Week 2

Part 1: The 4-sum problem

Brute-force

This brute-force method is run on four nested loops, making it $O(n^4)$.

$$f(n) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \sum_{k=j+1}^{n} \sum_{l=k+1}^{n} 1$$

We have $\sum_{j=i+1}^{n} 1 = \sum_{j=1}^{n} 1 - \sum_{j=1}^{i} 1 = n - j$

$$\iff \sum_{i=1}^{n} \sum_{j=i+1}^{n} \sum_{k=j+1}^{n} (n-k) \iff \sum_{i=1}^{n} \sum_{j=i+1}^{n} \sum_{k=j+1}^{n} n - \sum_{i=1}^{n} \sum_{j=i+1}^{n} \sum_{k=j+1}^{n} k$$

The larger term here is n, and because n is a constant we only example the sum with n:

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} \sum_{k=j+1}^{n} n \iff n \sum_{i=1}^{n} \sum_{j=i+1}^{n} \sum_{k=j+1}^{n} 1 \iff n \sum_{i=1}^{n} \sum_{j=i+1}^{n} (n-j)$$

Similarly, we have n as the larger term, we only examine the first sum:

$$n \sum_{i=1}^{n} \sum_{j=i+1}^{n} n \iff n^2 \sum_{i=1}^{n} \sum_{j=i+1}^{n} 1 \iff n^2 \sum_{i=1}^{n} (n-i)$$

Similarly, we only examine the summation with n:

$$\iff n^2 \sum_{i=1}^{n} n \iff n^3 \sum_{i=1}^{n} 1 \iff n^4$$

$$\Rightarrow O(n^4)$$

Optimized

```
FUNCTION four_sum_brute(arr):
    // Initialize a map to store pairs with their sum as the
    sum_pairs = EMPTY MAP

    // Populate the map with all pairs
    FOR i FROM 0 TO SIZE(arr) - 1:
        FOR j FROM i + 1 TO SIZE(arr) - 1:
            sum = arr[i] + arr[j]
            IF sum IS NOT IN sum_pairs:
                sum_pairs[sum] = EMPTY LIST
            APPEND (arr[i], arr[j]) TO sum_pairs[sum]

    // Initialize a set to store unique quadruples
    quadruples = UNORDERED_MAP

    // Find quadruples that sum to 0
    FOR EACH (sum, pairs) IN sum_pairs:
        IF -sum IS IN sum_pairs:
            FOR EACH (a, b) IN pairs:
                FOR EACH (c, d) IN sum_pairs[-sum]:
                    IF a, b, c, d ARE ALL DISTINCT:
                        temp = SORTED ARRAY [a, b, c, d]
                        ADD temp TO quadruples

    // Return the count of unique quadruples
    RETURN SIZE(quadruples)
```

The pseudocode presents the optimized sum of four function over the brute-force version. There are two loops in the function. The first one generates all possible unique pairs from the input array. The sum of the pairs will become the key in the map `sum_pairs = {sum_value:[(a,b), (c,d)]}`. This loop has the complexity of $O(n^2)$.

The operation in the inner loop is 1, as dictionary operation and if-clause are considered $O(1)$.

$$\sum_{i=1}^{n}\sum_{j=i+1}^{n}1 \Rightarrow \sum_{i=1}^{n}(n-i) = \sum_{i=1}^{n}n - \sum_{i=1}^{n}i = n^2 - \frac{n(n+1)}{2} = \frac{n(n-1)}{2}$$

As $\sum_{j+1}^{n}1 = \sum_{k=1}^{n}1 - \sum_{k=1}^{j}1 = n-j$

We can see that it is exactly the combination of two C_2^n, which is $O(n^2)$.

Now the problem is the second loop. The second loop is also a nested loop. The outer loop loops through all the unique sum values of all the pairs.

Generally speaking, we know that there should be C_2^n pairs, but it is not necessary C_2^n sum values. In the case where all the sum-value are unique, the dictionary `sum_pairs` has C_2^n key, each key has only one value. For this case, both inner loops are $O(1)$.

For the the case where there is the least number of sum-value key, we should have $2n-3$ keys. that case happens when the array is the arithmetic sequence. We can prove that by using matrix.

