888. Fair Candy Swap Binary Search Sorting Hash Table Easy

Problem Description

candies each of them has is different. Given two lists, aliceSizes representing the number of candies in Alice's boxes and bobSizes for Bob's boxes, the task is to find a pair of boxes (one from Alice and one from Bob) that they can exchange with each other so that they end up with the same total number of candies. In other words, we need to find two values, one from each list, such that when Alice gives Bob the box with her value and Bob

Alice and Bob each have a collection of candy boxes, with each box containing a certain number of candies. The total number of

gives Alice the box with his value, the total candies they each have will be equal. The problem ensures that there is at least one such pair and asks us to return any one of them as a 2-element array: [Alice's box of candies, Bob's box of candies].

The solution to this problem relies on the concept of the exchange itself and the resulting equality of candy totals for Alice and

Intuition

Bob. If x is the amount of candies in the box Alice gives to Bob, and y is the amount in the box Bob gives to Alice, after the exchange, the total candies for both should be the same. If we denote sumA as the total candies that Alice has and sumB for Bob, the equation after the exchange would be: (sumA - x) + y = (sumB - y) + x

```
sumA - sumB = 2 * (x - y)
```

Which can be simplified to:

By rearranging the terms, we get the difference diff between the totals divided by 2 equals the difference between x and y: diff = sumA - sumB

```
x - y = diff / 2
```

box counts, we can quickly check whether this target exists. Because Set operations in Python are O(1) on average, this check is efficient. If we find a match (target in Bob's Set), we return that pair [a, target] as our solution.

Given this, we can then iterate through Alice's boxes, and for each candy count a, we can calculate the corresponding count

target = a - diff / 2 that we're looking for in Bob's boxes. Then by utilizing a Set in Python that contains all of Bob's candy

The fairCandySwap algorithm takes two lists of integers as input: aliceSizes and bobSizes. Each list represents the sizes of

1. Calculate the difference diff between Alice's and Bob's total candy amounts and divide it by 2.

3. Iterate through Alice's candy sizes, calculate the corresponding target size for Bob.

The approach starts by calculating the difference in total candy amounts between Alice and Bob and dividing it by 2. The division by 2 comes from rearranging the equation sumA - sumB = 2 * (x - y) into x - y = (sumA - sumB) / 2. This value is stored in

Solution Approach

candy boxes for Alice and Bob, respectively.

size target that Bob needs to provide:

2. Create a Set from Bob's candy sizes for efficient lookup.

5. Return the first pair [a, target] where target is in Bob's Set.

4. Check if target exists in Bob's Set.

for a in aliceSizes:

the variable diff, signifying the amount of candy that needs to be compensated for in the swap to balance the totals: diff = (sum(aliceSizes) - sum(bobSizes)) >> 1

In the code, the right-shift operator >> 1 is used as an efficient way to divide the difference by 2. Next, a Set called s is created from Bob's list of candy sizes. Sets allow for constant time complexity (O(1)) checks for the existence of an element, which is leveraged later:

```
s = set(bobSizes)
 The algorithm then iterates through the list of Alice's candy sizes. For each candy size a, it computes the corresponding candy
```

target = a - diff if target in s: return [a, target]

```
This loop checks whether the calculated target exists in Bob's set of candy sizes (s). If it does, that means there is a pair of
candy boxes [a, target] that can be swapped to ensure both Alice and Bob end up with the same total amount of candy. The
algorithm then returns this pair as soon as it's found.
In summary, the algorithm follows these steps:
```

straightforward and performant. **Example Walkthrough**

This approach leverages arithmetic to determine the needed exchange and a Set for efficient lookup, making the solution both

Here's how we would use the solution approach to find the pair of boxes to be exchanged:

• bobSizes = [2, 4]

• aliceSizes = [1, 2, 5]

 \circ sumA = 1 + 2 + 5 = 8 \circ sumB = 2 + 4 = 6

Therefore, the difference diff = (sumA - sumB) / 2 = (8 - 6) / 2 = 1.

