# 1. Introduction

The Report deals with the 3-Sum problem and how to solve it with a brute force algorithm to solve it. With the drawbacks that come with that implementation.

## A. Problem Formulation

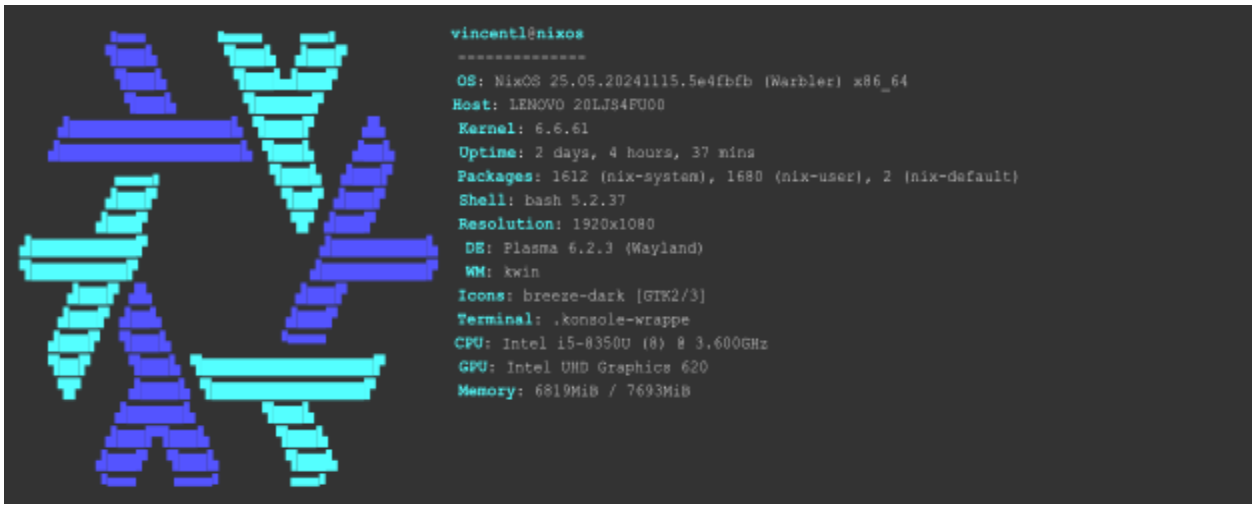
- **3-Sum Problem:**
  Given a list of integers ($n_1$ ,$n_2$ ,$n_3$ , . . . ,n) and an integer s, find all unique pairs $n_i$, $n_j$, $n_k$ (i != j != k, i <= j <= k) such that $n_i + n_j + n_k = s$.
- Returns:
  A list of unique pairs ($n_i$ , $n_j$, $n_k$) where $n_i <= n_j <= n_k$. There by there are no duplicate pairs and pairs with the same value (e.g. (1, 2) and (2, 1)

## B. Experiment setup

The Algorithm was tested on small lists that allow for manual inspection. (all tests were ran with s = 0)

```
Input: [-2, 1, -9, 2, -3, 9, -1, 3, 2, -10]
Output: [(3, -1, -2), (9, 1, -10), (2, 1, -3)]
```

All experiments were ran on this Computer(Lenovo ThinkPad X380 Yoga:




While running the experiments all user facing applications except VS Code were closed and the computer was restarted. For a given size sz the 3-sum input is

always a list of size sz with random integers in the range $[-10 \cdot sz, 10 \cdot sz]$. The target was always 0. That is, we try to find pairs $n_i$, $n_j$, $n_k$ such that $n_i + n_j + n_k = 0$.

## C. Brute Force

The Brute Force solution always tests all possible combinations such that the pairs follow our requirements listed in “A.  Problem Formulation”. This is the Python Code:

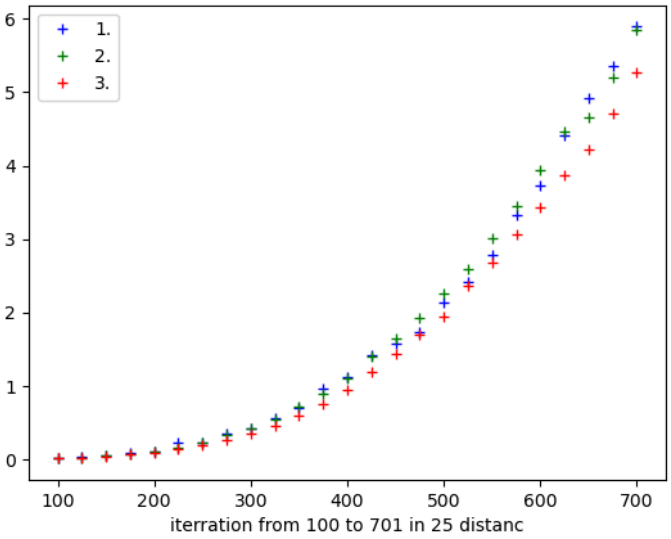
```
def sum_3_brut(lst, s = 0):
   pairs = set()

   for x in range(len(lst) - 2):
       Nx = lst[x]
       for y in range(x + 1, len(lst) - 1):
           Ny = lst[y]
           for z in range(y + 1, len(lst)):
               Nz = lst[z]
               if (Nx + Ny + Nz) == s :
                   if Nx >= Ny and Nx >= Nz:
                       if Ny >= Nz:
                           pairs.add((Nx, Ny, Nz))
                       else:
                           pairs.add((Nx, Nz, Ny))
                   elif Ny >= Nx and Ny >= Nz:
                       if Nx >= Nz:
                           pairs.add((Ny, Nx, Nz))
                       else:
                           pairs.add((Ny, Nz, Nx))
                   else:
                       if Nz >= Ny and Nz >= Nx:
                           if Ny >= Nx:
                               pairs.add((Nz, Ny, Nx))
                           else:
                               pairs.add((Nz, Nx, Ny))
   return pairs
```