	1	2	3	4	5
1	2	3	4	5	6
2	3	4	5	6	7
3	4	5	6	7	8
4	5	6	7	8	9
5	6	7	8	9	10

	1	2	3	4	5
1		3	4	5	6
2			5	6	7
3				7	8
4					9
5					

	1	2	3	4	5
1		3	4	5	6
2					7
3					8
4					9
5					

Because there is no (a,a), and $(a,b) = (b,a)$, we would cut the matrix in half along the diagonal line, inclusively. Temporarily ignore the duplicates, we have $n-1$ for the top row, and $n-2$ for the rightmost column resulting in $2n-3$ unique sum values for the `sum_pairs` dictionary. For each array in each key value, The minimum size is 1 and the maximum is $n//2$.

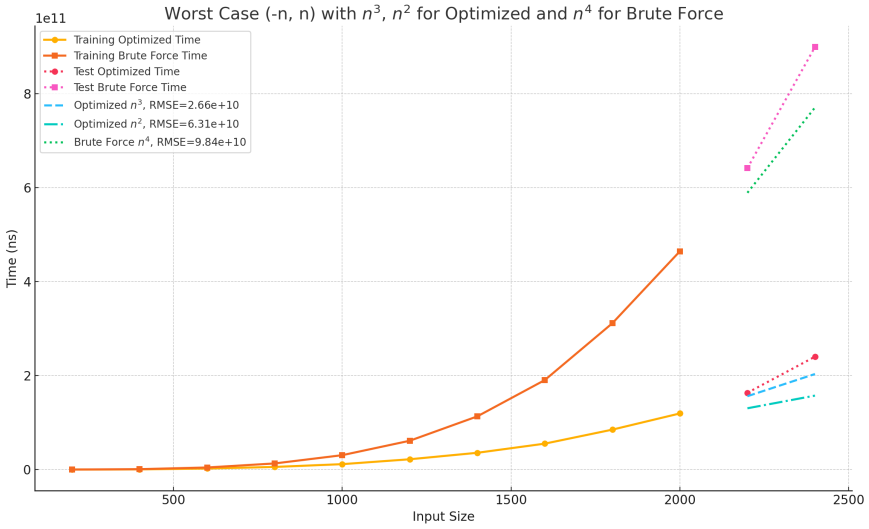
To look for the quadruples, we loop through all the unique sum values. When the traverse happens with minimum to no complementaries of any sum-value key(there is sum-value s, but no $-s$), and if there is a minimum number of

unique sum-value key($2n-3$), the running time would be $O(n)$, but because the first loop is $O(n^2)$, the entire algorithm should be $O(n^2)$.

The worst case should be when half of the sum-value keys are complementary to the other half. There are always C_2^n pairs, and when thing approach the worst case, there are $2n-3$ pairs. Because there is a high number of complementaries, all of the pairs in all of the keys are traversed, making the entire loop and the entire algorithm $O(n^3)$.

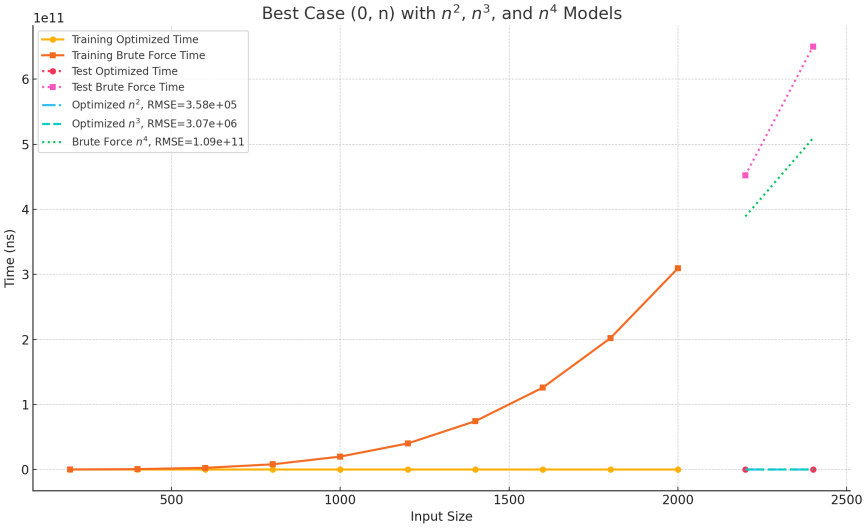
The average case is harder to predict as it depends on the nature of the data. The running time can be $O(n^2)$, or $O(n^3)$.

