Program Structures and Algorithms

Spring 2023 (Section 3)

Assignment 2 – Three Sum

Name: Abhinav Choudhary

NUID: 002780326

Task

- To solve "Three Sum" problem with different algorithms with differing time complexities and time each algorithm with increasing values of N (size of input array) to find out which algorithm completes in the least time.
- In the "Three Sum" problem, given an array of numbers (positive and negative), we must find triplets or set of three numbers whose sum equal to 0. These triplets should be unique. Finally, we count or display the unique triplets for the given array.
- This problem can be solved using multiple algorithms. Brute force algorithm (cubic method) which gives time complexity of O(n^3), Quadrithmic algorithm with complexity of O($n^2 logn$), and Quadratic (with and without calipers) approach with complexity of O(n^2).

Relationship Conclusion

By running all the algorithms for increasing values on N multiple times, we found out that the quadratic method (without calipers) performs the best O(n^2). Followed by quadratic (with calipers) method O(n^2), Quadrithmic method O($n^2 logn$), and finally cubic method O(n^3). However, for small values of N (less than 1000 elements), quadratic (with calipers) seems to perform the best.

Quadratic (without calipers) < Quadratic (with calipers) < Quadrithmic < Cubic

Evidence to support conclusion

All the different algorithms work in the following ways:

- **Cubic**
  - In this method, we have three loops, one inside the other (outer i loop -> inner j loop -> inner k loop), we increment values of each loop and check with all the elements of the input array one by one to find out the triplets.
- **Quadrithmic**
  - In this approach, we sort the array first and then run two loops (outer i loop -> inner j loop). After that, we find out the sum of elements at i and j positions and then we use binary search to find the third element whose value should be equal to the sum of the first two elements, with the opposite sign (positive or negative).
- **Quadratic (without calipers)**
  - In this approach, we sort the input array and select an element from the middle of the array. After that, we set up two pointers, one at the leftmost point (or start) and the other at the rightmost point (or end) of the array. We check if the sum of elements at the leftmost point, middle, and rightmost point are equal to 0 or not. If

not, we check if the resultant sum is greater or smaller than 0. If greater than 0, we decrement the pointer at the rightmost point and vice versa for smaller than 0.

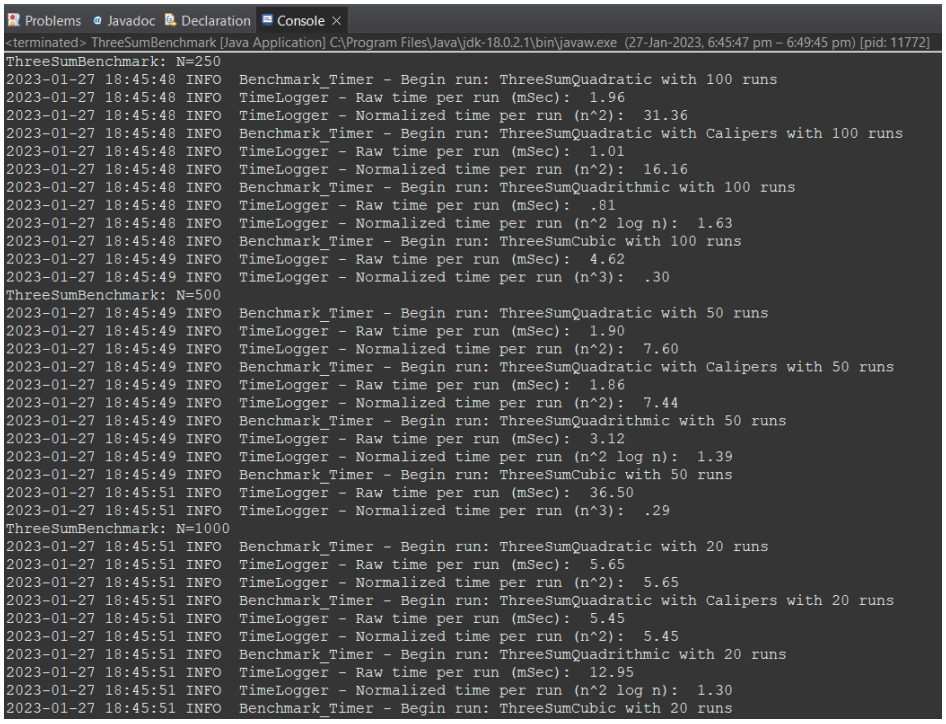
- **Quadratic (with calipers)**
    - In this method, we sort the input array and select an element from the start of the array. After that, we set up two pointers, one adjacent to the initial pointer and the other at the rightmost point (or end) of the array. We check if the sum of elements at all the three points equal to 0 or not. If not, we check if the resultant sum is greater or smaller than 0. If greater than 0, we decrement the pointer at the rightmost point and vice versa for smaller than 0.