The outermost loop begins at the first index and goes to the 3. last index and is the $n_i$. In that loop the 2. one begins 1 index behind j and ends one before the last index and is the j . The innermost loop begins at j + 1 and goes to the end and represents k. Then the

values are sorted from the smallest to the biggest to mitigate duplicate when they are added to the set (pairs). Since there are three nested loops which have the time complexity of O(n) the Brute Force algorithm should have a time complexity of $O(n^3)$.

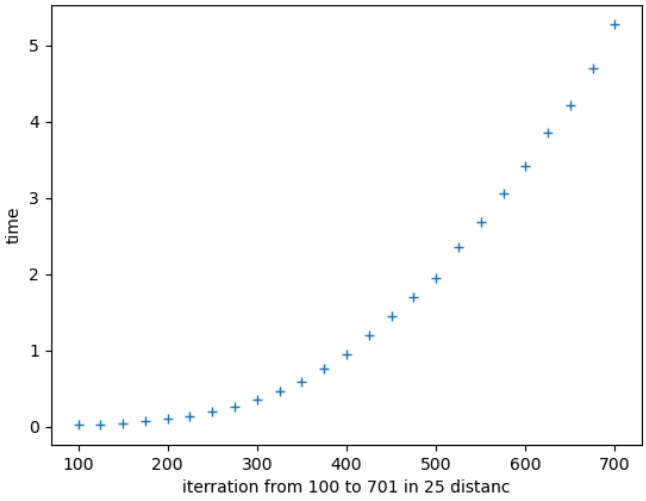
## D. Brute Force Experiments

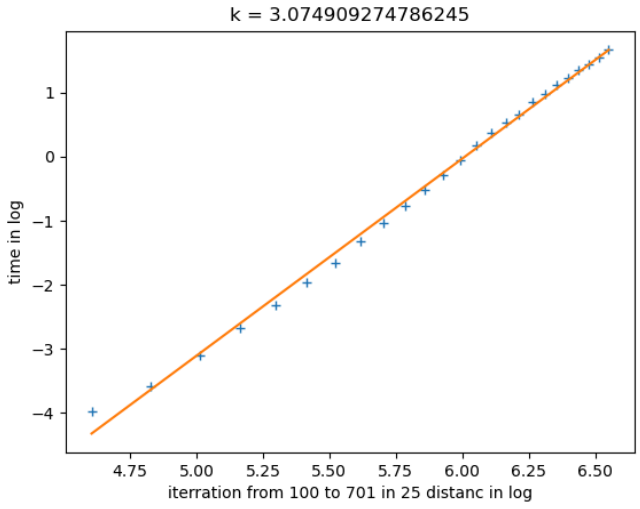
By running experiments with various list sizes we found that sizes in range 100 to 700 (step 25) gives reasonable running times (in range 0.01 to 5 seconds). Next, in order to visually confirm that our implementation behaves as expected we run three separate experiments to see if repeated runs give approximately the same outcome. The result of this experiment is shown in the diagram on the right. The results are not very consistent, but they have a good looking upward curve which we expect from an $O(n^3)$ algorithm.




## E. Time Complexity

To verify the $O(n^3)$ time complexity for the Brute Force algorithm there are two steps:
1. We made a new experiment now computing the average over five separate runs.
2. We use the log-log with linear regression approach outlined in Lecture 2 to find a numerical value for k in our assumption $O(n^k)$. This are the results:



The right figure shows the average of five runs. As (now) expected, a smooth upward bend with no outliers. The blue log-log markers in the right figure shows a straight line which indicates that our assumption of a polynomial behavior $O(n^k)$ is correct. The red line in the bottom figure, a straight line fit using linear regression, seems to be a good fit to the data points. Good! Furthermore, linear regression results in a coefficient k = 3.07 which is close enough to 3 to allow the conclusion that our implementation is indeed $O(n^3)$ as expected.

# 2. Summary and Conclusions

We have taken a deep look into the  Brute Force algorithm for the 3-Sum problem, where we presented an implementation in python, the time complexity $O(n^3)$ which we then proved in an experiment. It was a bit surprising that the consistency was so bad. But that could be because of background tasks. The fact that the computer also after a fresh start already uses 4gb of system memory and 3gb of swap. In addition to that, this computer has some services running on it like a webpage and a few docker containers.  The full code is on github: https://github.com/VinX-To-play/DV505