575. Distribute Candies

Easy Array Hash Table

Problem Description

Alice has a number of candies, with each candy being a certain type. To reduce her sugar intake, her doctor recommends she only eats half of the total amount of candies she has. Alice loves variety and wants to maximize the different types of candies she can enjoy while adhering to this health advice. Our task is to determine the maximum number of unique candy types Alice can eat if she eats only half of her total candies. The types of candies are represented in an array where each element corresponds to a specific candy type.

Intuition

To maximize the varieties of candies Alice can consume, she should aim to pick as many different types as possible. However, there's a limit on the maximum number of candies she can eat, which is precisely half of her total candy count. Here's the intuition behind the solution:

- First, we count the number of different types of candy. This is done by converting the `candyType` array to a set and measuring its length, which removes duplicates and gives us the number of unique candy types.
- Second, since Alice can eat at most `n / 2` candies, we compare this number to the number of unique candy types available.
- The actual number of types Alice can eat is the minimum of these two numbers — the unique candy types and `n / 2`. This ensures Alice eats the maximum number of different types without exceeding the doctor's advice limit.
- By using a bitwise right shift operator (`>> 1`) instead of a standard division by 2, the solution efficiently computes the half of the total candy count.

Solution Approach

The solution approach for finding the maximum number of different types of candies that Alice can eat is straightforward. Let's walk through the implementation details:

- First, we start with the `distributeCandies` function, which receives the `candyType` list as input.
- We make use of Python's set data structure to find the number of unique candy types. By converting `candyType` into a set (`set(candyType)`), duplicates are removed, and we're left with only unique elements. We use the `len()` function to find the total count of these unique candy types.
- Next, to calculate the maximum number of candies Alice is allowed to eat, we take the length of the `candyType` list (which represents the total number of candies Alice has) and perform a right bitwise shift using `>> 1`. This operation is equivalent to dividing by 2 but is often faster in practice than using the division operator. This calculation represents the doctor's advice, which limits Alice to eating only half of her candies.
- The final step involves finding the minimum value between the number of unique candy types and the maximum number of candies Alice can eat (which is `n / 2`). We can express this step in code as `min(len(candyType) >> 1, len(set(candyType)))`.
- The reason we take the minimum is because if there are more unique candy types than the number of candies Alice can eat, she is still limited by the amount she can consume (`n / 2`). Conversely, if there are fewer unique types than `n / 2`, she can only eat as many different types as are available.

Overall, the algorithm relies on the efficiency of set operations and bitwise calculations to determine the solution. The time complexity of converting the list to a set is O(n), where n is the number of elements in the `candyType` list. Finding the length of a list or set is an O(1) operation. The bitwise shift and comparison (in the `min` function) are also O(1) operations. Thus, the overall time complexity is dominated by the set conversion, making it O(n).

Example Walkthrough

Let's assume Alice has the following array of candy types:

```
candyType = [1, 1, 2, 2, 3, 3, 4, 4]
```

In this array:

- Candy type 1 appears twice.
- Candy type 2 appears twice.
- Candy type 3 appears twice.
- Candy type 4 appears twice.

There are a total of 8 candies, and since Alice can only eat half of them based on her doctor's advice, she can eat 4 candies.

Now let's follow the solution approach step by step:

1. First, we convert the `candyType` array to a set to find the number of unique candy types:

```
unique_candies = set(candyType)
# unique_candies will be {1, 2, 3, 4}
```

After converting to a set, we find that there are 4 unique candy types.

2. Next, we determine the maximum number of candies that Alice can eat, which is half the total count:

```
max_candies_Alice_can_eat = len(candyType) >> 1
# This is equivalent to len(candyType) / 2 which is equal to 4
```

3. We then find the minimum between the number of unique candy types and the total number of candies Alice can consume:

```
max_unique_types_Alice_can_eat = min(max_candies_Alice_can_eat, len(unique_candies))
# This will be min(4, 4) which equals 4
```

4. Therefore, the maximum number of unique candy types Alice can eat is 4. In this case, because the number of unique types is the same as the maximum she can eat, she can enjoy one of each type without exceeding the limit placed by her doctor.

By following this approach, we can efficiently conclude that Alice can eat all four unique types of candies, getting the maximum variety without exceeding the limit of candies she can consume.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def distributeCandies(self, candy_types: List[int]) -> int:
        # Calculate the maximum number of different candy types a sister can receive.
        # This is the minimum between half the total number of candies and the number of unique candy types.

