The 3-sum Report

Aditya Nair

Department of Computer Science, Linnaeus University
an224za@student.lnu.se

I. Introduction

This report deals with the 3-sum problem and various approaches to handle this problem. It also involves experiments evaluating and comparing different approaches.

A. Problem Formulation

3-sum Problem: Given a list of integers $n_1, n_2, n_3, ..., n_p$ and an integer s, find all unique pairs n_i, n_j, n_j $(i \neq j \neq k, n_i \leq n_j \leq n_k)$such that $n_i + n_j = s$.

Returns: List of unique pairs (n_i, n_j, n_k) where $n_i \leq n_j \leq n_k$. This formulation excludes duplicate pairs and (e.g. $(9, 2, 1)$ and $(2, 9, 1)$) having the same three numbers. All algorithms discussed here have been thoroughly tested, using both automated testing and manual verification. Input lists and expected answers were sourced from LeetCode.com for reliability. Short lists were also used for manual testing to confirm accuracy.

Input: [-1, 0, 1, 2, -1, -4]
Matches: [(-1, 0, 1), (-1, -1, 2)]

B. Experimental Setup

All experiments were performed on the same computer: a MacBook Air from 2023 with an Apple M2 processor, equipped with 16 GB of memory and running macOS Sonoma (version 14.7). During the experiments, we minimized external disturbances by closing all unnecessary applications on the computer.

Each time measurement is the average of five repeated runs. The input size was chosen to ensure the running time was between 0.01 and 10 seconds. For a given size y, the 3-sum input is a list of y random integers ranging from [0, 1000]. The target value was always 0, meaning we looked for triplets n_i, n_j, n_k such that $n_i+n_j+n_k=0$.

Brute Force

This approach aims to identify all unique triplets (ni,nj,nk)(n_i, n_j, n_k)(ni,nj,nk) in a list whose sum matches a specified target value (in our case, 0). The brute-force method exhaustively examines every possible combination of three elements from the list to find valid triplets.

Figure 1 presents the Python implementation used in our experiments. The algorithm employs three nested loops to traverse the list and generate all potential triplets. For each combination, it checks whether the sum of the elements equals the target value. To avoid counting duplicates, the algorithm applies two main techniques. First, each triplet is sorted in ascending order (as shown in Figure 2), which standardizes the representation of triplets—so variations like (1, -1, 0) and (0, -1, 1) are considered equivalent.

```python
def three_sum_brute_force(arr):  2 usages
    n = len(arr)
    triplets = []
    for i in range(n):
        for j in range(i + 1, n):
            for k in range(j + 1, n):
                if arr[i] + arr[j] + arr[k] == 0:
                    triplet = tuple(sorted((arr[i], arr[j], arr[k])))
                    if triplet not in triplets:
                        triplets.append(triplet)
    return triplets

```

Fig. 1
THE BRUTE FORCE SOLUTION TO THE 3-SUM PROBLEM

The algorithm iterates through the list using three loops. The outer loop iterates with index i, the middle loop with index j (where $j \neq i$), and the inner loop with index k (where $k \neq i$ and $k \neq j$).

The time complexity of the brute force solution is $O(n^3)$ due to the three nested loops, each iterating over n elements.

C. Brute Force Experiments

By running experiments with various list sizes we found that sizes in range 500 to 1500 (step 100) gives reasonable running times (in range 0.01 to 70 seconds). Next, in order to visually confirm that our implementation behaves

as expected we run three separate experiments to see if repeated runs give approximately the same outcome. The result of this experiment is shown in Figure 3.

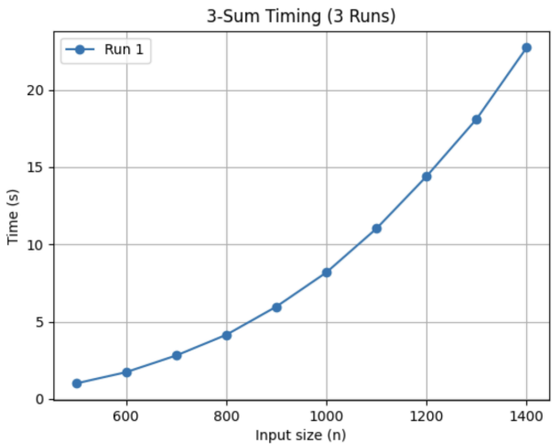

Fig. 2
THREE REPEATED RUNS WITH THE BRUTE FORCE SOLUTION

The three runs are very similar, no outliers, and show the upward bend we can expect from an $O(n^3)$ algorithm.

Method to figure out the Time complexity:

1. Plot the average runtime against the input size on a log-log scale. Fit a linear regression line to the plotted points. The slope of this line will approximate the value of k in the assumed time complexity $O(n^k)$.

For the brute force Three Sum algorithm, we expect k to be approximately 3.

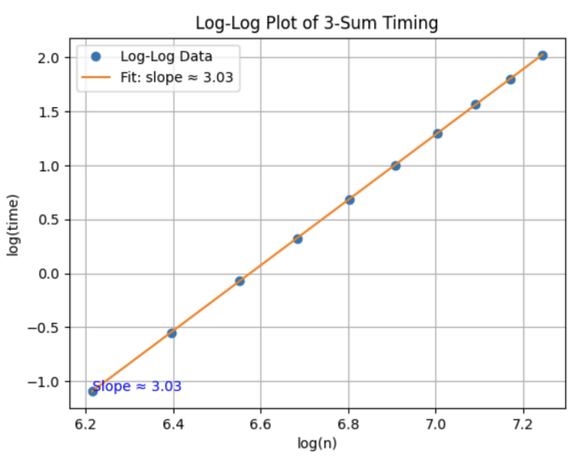

Figure 3

The top figure shows the average execution time over five runs. As expected, the trend exhibits a smooth upward curve, consistent with the increasing cubic complexity of the Brute Force algorithm. There are no significant outliers, confirming the reliability of our results.

The bottom figure shows log-log markers represent the relationship between *log(size)* and *log(time)*. The nearly straight-line trend validates our assumption of a polynomial time complexity $O(n^k)$.

The red line represents a straight-line fit using linear regression. The fit is strong, indicating that our data aligns well with the assumed polynomial model. Linear regression yields a coefficient k=3.03, which is sufficiently close to 3, confirming that the implementation of the 3-Sum Brute Force algorithm has a time complexity of $O(n^3)$, as expected.

Conclusion:

In this report, we explored the 3-Sum problem and analyzed multiple algorithmic approaches to solve it, with a specific focus on the brute-force method. We clearly formulated the problem, described the implementation details, and rigorously tested the algorithm using both automated and manual methods.

Experimental evaluations confirmed the expected cubic time complexity of the brute-force approach. The time measurements were consistent across repeated runs, and log-log analysis further verified the O(n3)O(n^3)O(n3) growth, with a fitted slope of approximately 3.03. These results demonstrate the correctness of our implementation and highlight the inefficiency of brute force for large input sizes.