The 2-sum Report

Dr Jonas Lundberg

Department of Computer Science, Linnaeus University
Jonas.Lundberg@lnu.se

Abstract— **The is an example report for the course 1DV505/1DT910. It is a short and concise report about the 2-sum problem intended as an example for what a report involving experiments might look like. Students will be asked to put together similar reports for their experiments.**

I. Introduction

This report deals with the 2-sum problem and various approaches to handle this problem. It also involves experiments evaluating and comparing different approaches.

A. Problem Formulation

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

This formulation excludes duplicate pairs and that we do not keep pairs (e.g. $(5, 7)$ and $(7, 5)$) having the same two numbers.

All algorithms presented here have been tested numerous times using short lists that allows for manual inspection. For example (using `sum = 1`):

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

B. Experimental Setup

All experiments were performed on the same computer: Macbook Pro from 2023 with an Apple M3 Pro processor equipped with 36 GB of memory and using MacOS Sonoma (version 14.7) . While doing experiments we tried to avoid external disturbances by closing down applications on the computer not needed for the experiments.

Furthermore, each time measurement presented here is an average of five repeated runs and the size of the input data was chosen to give time measurements in the range 0.01 - 10 seconds. For a given size sz the 2-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 such that $n_i + n_j = 0$.

C. Brute Force

The Brute Force solution to the 2-sum problem tests all possible pairs (n_i, n_j) to find pairs such that $n_i + n_j = 0$. Figure 1 shows the Python implementation we used in our experiments.

The inner loop `range(i+1, len(lst))` makes sure that we handle each possible pair only once. Notice also the use of a set (`unique_pairs`) and that we always add pairs

```python
def twosum_brute(lst, sum = 0):
    unique_pairs = set()
    for i in range(len(lst)-1):
        v1 = lst[i]
        for j in range(i+1, len(lst)):
            v2 = lst[j]
            if v1 + v2 == sum:
                if v2 < v1:
                    v1, v2 = v2, v1
                unique_pairs.add((v1, v2))
    return list(unique_pairs)
}
```

Fig. 1
The brute force solution to the 2-sum problem.

`(v1, v2)` such that `v1 < v2`. This guarantees that we only keep unique pairs (n_i, n_j) such that $n_i \leq n_j$.

The two nestled loops (both $O(n)$) gives a time complexity estimate of $O(n^2)$ for the Brute Force algorithm.

D. Brute Force Experiments

By running experiments with various list sizes we found that sizes in range 1000 to 20.000 (step 1000) gives reasonable running times (in range 0.01 to 4 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 i Figure 2.

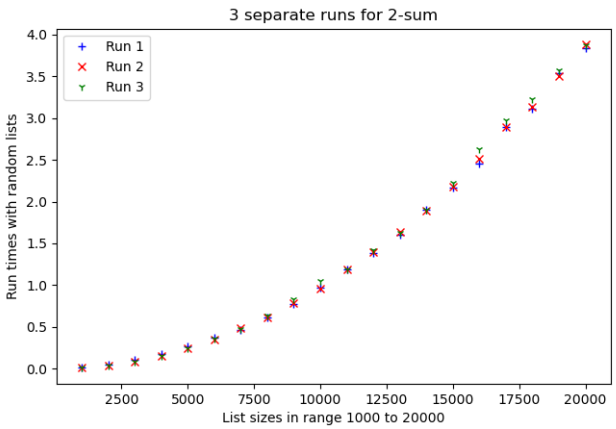

Fig. 2
Three repeated runs with the brute force solution.

The results looks promising. The three runs are very similar, no outliers, and show the upward bend we can expect from a $O(n^2)$ algorithm.

E. Time Complexity

Our next task is to verify the $O(n^2)$ time complexity for the Brute Force algorithm. This is done in two steps:
1. We make 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 $O(n^k)$.
The results of these two steps are shown in Figure 3.