        # Find the total number of candies.
        total_candies = len(candy_types)

        # Find the number of unique candy types.
        unique_candy_types = len(set(candy_types))

        # The sister can receive at most half the total number of candies, if there are enough unique types.
        # Otherwise, she gets as many unique types as are available.
        max_unique_candies = min(total_candies // 2, unique_candy_types)

        return max_unique_candies
```

Java

```java
import java.util.Set;
import java.util.HashSet;

class Solution {
    // Method to determine the maximum number of different types of candies
    // one can eat if only allowed to eat n / 2 of them
    public int distributeCandies(int[] candyTypes) {
        // Creating a HashSet to store unique candy types
        Set<Integer> uniqueCandyTypes = new HashSet<>();

        // Iterating over the array of candy types
        for (int candyType : candyTypes) {
            // Adding the candy type to the set to ensure uniqueness
            uniqueCandyTypes.add(candyType);
        }

        // Calculating the half number of total candies
        int halfCandies = candyTypes.length / 2;

        // The number of types one can eat is the minimum of
        // half the total number of candies and the number of unique candy types
        return Math.min(halfCandies, uniqueCandyTypes.size());
    }
}
```

C++

```cpp
#include <vector>       // Necessary for vector usage
#include <unordered_set> // Necessary for unordered_set usage

class Solution {
public:
    int distributeCandies(vector<int>& candyType) {
        unordered_set<int> uniqueCandies; // Using a set to store unique candy types

        // Inserting each candy type into the set to ensure uniqueness
        for (int type : candyType) {
            uniqueCandies.insert(type);
        }

        // The sister can have at most half of the total candies
        int maxCandiesForSister = candyType.size() / 2;

        // The number of types sister can have is either limited by
        // half of the total number of candies or the number of unique candies,
        // whichever is smaller.
        return min(maxCandiesForSister, uniqueCandies.size());
    }
};
```

TypeScript

```typescript
// Necessary imports in TypeScript:
// In TypeScript, we usually don't need to import constructs
// for primitive operations such as working with arrays or sets.

// Function to distribute candies
function distributeCandies(candyType: number[]): number {
    // Using a Set to store unique candy types
    const uniqueCandies: Set<number> = new Set();

    // Inserting each candy type into the set to ensure uniqueness
    for (const type of candyType) {
        uniqueCandies.add(type);
    }

    // The sister can have at most half of the total candies
    const maxCandiesForSister: number = candyType.length / 2;

    // The number of types the sister can have is either limited by
    // half of the total number of candies or the number of unique candies,
    // whichever is smaller.
    return Math.min(maxCandiesForSister, uniqueCandies.size);
}

// Export the function if this module is to be used in other parts of the application
export { distributeCandies };
```

```python
from typing import List

class Solution:
    def distributeCandies(self, candy_types: List[int]) -> int:
        # Calculate the maximum number of different candy types a sister can receive.
        # This is the minimum between half the total number of candies and the number of unique candy types.

        # Find the total number of candies.
        total_candies = len(candy_types)

        # Find the number of unique candy types.
        unique_candy_types = len(set(candy_types))

        # The sister can receive at most half the total number of candies, if there are enough unique types.
        # Otherwise, she gets as many unique types as are available.
        max_unique_candies = min(total_candies // 2, unique_candy_types)

        return max_unique_candies
```

Time and Space Complexity

Time Complexity

The time complexity of the code is determined by two operations: the conversion of the `candyType` list into a set with `set(candyType)`, and the calculation of the minimum with `min(len(candyType) >> 1, len(set(candyType)))`.

1. `len(candyType)`: The `len` function has constant time complexity, which is `O(1)`.
2. `set(candyType)`: Converting a list to a set has a time complexity of `O(n)` because it involves iterating over all elements of the list to create a set of unique elements.
3. `len(set(candyType))`: Once the set is created, calculating its length is `O(1)`.
4. `min(a, b)`: Calculating the minimum of two numbers has constant time complexity, `O(1)`.

Putting it all together, the dominant term is the conversion from list to set, which gives us a final time complexity of `O(n)` where `n` is the number of elements in `candyType`.

Space Complexity

The space complexity of the code involves the additional space required for the set of unique candies.

1. `set(candyType)`: This set can contain at most `n` elements, if all candy types are unique. Thus, the space complexity for storing the set is `O(n)`.

Therefore, the total space complexity of the function is also `O(n)`.