Assignment 1: 3SUM

Alberto Beyersdorff

2023-09-20

1 Introduction

The 3SUM problem involves finding a triplet of distinct integer values in a given list that adds up to zero, represented as $a + b + c = 0$, where a, b, and c belong to the set of integers. Our goal was to create an experimental environment to compare various implementations of the 3SUM algorithm in terms of their processing time. Additionally, we extended the scope to the 4SUM problem, where we seek a quartet of integers (a, b, c, d) in the list such that $a + b + c + d = 0$, further expanding the computational challenge.

2 Implementation

Implemented in Java via a Gradle project, the tasks involved four methods for 3SUM and three for 4SUM. Specifically, for 3SUM, three distinct approaches — Cubic, Quadratic, and HashMap — were explored. The cubic algorithm is typically solved using three nested loops, resulting in a cubic time complexity $O(n^3)$, where n is the number of elements in the input array. For the quadratic algorithm, the idea is to first sort the list of numbers. Once we have a sorted list, we use three pointers: a starts from the beginning, b just after a, and c from the end of the list. This approach efficiently finds triplets with a time complexity of $O(n \log n + n^2)$. Finally, the HashMap approach used here is to create a hash map where each element of the input list is stored along with its index in the original list. This approach utilizes a hash map for efficient $O(1)$ average time complexity lookups. However, due to nested iteration over each pair of input elements, the overall time complexity remains $O(n^2)$.

An additional method was included in the class: the implementation of the HashMap approach without the $j < k$ condition. This condition prevents an element from being counted twice, ensuring that the third element must

exist in the remainder. This condition ensures that an element can not be considered twice, i.e. that the third element must be in the remainder. Furthermore, there are three approaches to the 4SUM problem, all of these following the same strategies as for the 3SUM. The main method is only responsible for reading the arguments (name, size of the list, list elements), calling the respective method, and printing the result, either the elements summing up to zero or *null*. For the method names the following naming pattern was chosen, *t_name* for 3SUM and *f_name* for 4SUM, i.e. *t_cubic* representing the cubic algorithm for the 3SUM problem.

To ensure comprehensive testing, we implemented two classes covering diverse scenarios: (1) no triplet/quadruplet, (2) one triplet/quadruplet, (3) multiple triplets/quadruplets, (4) fewer elements than expected, (5) empty list, (6) duplicate values, (7) only negative integers, and (8) special cases of HashMap. Notably, two cases, namely [0, 0] and [3, -6, 1], revealed differences between the standard HashMap approach and the version without comparison. All tests were executed using JUnit, incorporating assertions to validate method outputs against expected results.

3 Experiments

In the experiment, Python 3.10 executed Java 17.0.6 JAR files, measuring the runtime of three 3SUM implementations (Figure 1) as n increased, with multiple repetitions to mitigate measurement fluctuations. The setup used a macOS laptop with a Dual-Core Intel Core i7 processor and 16 GB RAM. Post-processing involved calculating mean and standard deviation to interpret runtime results from the experiment output.

Test scenarios encompassed lists of varying sizes, with n growing by a factor of the square root of 2 from 30 onwards. Results indicate that the quadratic solution reached a maximum of 320,654 elements before timing out, while the cubic approach, as detailed in Table 1, managed 7,321 elements, and the hashMap-based solution coped with 57,536 elements. As n increased, all three implementations exhibited exponential growth, emphasizing the computational complexities when scaling the problem size.

In the extended experiments covering the 4SUM problem (Figure 2), several notable findings emerged. Unlike the 3SUM problem, the hashMap implementation consistently faced memory limitations (Java heap space) even with increased memory allocation. Interestingly, the quartic solution encountered a timeout much earlier than the others, capping at 931 elements, while the cubic and quadratic implementations exhibited similar performance, timing out at 5,192 elements. Moreover, an intriguing shift occurred as list size

Table 1: Results of the cubic solution.

n	Average (s)	Standard deviation (s)
30	0.100986	0.007270
42	0.101432	0.013393
59	0.101678	0.005566
84	0.104181	0.002235
118	0.120274	0.006068
167	0.114391	0.005615
235	0.110635	0.005927
332	0.131344	0.024859
468	0.143811	0.012734
660	0.185039	0.039915
931	0.194152	0.012537
1313	0.291307	0.015729
1852	0.522156	0.001126
2611	1.157966	0.029947
3682	2.926966	0.023365
5192	7.890010	0.081953
7321	21.670260	0.142357

increased: the cubic algorithm surpassed the hashMap approach in efficiency with larger lists, indicating a shift in relative performance.

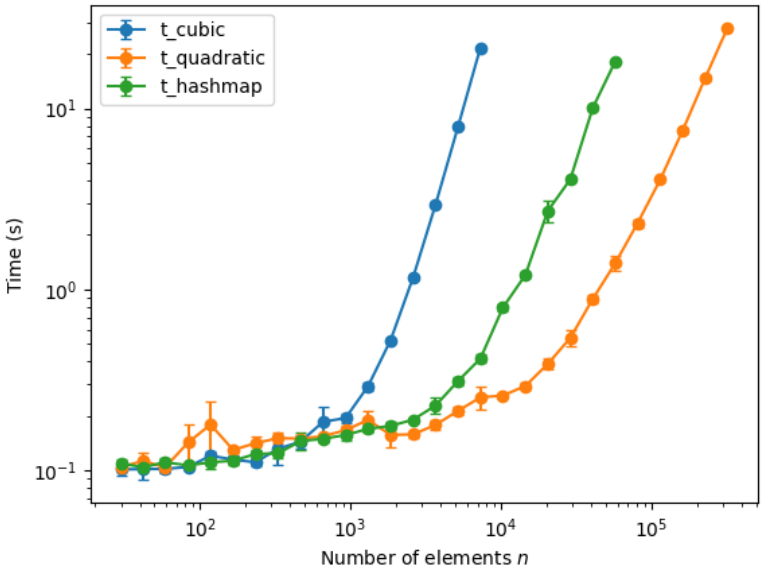

Figure 1: Comparison of runtime across different 3SUM implementations.

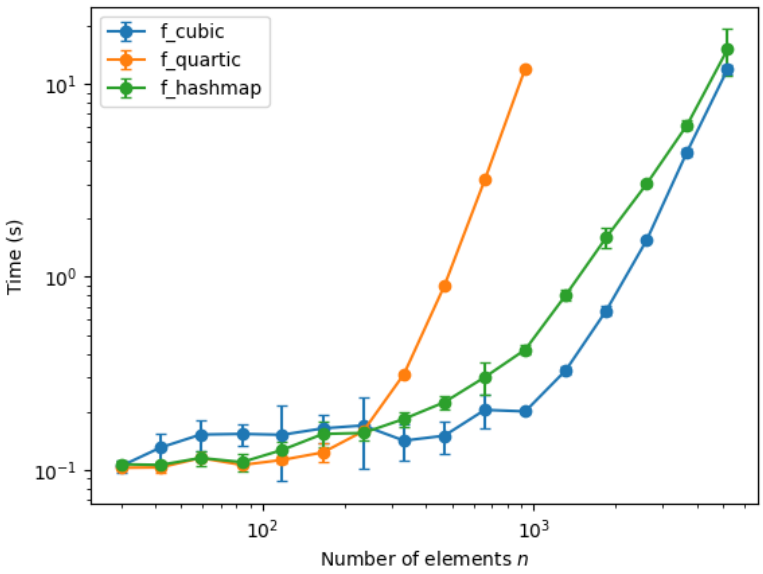

Figure 2: Comparison of runtime across different 4SUM implementations.