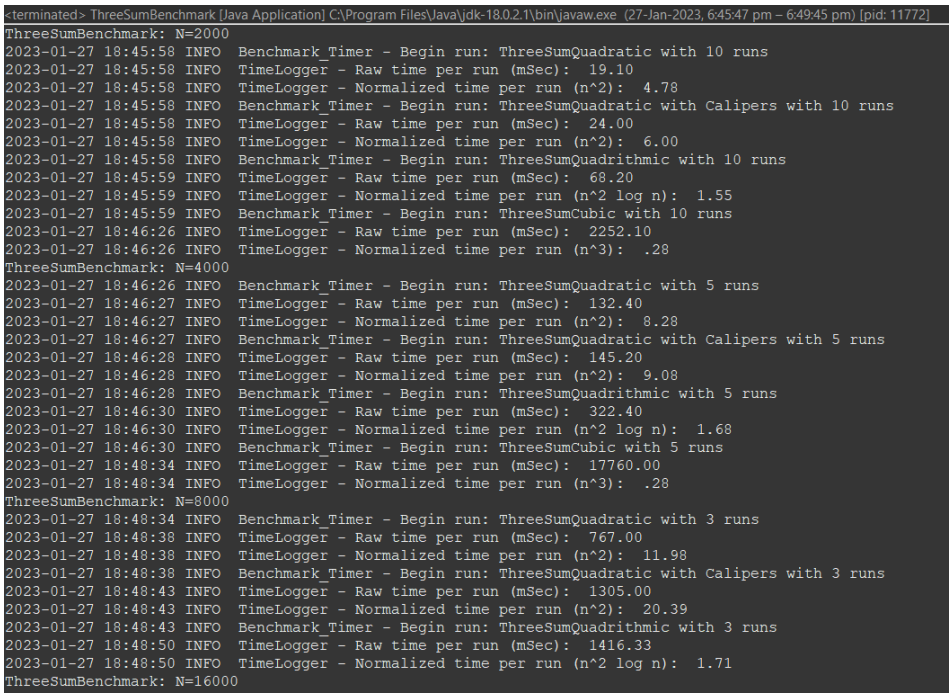
After running all the four algorithms for increasing values of N (size of input array) and for multiple runs, following raw run time (in milliseconds) were noted:

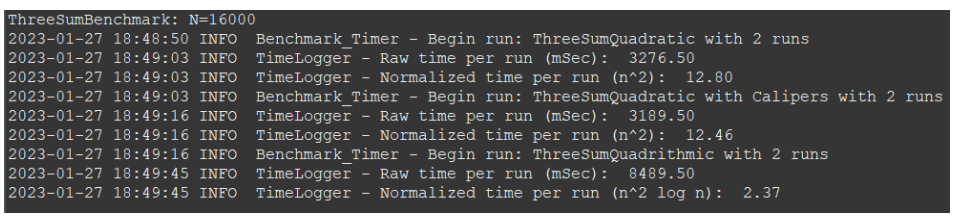
N	**Runs**	**Time (in ms) - Quadratic**	**Time (in ms) - Quadratic with Calipers**	**Time (in ms) - Quadrithmic**	**Time (in ms) - Cubic**
250	100	1.96	1.01	0.81	4.62
500	50	1.9	1.86	3.12	36.5
1000	20	5.65	5.45	12.95	289.45
2000	10	19.1	24	68.2	2252.1
4000	5	132.4	145.2	322.4	17760
8000	3	767	1305	1416	
16000	2	3276.5	3189.5	8489.5	

In this table, we can clearly see that at the beginning (or small values of N), quadratic with calipers perform the best with quadratic without calipers and quadrithmic not that far behind, whereas cubic has the highest time by a large margin from the start. At around N=1000 mark, we can see that quadratic without calipers has the best raw run time with quadratic with calipers close and quadrithmic and cubic far ahead. This trend can be seen for the rest of the values of N, where quadratic with and without calipers stay close to each other and perform the best, with cubic taking the most time and quadrithmic somewhere in the middle.

Following are the screenshots from the benchmark:





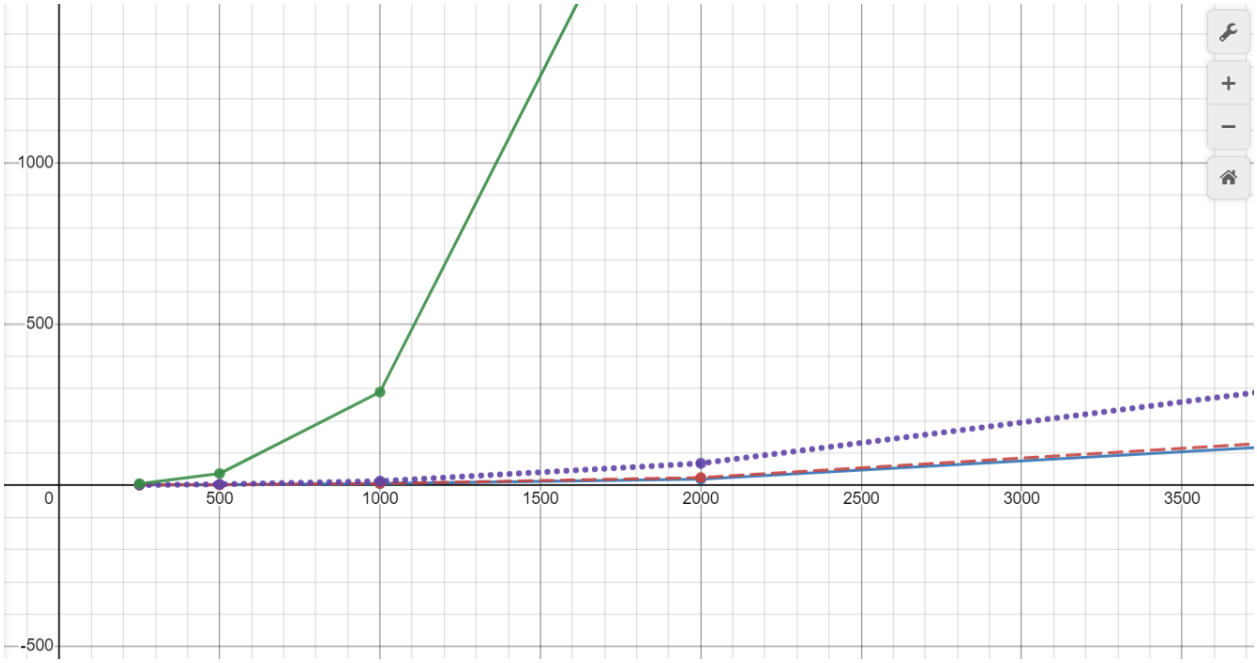


With these timing observations, we can conclude that for large values of N, quadratic without calipers is the best algorithm followed by quadratic with calipers, quadrithmic, and cubic.

Graphical Representation

Following is the graph plotted using the timing observations, with time (in ms) along the y axis and N along the x axis. Solid blue line represents quadratic without calipers, red dashed line indicate quadratic with calipers, purple dotted line indicates quadrithmic, and green solid line represents cubic.

(Desmos Graphing Calculator was used to plot the points and create the graph: Desmos | Graphing Calculator)



