1128. Number of Equivalent Domino Pairs

Easy Array Hash Table Counting

Leetcode Link

Problem Description

In this problem, we are given a list of tiles representing dominoes, each domino represented as a pair of integers `[a, b]`. Two dominoes `[a, b]` and `[c, d]` are defined to be equivalent if one can be rotated to become the other, meaning either `(a == c and b == d)` or `(a == d and b == c)`. The goal is to return the number of pairs `(i, j)` where `i` is less than `j` and `dominoes[i]` is equivalent to `dominoes[j]`.

Intuition

The intuition behind the solution is to efficiently count the pairs of equivalent dominoes. To do this, we can represent each domino in a standardized form so that equivalent dominoes will have the same representation, regardless of their orientation. We store the count of each unique domino in a hash map (`Counter` in Python).

Instead of using a tuple to represent the standardized domino (since `[1, 2]` is equivalent to `[2, 1]`), we create a unique integer representation `x` for each domino by multiplying the larger of the two numbers by `10` and adding the smaller one. This ensures that both `[1, 2]` and `[2, 1]` will have the same representation `12`.

We then iterate over the list of dominoes and for each domino, we calculate its standardized representation `x`. We then increment the answer `ans` by the current count of `x` already seen (since for each new domino found, it can pair up with all previous identical ones). We update the counter by adding one to the count of `x`.

This approach allows us to efficiently calculate the number of pairs without the need to explicitly compare each pair of dominoes.

Solution Approach

The solution uses a hash map, implemented as a `Counter` object in Python, to count occurrences of the standardized representations of dominoes. The hash map allows for quick look-ups and insertions, which is key to the efficiency of this algorithm.

Walking through the implementation:

1. Initialize a `Counter` object (`cnt`) which will keep track of the counts of standardized representations of the dominoes.
2. Set `ans` to `0`. This will hold the final count of equivalent pairs.
3. Iterate over each domino in the `dominoes` list, which is a list of lists where each sublist represents a domino `[a, b]`.
4. For each domino, we create a standardized representation `x`. If `a < b`, we construct `x` by `a * 10 + b`, otherwise, if `b <= a`, by `b * 10 + a`. This step ensures equivalent dominoes have the same representation regardless of order.
5. We add to `ans` the current count of `x` from our `Counter`. This is due to the fact that if we have already seen a domino of this representation, the current one can form a pair with each of those. For instance, if `x` has a count of 2, and we find another `x`, we can pair this third one with each of the two previous ones, adding 2 to our `ans`.
6. We then increment the count of `x` in our `Counter` because we've encountered another domino of this type.

The time complexity of the algorithm is O(n), where n is the number of dominoes, since we go through each domino exactly once. The space complexity is O(n) as well, since in the worst case we might have to store a count for each unique domino.

Here is the algorithm translated into Python code:

```python
class Solution:
    def numEquivDominoPairs(self, dominoes: List[List[int]]) -> int:
        cnt = Counter()
        ans = 0
        for a, b in dominoes:
            x = a * 10 + b if a < b else b * 10 + a
            ans += cnt[x]
            cnt[x] += 1
        return ans
```

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Imagine we have the following list of dominoes:

```python
dominoes = [[1, 2], [2, 1], [3, 4], [5, 6], [6, 5], [3, 4]]
```

We want to find pairs of equivalent dominoes. Our approach involves creating a standardized representation for each domino.

1. Initialize the counter object `cnt` to keep track of domino representations.
2. Set `ans` to 0.
3. The first domino is `[1, 2]`. Because `1 < 2`, its representation `x` is `1*10 + 2 = 12`.
4. Since this is the first time we see `x = 12`, `cnt[x]` is 0 and so we do not add to `ans`.
5. Increment `cnt[12]` to 1.

Now, we move on to the second domino:

1. The second domino is `[2, 1]` which is equivalent to `[1, 2]`. Its representation is thus `12`.
2. Now, as `cnt[12]` is 1, we add 1 to `ans` because this new domino can pair with one previous equivalent domino.
3. Increment `cnt[12]` to 2.

Next, we have `[3, 4]`:

1. For domino `[3, 4]`, `x` is `3*10 + 4 = 34`.
2. Since this is the first domino of its kind, no addition to `ans`.
3. Update `cnt[34]` to 1.

For `[5, 6]`, we set `x` to `56` and follow the same steps.

Next, consider the domino `[6, 5]`, which is equivalent to `[5, 6]`:

1. Its standardized form is `56`.
2. We find `cnt[56]` equals 1, so we add 1 to `ans`.
3. Increment `cnt[56]` to 2.

Finally, for the second `[3, 4]` domino:

1. The standardized form is `34`.
2. `cnt[34]` equals 1, so we increment `ans` by 1.
3. `cnt[34]` becomes 2.

After processing all the dominoes, our `ans` value is the sum of additions made which, in this example, is `0 + 1 + 0 + 0 + 1 + 1 = 3`. Therefore, there are 3 equivalent pairs of dominoes in the list.

Representing this in Python code according to the given solution approach we have:

```python
from collections import Counter

class Solution:
    def numEquivDominoPairs(self, dominoes: List[List[int]]) -> int:
        cnt = Counter()
        ans = 0
        for a, b in dominoes:
            x = a * 10 + b if a < b else b * 10 + a
            ans += cnt[x]
            cnt[x] += 1
        return ans

# Initial dominoes list
dominoes = [[1, 2], [2, 1], [3, 4], [5, 6], [6, 5], [3, 4]]

# Instantiate solution and calculate the equivalent pairs
solution = Solution()
print(solution.numEquivDominoPairs(dominoes))  # Output: 3
```

The output, as expected, is `3`, meaning we have found three pairs of equivalent dominoes in our list.

