<https://leetcode.com/problems/pairs-of-songs-with-total-durations-divisible-by-60/description/>

You are given a list of songs where the ith song has a duration of time[i] seconds.

Return the number of pairs of songs for which their total duration in seconds is divisible by 60. Formally, we want the number of indices i, j such that i < j with (time[i] + time[j]) % 60 == 0.

**Example 1:**

**Input:** time = [30,20,150,100,40]

**Output:** 3

**Explanation:** Three pairs have a total duration divisible by 60:

(time[0] = 30, time[2] = 150): total duration 180

(time[1] = 20, time[3] = 100): total duration 120

(time[1] = 20, time[4] = 40): total duration 60

**Example 2:**

**Input:** time = [60,60,60]

**Output:** 3

**Explanation:** All three pairs have a total duration of 120, which is divisible by 60.

**Constraints:**

1 <= time.length <= 6 \* 10^4

1 <= time[i] <= 500

**Attempt 1: 2024-11-30**

**Solution 1: Hash Table (10 min)**

**Must use 'long' type for 'result' in case of test case 35 exceed integer range**

class Solution {

    public int numPairsDivisibleBy60(int[] time) {

        int[] freq = new int[60];

        for(int t : time) {

            freq[t % 60]++;

        }

        long result = 0;

        for(int i = 1; i < 30; i++) {

            result += (long) freq[i] \* freq[60 - i];

        }

        result += (long) freq[0] \* (freq[0] - 1) / 2;

        result += (long) freq[30] \* (freq[30] - 1) / 2;

        return (int) result;

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to**

<https://algo.monster/liteproblems/1010>

**Problem Description**

In this problem, you are given a list of songs with their play durations in seconds. Your task is to find out how many unique pairs of songs have total durations that are divisible by 60. Specifically, if you have two distinct indices i and j such that i < j, you need to count the number of pairs (time[i], time[j]) where the sum of their durations is a multiple of 60 seconds. Mathematically, this means that (time[i] + time[j]) % 60 == 0.

**Intuition**

The intuition behind the solution draws from the modular arithmetic properties. Any time two numbers add up to a multiple of 60, their remainder when divided by 60 must also add up to either 60 or 0. In other words, if one song has a remainder of r when divided by 60, then we need to find another song that has a remainder of 60 - r.

The approach is made efficient by using the following methodology:

We first create a counter that holds how many times each remainder occurs when we divide each song's length by 60.

We iterate over the possible remainders from 1 to 29. Why? Because each unique pair's remainders would add up to 60, and we don't need to check beyond 30 since 30's pair would be 30 itself and that would result in double counting.

For each remainder x, we find its complementary remainder 60-x. The number of pairs that can be formed with these remainders is the product of their occurrences: cnt[x] \* cnt[60 - x].

We separately calculate pairs for the songs which themselves are a multiple of 60 (cnt[0]). These form pairs with themselves, and thus the number of such combinations is cnt[0] \* (cnt[0] - 1) / 2.

We perform a similar calculation for the songs which have a remainder of 30 seconds, since they also pair with themselves, and the number of combinations is cnt[30] \* (cnt[30] - 1) / 2.

We sum up these results to get the total number of valid pairs.

In summary, we use a counting method and leverage the property of numbers and their remainders when dividing by 60 to efficiently calculate the total number of pairs.

**Solution Approach**

The solution is implemented in Python and uses several key concepts to optimize the counting of song pairs:

Hash Table (Counter): The Counter class from Python's collections module is used to keep track of how many times each remainder occurs when the song lengths are divided by 60. Hash tables allow for efficient lookup, insertion, and update of the count, which is crucial for this problem.

Example: If time = [30, 20, 150, 100, 40], then cnt after modulo and counting will be {30: 1, 20: 1, 30: 1, 40: 1, 40: 1} which simplifies the process of finding complements.

Modular Arithmetic:

The core principle used here is that if two numbers a and b satisfy (a + b) % 60 == 0, then a % 60 + b % 60 must equal either 0 or 60.

For each remainder from 1 to 29, we find its complement 60 - x and calculate possible pairs between them.

Special Cases for Remainders 0 and 30:

For songs that are exactly divisible by 60 (remainder == 0), they can only pair up with other songs that are also exactly divisible by 60. The number of such pairs is a combination count calculated by cnt[0] \* (cnt[0] - 1) / 2, using the formula n \* (n - 1) / 2 for pairs.

The same logic applies to songs with a remainder of 30 since they also pair with themselves. This is calculated by cnt[30] \* (cnt[30] - 1) / 2.

