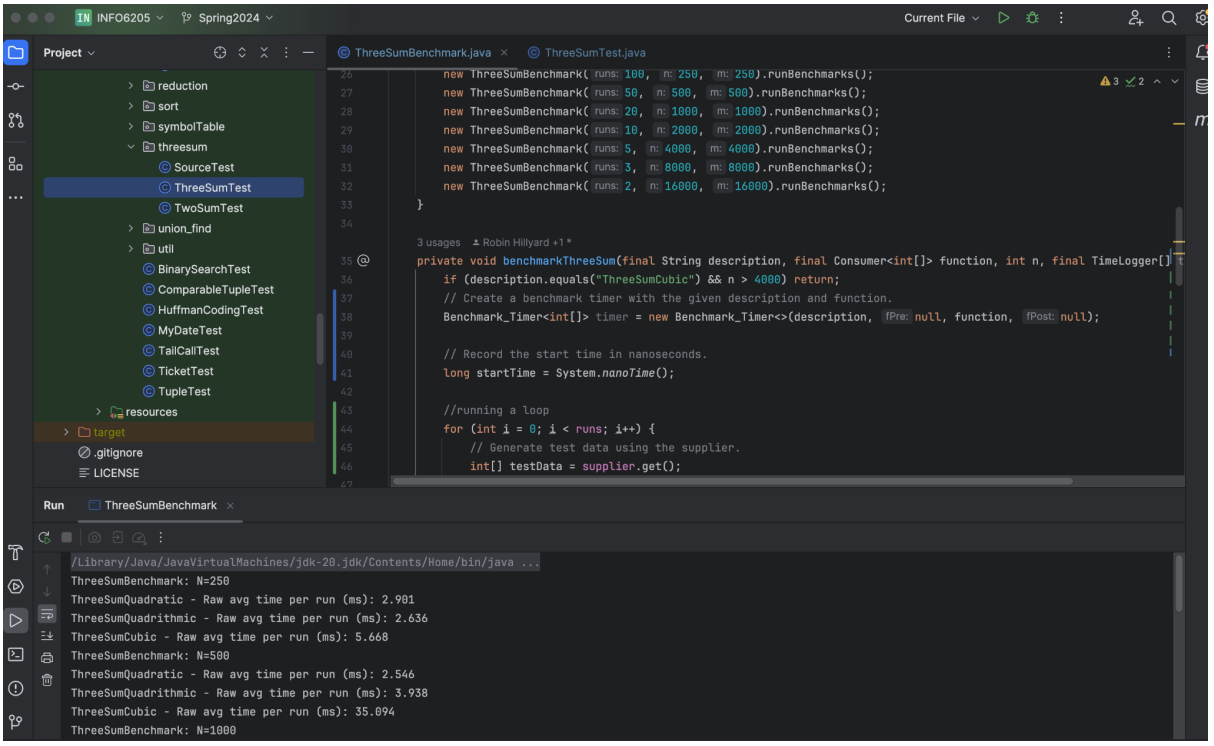


(a)



(b)

Code simulation



Explanations

Used system.nanotime() method to take the precise timestamps in nanoseconds, start to end of algorithm execution

ThreeSumQuadratic:-
The algorithm has a quadratic time complexity, meaning the running time grows proportional to the square of the input size (n). In the provided code, the time complexity is expressed as "Normalized time per run (n^2): ".

Output Analysis - As N increases, the raw average time per run also increases, and the growth is proportional to N^2. This is consistent with a quadratic time complexity

ThreeSumQuadrithmic:
The algorithm has a quadrithmic time complexity, which is a combination of quadratic (n^2) and logarithmic (log n) terms.

Output Analysis
As N increases, the raw average time per run increases, and the growth is proportional to N^2 log N. This indicates a higher growth rate compared to the quadratic algorithm.

ThreeSumCubic:-
The algorithm has a cubic time complexity, meaning the running time grows proportional to the cube of the input size (n).

CODE -

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

import edu.neu.coe.info6205.util.Benchmark_Timer;
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;
        // Create a benchmark timer with the given description and
function.
        Benchmark_Timer<int[]> timer = new
Benchmark_Timer<>(description, null, function, null);

        // Record the start time in nanoseconds.
        long startTime = System.nanoTime();

        //running a loop
        for (int i = 0; i < runs; i++) {
            // Generate test data using the supplier.
            int[] testData = supplier.get();

            // Execute the algorithm with the generated test data.
            function.accept(testData);
        }

        // Record the end time in nanoseconds.
        long endTime = System.nanoTime();
        long totalTime = endTime - startTime;
        // Calculate the average time per run in milliseconds.


        //Convert total execution time from nanoseconds to
milliseconds and divide by the number of runs.
        double averageTimeMs = (totalTime / 1e6) / runs;

        //here we are checking the raw time
        System.out.println(description + " - Raw avg time per run
(ms): " + String.format("%.3f", averageTimeMs));

    }

    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;
}
```