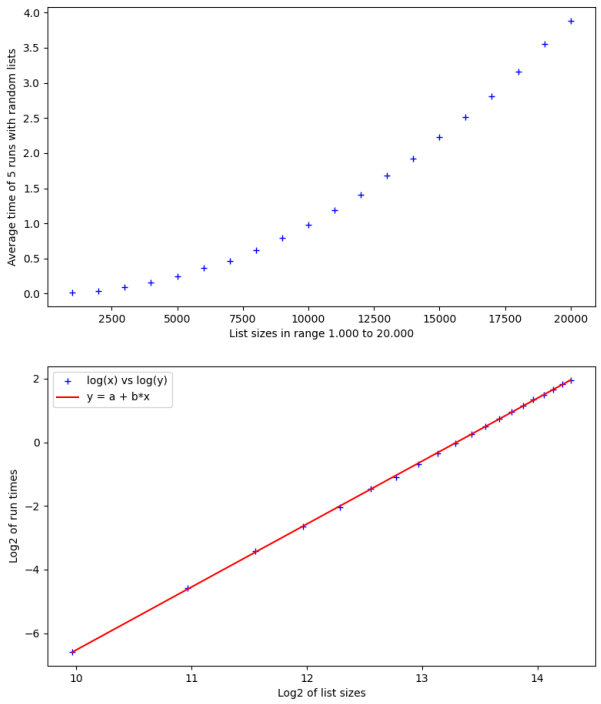

Fig. 3
Average of five runs (top) and fitting a straight line to log(size) vs log(time) (bottom).

The topmost figure shows the average of five runs. As (now) expected, a smooth upward bend with no outliers.

The blue log-log markers in the bottom 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 $k = 1.978$ which is close enough to 2 to allow the conclusion that our implementation is indeed $O(n^2)$ as expected!

II. Faster approaches (VG Exercise)

The Brute Force solution to the 2-sum problem is straight forward and intuitive. In what follows we will take a look at two alternative solutions to the 2-sum problem that comes with a lower time complexity. They are not lengthy in any way but less intuitive.

A. The Two-pointer Approach

The two-pointer algorithm (Figure 4) makes use of the fact that we can solve to 2-sum problem in linear time $O(n)$ if the list is sorted.

```python
def twosum_pointers(lst, sum=0):
    lst = sorted(lst)  # O(n*log(n))
    unique_pairs = set()  # Set ==> unique pairs

    fp, bp = 0, len(lst) - 1
    while fp < bp:
        p = lst[fp] + lst[bp] - sum
        if p == 0:    # Match ==> save pair
            unique_pairs.add((lst[fp], lst[bp]))
            fp += 1
        elif p < 0:    # Move fp to larger value
            fp += 1
        else:          # Move bp to smaller value
            bp -= 1
    return list(unique_pairs)    # Convert set to list)
```

Fig. 4
The two-pointer solution to the 2-sum problem.

Our two-pointer solution once again uses a set `unique_pairs` to make sure that we do not have any duplicate pairs. It uses two pointers `fp` (front pointer) and `bp` (back pointer) initialized to reference the first and last element in the list. Then, if `lst[fp] + lst[bp] < sum` we move `fp` one step to the right to make the sum `lst[fp] + lst[bp]` larger. Similarly if `lst[fp] + lst[bp] > sum` we move `bp` one step to the left to make the sum `st[fp] + lst[bp]` smaller. Using this approach we will find all matching pairs by just visiting each element in the list once. Hence, $O(n)$. As it is, it turns out that the most costly operation is the first line where we sort the list, $O(n \cdot \log(n))$. Hence, the expected time complexity of the two-pointer approach is $O(n \cdot \log(n))$.

B. 2-sum with Caching

The Caching algorithm (Figure 5) iterates over the list elements n_i once and checks if the set of elements we already have visited (`cach`) contains the complement $n_j = sum - n_i$. If that is the case, it means that we have found a pair (n_i, n_j) such that $n_i + n_j = sum$.

```
def twosum_caching(lst, sum=0):
    unique_pairs = set()  # Set ==> unique pairs
    cach = set()  # values already visited

    for v1 in lst:
        v2 = sum - v1  # v2 such that v1+v2 = sum
        if v2 in cach:   # Found!
            if v2 < v1:
                v1, v2 = v2, v1
            unique_pairs.add((v1, v2))
        cach.add(v1)
    return list(unique_pairs)
```

Fig. 5
The caching solution to the 2-sum problem.

