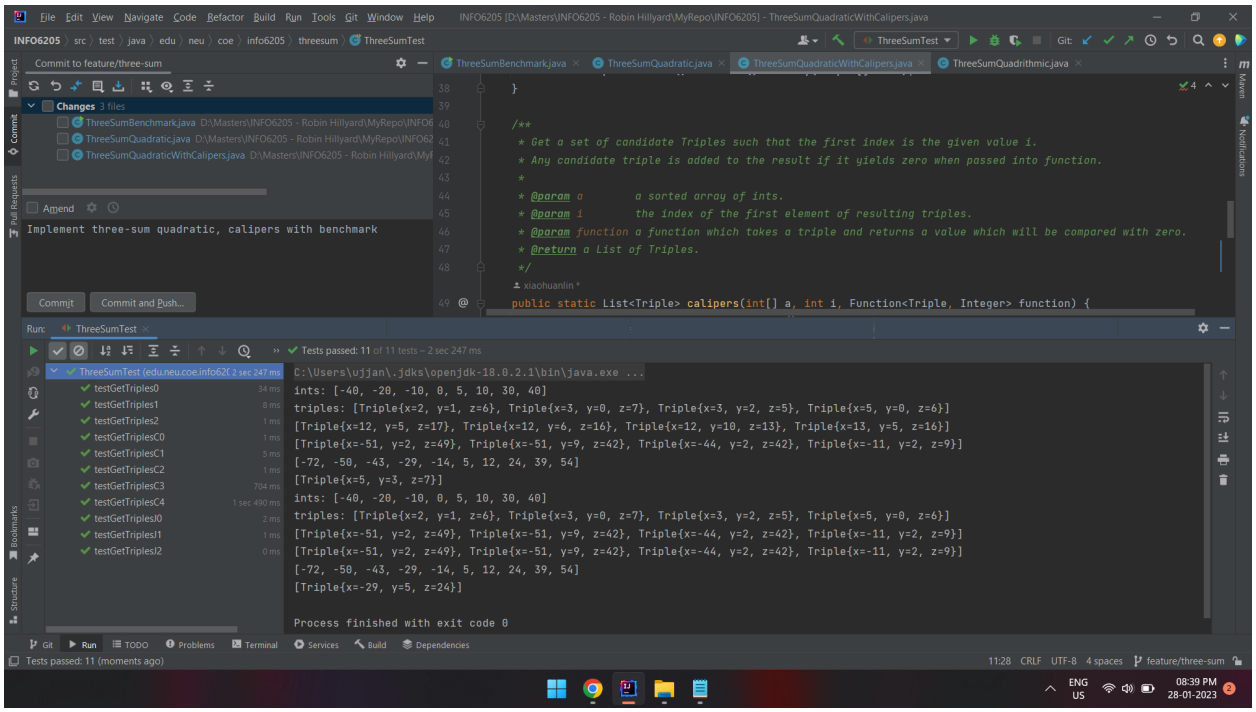
Ujjanth Arhan
002108348

Part a: Unit test cases with git changes

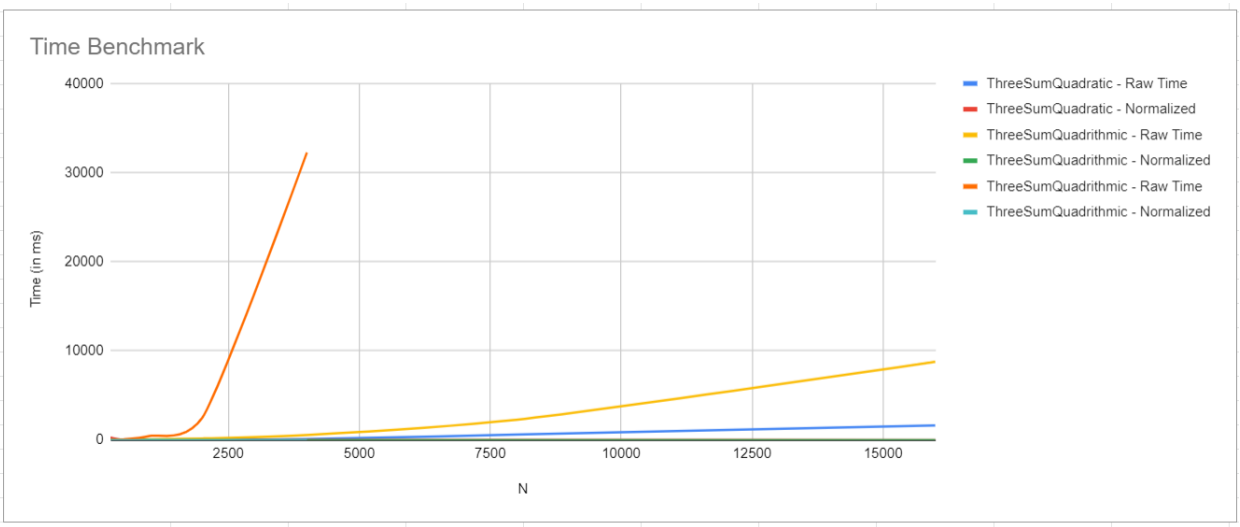


Part b:

a. Time benchmark table

	ThreeSumQuadratic		ThreeSumQuadrithmic		ThreeSumCubic	
	n^2		n^2 log n		n^3	
N	ThreeSumQuadratic - Raw Time	ThreeSumQuadratic - Normalized	ThreeSumQuadrithmic - Raw Time	ThreeSumQuadrithmic - Normalized	ThreeSumQuadrithmic - Raw Time	ThreeSumQuadrithmic - Normalized
250	17	272	10	20.09	26	1.66
500	9	36	13	5.8	79	0.63
1000	69	69	69	6.92	467	0.47
2000	52	13	188	4.29	2485	0.31
4000	117	7.31	577	3.01	32251	0.5
8000	617	9.64	2286	2.75		
16000	1644	6.42	8778	2.46		

b. Graph



Part c:

a. Explanation of quadratic method

Overview:

The method works by using a simple approach that involves usage of three variables. The first variable will iterate from the mid variable till the first index. The third variable will iterate from the mid variable till the last index. If the sum of these values total to 0, it is added to the list of tuples.

Procedure and complexity:

This method works on an array which is already sorted. The array is then iterated by using a combination of two inner loops. The outer loop acts as a point around which the iteration occurs. The inner loop is used for the purpose of manipulating the two pointers. The first pointer points to one less than the i^{th} index and the the other pointer points to one greater than the i^{th} index. The outer pointers head towards the extremities while the inner one stays at the same location and the process repeats for the entire length of the array.

The procedure uses two nested loops and each statement involves steps related to initialization, modification of the values, comparisons or addition to the list.

The outer loop runs from 0 to length - 1. For each iteration of the outer loop, the inner loop runs for length - 1 times. Since each operation can be considered that they can be completed in $O\ (1)$ time, it's the total comparison which is responsible for time complexity. So, for every outer loop iteration, the inner loop runs for (N - 1) * (N - 2)... 1 time. Since the majority of the time taken is because of the repeated execution of the statements in the nested loop, the total time complexity becomes $O\ (N^2)$.

b. Explanation of quadratic with calipers method

Overview:
The method works by using a simple approach that involves usage of three variables. The first variable will be fixed and the other two variables iterate around it. The second variable will iterate from the $i + 1$ to the end of the array and the third variable will iterate from $length - 1$ to 0. If the sum of these values total to 0, it is added to the list of tuples.

Procedure and complexity:
The array is iterated by using a combination of two inner loops. The outer loop values act as a pivot for the iteration. The inner loop is used for the purpose of manipulating the two pointers. The first pointer points to one greater than the i^{th} index and the other pointer points to one lesser than the length. The left pointer heads towards the end of the array while the other one moves from the direction of the end of the array to 0. The process repeats for the entire length of the array.
The procedure uses two nested loops and each statement involves steps related to initialization, modification of the values, comparisons or addition to the list. These operations can be considered that they can be completed in $O(N)$ time.
So, for every outer loop iteration, the inner loop runs for (N - 1) * (N - 2)... 1 time. Since the majority of the time taken is because of the repeated execution of the statements in the nested loop, the total time complexity becomes $O(N^2)$.

	ThreeSumQuadratic		ThreeSumQuadrithmic		ThreeSumCubic	
	n^2		n^2 log n		n^3	
N	ThreeSumQuadratic - Raw Time	ThreeSumQuadratic - Normalized	ThreeSumQuadrithmic - Raw Time	ThreeSumQuadrithmic - Normalized	ThreeSumQuadrithmic - Raw Time	ThreeSumQuadrithmic - Normalized
250	17	272	10	20.09	26	1.66
500	9	36	13	5.8	79	0.63
1000	69	69	69	6.92	467	0.47
2000	52	13	188	4.29	2485	0.31
4000	117	7.31	577	3.01	32251	0.5
8000	617	9.64	2286	2.75		
16000	1644	6.42	8778	2.46