Summation of Pairs:

The total number of pairs is calculated by summing the pairs formed by remainders x and 60 - x for x from 1 to 29, and the special case pairs where the remainder is exactly 0 or 30.

This is achieved by iterating through range 1 to 29, and adding the special cases separately, followed by returning the sum as the final answer.

The implementation cleverly avoids double-counting by ensuring that pairs are only counted once by considering only x from 1 to 29 and calculating the complement pairs directly. It also deals with the edge cases where the song lengths are either a multiple of 60 or half of 60. The use of the Counter data structure, modulo operation, and understanding of how remainders can be paired provides an efficient and elegant solution to the problem.

**Example Walkthrough**

Let's assume we have the following list of song durations in seconds: time = [60, 20, 120, 100, 40]. We want to find how many unique pairs of songs have total durations that are divisible by 60.

Following the solution approach given in the problem content:

Use a Counter to tally remainders: First, we take the modulo 60 of each duration to find the remainder, and we create a counter to hold how many times each remainder occurs. Doing this for our example, we get:

The remainders for [60, 20, 120, 100, 40] after modulo 60 are [0, 20, 0, 40, 40].

Thus, the counter cnt becomes {0: 2, 20: 1, 40: 2}.

Calculate pairs for remainders from 1 to 29: We iterate over the possible remainders and find the count of their complements. For our example, the relevant remainders and their complements are:

For remainder 20: The complement is 60 - 20 = 40.

The count for remainder 20 is 1, and the count for its complement 40 is 2.

Thus, the number of pairs with remainders that would add up to 60 is 1 \* 2 = 2.

Special cases for remainders 0 and 30: We check for the special cases:

For remainder 0: There are 2 songs with a duration that is perfectly divisible by 60.

The number of pairs among them is cnt[0] \* (cnt[0] - 1) / 2, which in numbers is 2 \* (2 - 1) / 2 = 1.

In this example, there is no song with remainder 30 so we do not have that case.

Summation of pairs: Finally, we sum all the counts for valid pairs to get the total number of pairs which are divisible by 60.

From steps 2 and 3, we have 2 + 1 = 3 pairs.

In conclusion, the example time list has 3 unique pairs of songs that have total durations which are divisible by 60, which are the pairs (60, 100), (20, 40), and (20, 40) (note that the last two pairs are different because they represent different indices in the list, not the same pair counted twice).

**Solution Implementation**

class Solution {

public int numPairsDivisibleBy60(int[] times) {

int[] count = new int[60]; // Create an array to store counts for each remainder when divided by 60

// Count the occurrences of each remainder when the song lengths are divided by 60

for (int time : times) {

count[time % 60]++;

}

int numberOfPairs = 0; // Initialize the number of pairs that are divisible by 60

// For each pair of remainders (x, 60-x), calculate the number of valid combinations

for (int i = 1; i < 30; i++) {

numberOfPairs += count[i] \* count[60 - i];

}

// Add the special cases where the remainders are exactly 0 or 30 (since 30 + 30 = 60)

// Calculate combinations using the formula n \* (n - 1) / 2 for each special case

numberOfPairs += count[0] \* (count[0] - 1) / 2; // Pairs where both times have no remainder

numberOfPairs += count[30] \* (count[30] - 1) / 2; // Pairs where both times leave a remainder of 30

// Return the total number of pairs that have song lengths summing to a multiple of 60

return numberOfPairs;

}

}

**Time and Space Complexity**

**Time Complexity**

The time complexity of the given code can be broken down into a few parts:

The creation of the counter cnt that stores the frequencies of each remainder when the time elements are divided by 60. This requires one pass over the time list, which makes this part O(N) where N is the length of time.

The summation sum(cnt[x] \* cnt[60 - x] for x in range(1, 30)) iterates from 1 to 29, which is a constant number of iterations. Therefore, this part takes O(1) time, as the range does not depend on the size of the input.

The last two lines calculate the pairs for the special cases where the elements are directly divisible by 60 (remainder is 0) and the pairs where the remainders are exactly 30. These calculations are also done in constant time O(1).

Combining all these, since O(N + 1 + 1) simplifies to O(N), the overall time complexity of the code is O(N).

**Space Complexity**

The space complexity of the code is dominated by the space required for the Counter object cnt. In the worst case, if all time elements give a different remainder when divided by 60, the counter will contain up to 60 keys (since remainders range from 0 to 59). Therefore, the space complexity is O(1) because the space required does not grow with N, it is limited by the constant number 60.