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. It is the worst case scenario where each pair produces a unique sum value making the dictionary `sum_pairs` having C_2^n key, each key has only one value. For this case, both inner loops are $O(1)$. At this step of analysis, I take all of the operations in the three loops to be $O(1)$.

For the best case scenario, we need to find the array whose pairs of elements produce the smallest number of sum values. To do that, the array needs to be a sequence with values as close to each other as possible because the sum of close values would more likely be that of the next-step pairs. The closest one 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					

First, the diagonal line is omitted because there is no duplicate in one pair(no (a, a)), and we need only to look at one half of the matrix. We can observe that the unique values are on the edge of the matrix. We have $n-1$ for the top row, and $n-2$ for the rightmost column. Together, we have $2n-3$ unique sum values for the `sum_pairs` dictionary. Now for each array in each key value, we can observe that the minimum number of values is 1 and the maximum number of value is $n//2$.

So we have worst case:

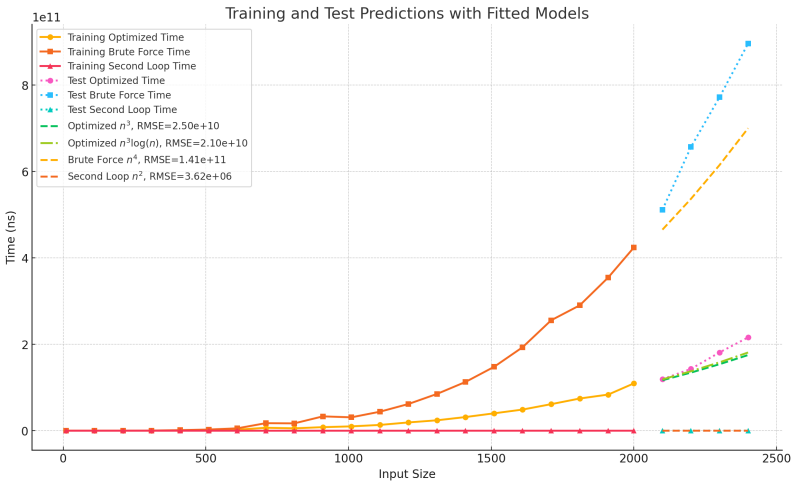
$$C_2^n = \frac{n(n-1)}{2}$$

In an run of the function, we loop through all the unique sum values. For the best case, we do $2n-3$ times. In the case represented in the matrix, there is no $-sum$ present, so there is no running of the inner loops, making it $O(n)$.

To calculate the average complexity, we can deduce by examining the average number of pairs per number sum values on both cases. We always have C_2^n pairs. For the former case, we have C_2^n unique sum values, making the average to be 1. For the latter case, there are $2n-3$ unique sum values, making the average $\frac{C_2^n}{2n-3} \sim \frac{C_2^n}{2n-2} = \frac{n}{4}$. This can lead to confusion as there are two inner loops that loops through the pairs making it seemingly $O(n^2)$; However, this average pair per sum value, when dealing with the `sum_pairs` map, only approaches $\sim \frac{n}{4}$ when all the pairs of all the sum values are traversed, which is rather rare in reality, meaning that the two inner loops often have the complexity of $O(1)$. In random situations, the behaviors of the `sum_pairs` is often closer to the former case than the latter. Therefore, the complexity on average is:

$$O(n^2)$$

However, test run on the source code give a slightly different result. It turns out that the complexity of the `unordered_map` is not $O(1)$. Below is the the chart depicting the running time between the brute-force, the optimized functions, and the optimized function without the inner operation(only the three nested loops analyzed to be $O(n^2)$).



The orange line represents the data obtained through the brute-force approach. The yellow line illustrates the optimized execution time, while the pink line corresponds to the execution time without the inclusion of the inner operation that supports the analytical process.

Under certain circumstances, the behavior of `unordered_map` can get up to $O(n)$. After multiple test, the running time of the second loop(quadruple search) goes much closer to $O(n^3log(n))$. This might be due to the combination of sorting, hashing, assigning and accessing the data with in the loop. These operations together might contribute to the $O(nlog(n))$, and within the $O(n^2)$, making the whole second loop being

$$O(n^3log(n))$$