Python Solution

```python
from collections import Counter

class Solution:
    def numEquivDominoPairs(self, dominoes: List[List[int]]) -> int:
        # Initialize a counter to keep track of the occurrences of each normalized domino
        domino_counter = Counter()
        # Initialize a variable to store the number of equivalent domino pairs
        equivalent_pairs_count = 0

        # Iterate over the list of dominoes
        for domino in dominoes:
            # Normalize the domino representation to be the same regardless of order
            # by ensuring the smaller number is in the tens place.
            normalized_domino = min(domino) * 10 + max(domino)

            # The current count of the normalized domino in the counter is the number of pairs
            # that can be formed with the current domino, since all previous occurrences
            # can form a pair with it.
            equivalent_pairs_count += domino_counter[normalized_domino]

            # Increment the count of the normalized domino in the counter
            domino_counter[normalized_domino] += 1

        # Return the total count of equivalent domino pairs
        return equivalent_pairs_count
```

Remember that `List` needs to be imported from `typing` if you're using a version of Python earlier than 3.9 in which `list` itself is not yet directly usable as a generic type. If you're using Python 3.9 or later, you can omit the import and use `list` with lowercase 'l'.

Here is the import statement for earlier versions of Python:

```python
from typing import List

```

Java Solution

```java
class Solution {

    public int numEquivDominoPairs(int[][] dominoes) {
        // This array holds the count of normalized representations of dominoes.
        int[] count = new int[100];
        int numberOfPairs = 0; // This will store the total number of equivalent domino pairs.

        // Loop through each domino in the array of dominoes.
        for (int[] domino : dominoes) {
            int lesserValue = Math.min(domino[0], domino[1]); // Find the lesser value of the two sides of the domino.
            int greaterValue = Math.max(domino[0], domino[1]); // Find the greater value of the two sides of the domino.

            // Normalize the representation of the domino so that the
            // lesser value comes first (e.g., [2,1] becomes [1,2]).
            int normalizedDomino = lesserValue * 10 + greaterValue;

            // If this normalized domino has been seen before, increment the number of pairs
            // by the count of how many times the same domino has been encountered. Then,
            // increment the count for this domino type.
            numberOfPairs += count[normalizedDomino]++;
        }

        return numberOfPairs; // Return the total count of equivalent domino pairs.
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to count the number of equivalent domino pairs.
    int numEquivDominoPairs(vector<vector<int>>& dominoes) {
        // Array to count occurrences of normalized domino pairs.
        int count[100] = {0};

        // Variable to store the number of equivalent domino pairs.
        int numOfPairs = 0;

        // Iterate through each domino in the given vector of dominoes.
        for (auto& domino : dominoes) {
            // Normalize the domino representation so that
            // the smaller number comes first (e.g., [2,1] is treated as [1,2]).
            int normalizedValue = domino[0] < domino[1]
                                ? domino[0] * 10 + domino[1]
                                : domino[1] * 10 + domino[0];

            // Increment the count for this domino representation.
            // Since we're finding the number of equivalent pairs, we add
            // the current count (before incrementing) to 'numOfPairs'
            numOfPairs += count[normalizedValue]++;
        }

        // Return the total number of equivalent domino pairs found.
        return numOfPairs;
    }
};

```

Typescript Solution

```typescript
// Type definition for a dominos pair.
type Domino = [number, number];

// Global count array to keep track of normalized domino pairs.
const count: number[] = new Array(100).fill(0);

// Function to count the number of equivalent domino pairs.
function numEquivDominoPairs(dominoes: Domino[]): number {

    // Variable to store the number of equivalent domino pairs.
    let numOfPairs: number = 0;

    // Iterate through each domino in the given array of dominoes.
    for (let domino of dominoes) {

        // Normalize the domino representation so that
        // the smaller number is the first element (e.g., [2,1] becomes [1,2]).
        let normalizedValue: number = domino[0] < domino[1]
            ? domino[0] * 10 + domino[1]
            : domino[1] * 10 + domino[0];

        // Increment the count for this normalized domino representation.
        // Since we're finding the number of equivalent pairs, we add
        // the current count (before incrementing) to 'numOfPairs'.
        numOfPairs += count[normalizedValue];

        // Now increment the count for future pairs.
        count[normalizedValue]++;
    }

    // Return the total number of equivalent domino pairs found.
    return numOfPairs;
}

// Example usage:
// const dominoes: Domino[] = [[1, 2], [2, 1], [3, 4], [5, 6]];
// const result: number = numEquivDominoPairs(dominoes);
// console.log(result); // Output will be the number of equivalent pairs.

```

Time and Space Complexity

Time Complexity

The given code iterates over each domino pair exactly once, which means the primary operation scales linearly with the number of dominoes. Inside the loop, the code performs constant-time operations: a conditional, basic arithmetic operations, and a lookup/update in a `Counter` data structure (which is a subclass of a dictionary in Python).

Dictionary lookups and updates typically operate in `O(1)` on average due to hashing. However, in the worst case, if a lot of collisions happen, these operations can degrade to `O(n)`. Since this is unlikely with the hash functions used in modern Python implementations for primitive data types like integers, we will consider the average case for our analysis.

Hence, the time complexity is `O(n)` where `n` is the number of domino pairs in the input list, `dominoes`.

Space Complexity

The space complexity is determined by the additional space used by the algorithm, which is primarily occupied by the `Counter` object `cnt`. In the worst case, if all domino pairs are unique after standardization (by sorting each tuple), the counter object will grow linearly with the input. This means we will have a space complexity of `O(n)`.

To summarize, the space complexity is `O(n)` where `n` is the number of domino pairs in `dominoes`.