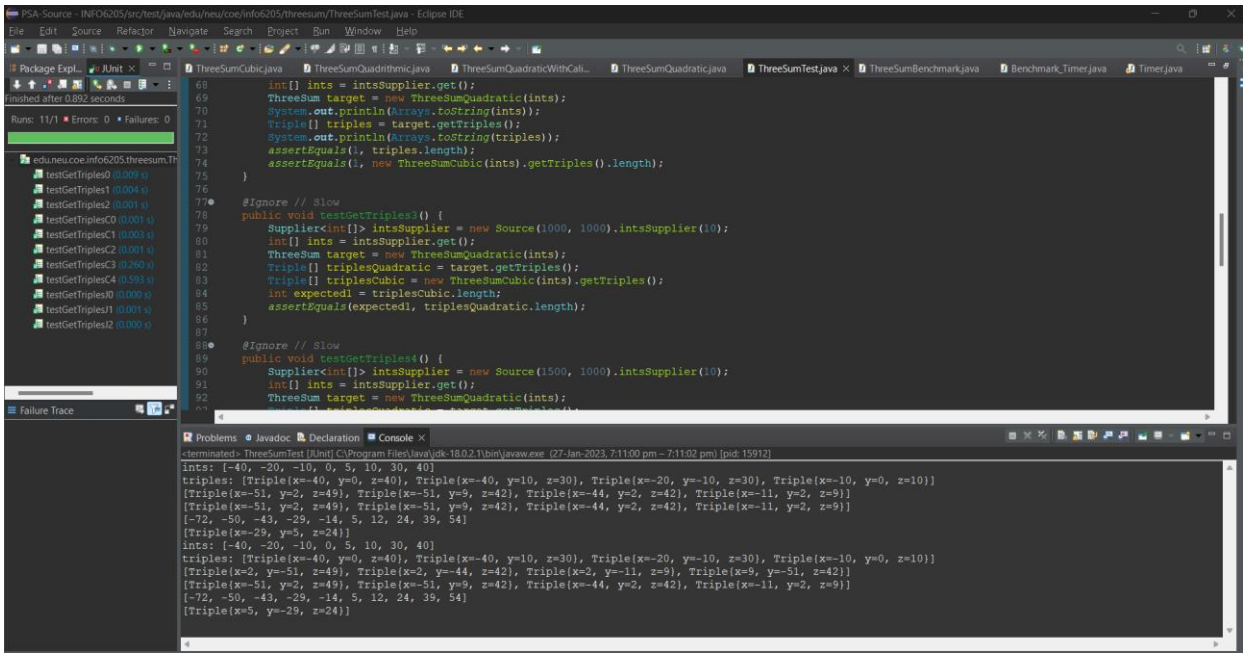
Why the Quadratic method work

- The quadratic method works because we already have a sorted array that means all the smaller negative values would be at the leftmost side or start of the array and all the positive values would be towards the rightmost side or end of the array.
- We try to set an initial pointer at the middle and then set two additional pointers at both the ends. One outer loop is used to maintain the middle pointer and a second inner loop can be used to maintain both the remaining pointers.
- When we take the sum of all the three points and it is equal to 0, we save the triplet and increment the leftmost pointer by one and decrement the rightmost pointer by one.
- If the resultant sum is greater than 0, we know that there is a larger value towards the rightmost part which is shifting the overall result in the positive side. In this case, we check with a smaller positive value by decrementing the rightmost pointer. As the array is sorted we know that the element at the immediate left of the element at rightmost pointer is smaller than it.
- If the resultant sum is smaller than 0, we know that there is a large negative number (taking absolute value) at the leftmost part, which is shifting the overall result towards the negative side. In that case, we increment the leftmost pointer because as we know the array is sorted the number to the immediate right of the number at the leftmost pointer would be larger (or would have a smaller absolute value).
- Thus, with the knowledge of how the numbers are arranged, we can reduce the overall number of loops and we would not have to check for every value of the input array one by one significantly bringing down the run time.

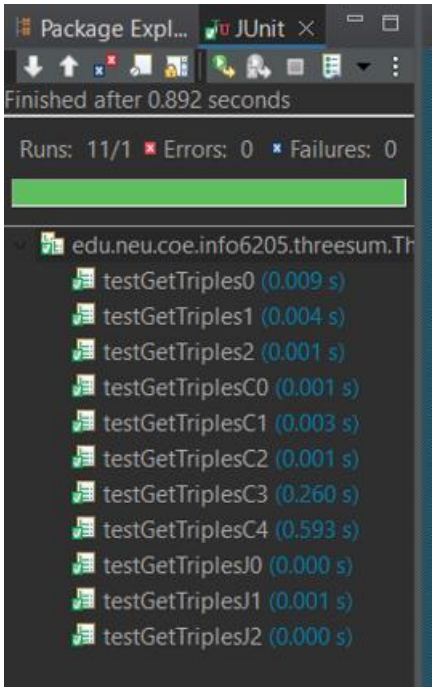
Unit Test

All the unit tests passed taking approximately 0.892 seconds. ThreeSumTest.java file was not edited for this test.

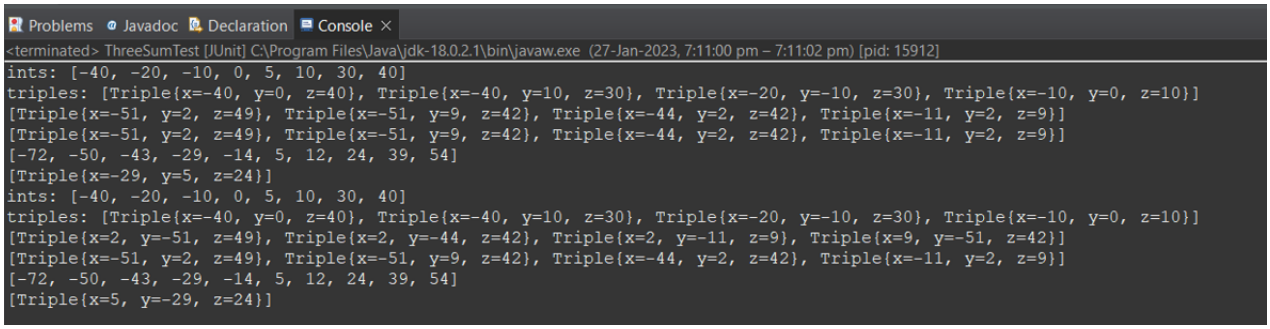
Following is the screenshot containing part of ThreeSumTest.java file code alongside test cases at the left and console output at the bottom.



Screenshot of unit tests passing:



Screenshot of console output:


Problems
Javadoc
Declaration
Console
<terminated> ThreeSumTest [JUnit] C:\Program Files\Java\jdk-18.0.2.1\bin\javaw.exe (27-Jan-2023, 7:11:00 pm – 7:11:02 pm) [pid: 15912]
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}]
[Triple{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}]
[Triple{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]
[Triple{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}]
[Triple{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}]
[Triple{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]
[Triple{x=5, y=-29, z=24}]