This algorithm is written based on the use of `unordered_map`, which is $O(1)$. Therefore, the all operations within the three nested loops are effectively $O(1)$.



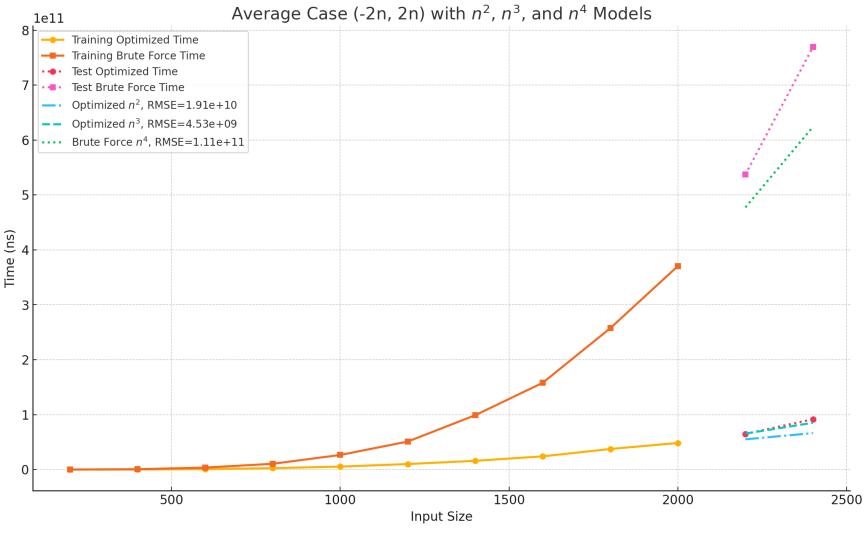
This is the worst case, when the sum-value key range is highly complementary. The test data is generated in the range $[-n, n]$.

The brute-force running-time is shown to be $O(n^4)$, and the optimized one is $O(n^3)$.

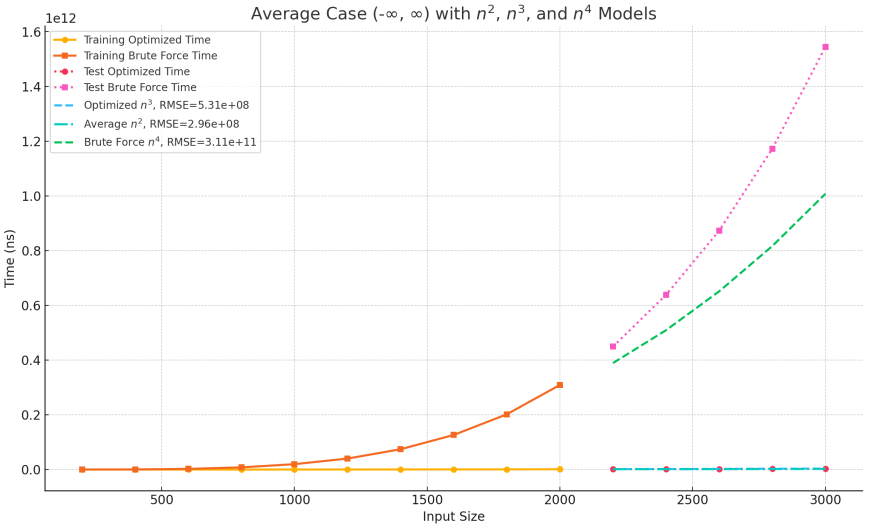


This is the best case when the test data is generated in the range $[0, n)$, resulting in $2n - 3$ unique sum-value key, and they are all positive.

There is only the need to traverse all the unique key, so $O(n)$. The algorithm is $O(n^2)$



This can be seen as a one of the average cases. The test data is generated in the range $[-2n, 2n]$. The sum-value key range still grows linearly. The optimized algorithm still runs at $O(n^3)$, but at a much lower rate than the worse case.



This is another average case. The test data is in range of $(-\infty, \infty)$, making the sum-value key range almost unique, meaning the number of keys approach $O(n^2)$. The chart shows that the running-time of the optimized algorithm is effectively $O(n^2)$.