We use the same approach with a set `unique_pairs` and an ordering `v1 < v2` to make sure that we only keep unique pairs. More importantly, we iterate over the list only once (i.e. $O(n)$) and in each iteration we look up a value in the set `cach`. Looking up a value in hash based set (Lecture 7) is done in constant time $O(1)$. Hence, the time complexity for the caching algorithm for the 2-sum problem is $O(n)$.

C. Experiments with Faster Algorithms

In this section we present time measurements for the two 2-sum algorithms Two-pointers and Caching. Our aim is to show that both of them are much faster than the Brute Force algorithm, and to find which of the two is the fastest one.

By running experiments with various list sizes we found that sizes in range 1.000.000 (10^6) to 10.000.000 (10^7) with step 500.000 gives reasonable running times (in range 0.2 to 4 seconds). The result of the experiment is shown in Figure 6.

The first thing to notice is that both algorithms have a very similar performance. Second, they are both much faster than Brute Force. (We used list sizes in range $[1.000, 20.000]$ for Brute Force whereas both these algorithms can handle millions of elements.)

That both algorithms have a similar performance is somewhat surprising since Two-Pointers has an estimated time complexity of $O(n \cdot \log(n)$ whereas Caching has time complexity $O(n)$. However, remember that the reason for the $O(n \cdot \log(n)$ complexity for Two-Pointers was due to an initial sorting of the elements. And that sorting (using `sorted()`) in Python is very quick since it is implemented in highly optimized C++ code. Our experiments indicate that the time for sorting can almost be neglected and that the remaining part (iterate over list from two ends) is basically $O(n)$. Hence, both algorithms seems to show a typical straight line $O(n)$ behavior.

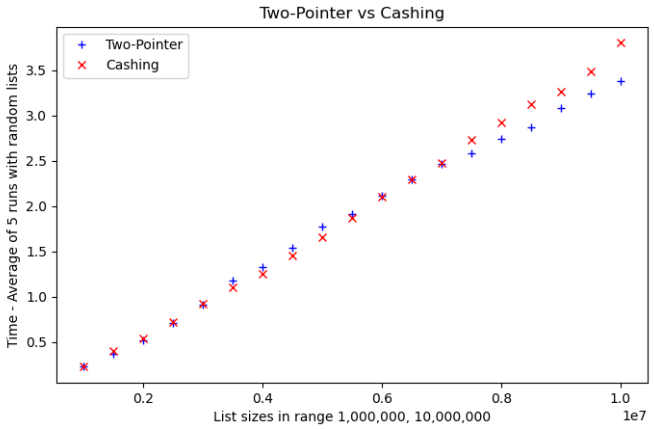

Fig. 6
Two-Pointers vs Cashing.

III. Summary and Conclusions

We have taken a look at three different solutions to the 2-sum problem. The first approach, The Brute Force algorithm, was thoroughly investigated: We presented an implementation, we provided a time complexity estimate $O(n^2)$, and we run experiments that verified the estimated time complexity.

In the second part of this report (the VG part) we presented, implemented, and run experiments on two faster algorithms for the 2-sum problem: 1) Two-pointers, and 2) Caching. Both algorithms are much faster than Brute Force, and should be used for lists larger than (say) 10.000 elements.

The surprise here was that in our experiments Two-pointers (with time complexity $O(n \cdot \log(n)$) was just as fast as Caching (with time complexity $O(n)$). Our explanation: the time consuming part (from a time complexity perspective) for the Two-pointer algorithm is the initial list sorting ($O(n \cdot \log(n))$ can be neglected since it is implemented in very fast C++ code. The remaining part of the algorithm is basically $O(n)$. The is supported by our experiments in which both Two-Pointers and Caching exhibit a typical $O(n)$ behavior.