Program Structures and Algorithms

Spring 2023(SEC 01)

NAME: Shivani Datar

NUID: 002772160

**Task:**

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

(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.

**Relationship Conclusion:**

After running 3-Sum code with different complexities, with the observed data we can see that Quadratic and Quadratic with Calipers are better in time complexity O(n²). Later on we have Quadrithmic with O(n² log n) and lastly cubic with O(n^3). We can verify this by using the time calculated for each execution using the Benchmark\_Timer class.

3-sum Quadratic :-

Here we already have a middle index of one of the element whose sum is 0. We are using a while loop with 2 pointers such that the first one would be less than provided index and the second one would be more than the provided index. In this subspace we use complexity of O(N) and the function which is calling this subspace has a for loop giving us a quadratic time complexity.

Text

Description automatically generated

3-Sum Quadrithmic:-

Here we already have a pair a[i] and a[j], we only need to find the third element, which is done by using the BinarySearch. This is divided into 2 subspaces, In first subspace we fix both elements a[i] and a[j] giving us a O(n²) time complexity. The second subspace has a complexity for O(log n) as we are using binary search to find the element. The total time taken here is quadrithmic or O(N^2 log n).

Text

Description automatically generated

3-Sum Quadratic with Calipers :-

In this scenario we have the first index out of the three elements, here as well we use two pointers method to calculate remaining two elements but the computation is less heavy as we know the first element’s index and it’s a sorted array we know for sure that the remaining two elements are after the first element’s index. Here we are provided a function, which we use to compare the sum with 0.

Text

Description automatically generated

Changes in ThreeSumBenchmark file :-

In the benchmark file we have different N values and a supplier which helps us to construct a random array given a seed for particular N and we also have total number of runs performed on the constructed array. This is approached by initializing a object of Benchmark\_Timer class. This class has 3 functions fpre,frun and fpost as properties. Fpre is a function which is supposed to be executed before we run our function. fRun is a function which we actually want to run and fPost is some function if we want to run later on the main executed function. In our case we just to run the getTriples function from each different complexity 3-Sum class, hence our fpre and fpost are null. After initializing the object, we call runfromsupplier method, where we are passing the description and a lambda function, this is sent to the new runtime initialized object of Timer class, using the repeat method we can get the time utilised by our function to run in miliseconds. We later on use the timeloggers to print time in mili seconds and normalized time with the time complexity observed.

Text

Description automatically generated

Output for the benchmark code :-

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

**Evidence to support that conclusion:**

The following table is a consolidated count of the time required for each algorithm with respective amount of N. The 3-sum cubic algorithm for n = 16000 took lot of time. It was in process for more that 1 and half hour and then I exited the process by stopping the code.

Table :-

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | Quadratic(ms) | Quadrithimic(ms) | Cubic(ms) | QuadracticCaliper(ms) |
| 250 | 0.91 | 0.92 | 6.2 | 0.91 |
| 500 | 1.81 | 3.4 | 47.24 | 0.83 |
| 1000 | 7.41 | 17.27 | 373.85 | 4.88 |
| 2000 | 27.32 | 93.79 | 2976.02 | 20.19 |
| 4000 | 161.77 | 467.35 | 23723.9 | 146.44 |
| 8000 | 796.82 | 2068.53 | 189171.99 | 616.52 |
| 16000 | 3333.64 | 8953.62 |  | 3419.62 |

**Graphical Representation:**

Graph for Quadratic:-

Observations:-

|  |  |
| --- | --- |
| N | Quadratic(ms) |
| 250 | 0.91 |
| 500 | 1.81 |
| 1000 | 7.41 |
| 2000 | 27.32 |
| 4000 | 161.77 |
| 8000 | 796.82 |
| 16000 | 3333.64 |

Chart, line chart

Description automatically generated

Graph for Quadrithmic:-

|  |  |
| --- | --- |
| N | Quadrithimic |
| 250 | 0.79 |
| 500 | 3.23 |
| 1000 | 17.26 |
| 2000 | 86.9 |
| 4000 | 476.3 |
| 8000 | 2204.5 |
| 16000 | 9763.8 |

Chart, line chart

Description automatically generated

Graph for Quadratic with calipers :-

|  |  |
| --- | --- |
| N | QuadracticCaliper(ms) |
| 250 | 0.91 |
| 500 | 0.83 |
| 1000 | 4.88 |
| 2000 | 20.19 |
| 4000 | 146.44 |
| 8000 | 616.52 |
| 16000 | 3419.62 |

Chart, line chart

Description automatically generated

Graph for cubic:-

|  |  |
| --- | --- |
| N | Cubic |
| 250 | 6.12 |
| 500 | 48.16 |
| 1000 | 374.27 |
| 2000 | 2977.68 |
| 4000 | 23750.67 |
| 8000 | 189061.88 |
| 16000 |  |

Chart, line chart

Description automatically generated

Graph For Complexity: -

Graphical user interface

Description automatically generated with medium confidence

**Unit Test Screenshots:**

Following is the screenshot of all 12 test cases clearing.

Graphical user interface, text

Description automatically generated