 \circ When a = 1, target = 1 - diff = 1 - 1 = 0. Since 0 is not in set s, we continue.

def fairCandySwap(self, alice sizes: List[int], bob sizes: List[int]) -> List[int]:

Compute the difference in candy sizes between Alice and Bob, divided by 2

because any swap should compensate for half of the total size difference.

Return a list containing Alice's candy size and the corresponding

int sumAlice = 0, sumBob = 0; // Initialize sums of Alice's and Bob's candies

Set<Integer> bobCandies = new HashSet<>(); // Create a set to store Bob's candy sizes

Calculate the total candies and the difference diff:

Create a Set of Bob's candy sizes for efficient lookup:

For a in [1, 2, 5], we calculate target as follows:

Return the first successful pair [a, target]:

We sum up the candies in Alice's and Bob's boxes:

Let's assume we have the following candy box sizes for Alice and Bob:

```
s = set(bobSizes), which gives us s = \{2, 4\}.
Iterate through Alice's candy sizes to find the corresponding target for Bob:
```

```
\circ When a = 2, target = 2 - diff = 2 - 1 = 1. Since 1 is not in set s, we continue.
\circ When a = 5, target = 5 - diff = 5 - 1 = 4. 4 is in set s.
```

candies, and they will both end up with 7 candies total, achieving a fair candy swap.

size_difference = (sum(alice_sizes) - sum(bob_sizes)) // 2

Create a set of Bob's candy sizes for constant-time look-up

target_size_bob = candy_size_alice - size_difference

return [candy_size_alice, target_size_bob]

Check if Bob has a candy of the target size

Bob's candy size for a fair swap

// Method to find the fair candy swap between Alice and Bob

#include <vector> // Include necessary library for vector usage

// Method to find a fair swap of candies between Alice and Bob

int sizeDifference = (aliceCandySum - bobCandySum) / 2;

// Create a set for Bob's candy sizes for efficient look-up.

unordered_set<int> bobSizeSet(bobSizes.begin(), bobSizes.end());

// The target size for Bob that would balance the swap.

result = vector<int>{aliceCandv, targetSize};

Create a set of Bob's candy sizes for constant-time look-up

Iterate through Alice's candy sizes to find the fair swap

target_size_bob = candy_size_alice - size_difference

return [candy_size_alice, target_size_bob]

Note: no need for an explicit return statement for no match scenario

Check if Bob has a candy of the target size

Bob's candv size for a fair swap

if target size bob in bob sizes set:

Calculate the target size of Bob's candy that would make the swap fair

Return a list containing Alice's candy size and the corresponding

break; // Found the correct swap, exit the loop.

int targetSize = aliceCandy - sizeDifference;

vector<int> fairCandySwap(vector<int>& aliceSizes, vector<int>& bobSizes) {

int bobCandySum = accumulate(bobSizes.begin(), bobSizes.end(), 0);

int aliceCandySum = accumulate(aliceSizes.begin(), aliceSizes.end(), 0);

// Calculate the difference between Alice's and Bob's candy sums divided by 2.

// Iterate through Alice's candy sizes to find a matching size in Bob's set.

// If the target size is in Bob's set, a fair swap is possible.

#include <numeric> // Include library for accumulate function

// Calculate the total sum of candies Alice has.

// Calculate the total sum of candies Bob has.

// Store the answer in a vector.

for (int aliceCandy : aliceSizes) {

if (bobSizeSet.count(targetSize)) {

vector<int> result;

#include <unordered set> // Include library for unordered set

public int[] fairCandySwap(int[] aliceSizes, int[] bobSizes) {

if target size bob in bob sizes set:

Iterate through Alice's candy sizes to find the fair swap for candy size alice in alice sizes: # Calculate the target size of Bob's candy that would make the swap fair

So, by utilizing the solution approach, we find that Alice can give Bob a box of 5 candies, and Bob can give Alice a box of 4

Since we found 4 (which is target) in Bob's set when Alice's box size was 5, the pair [5, 4] is returned.

```
# Note: no need for an explicit return statement for no match scenario
        # because the question implies there is always a valid swap.
Java
```

class Solution {

class Solution {

public:

Solution Implementation

bob_sizes_set = set(bob_sizes)

Python

class Solution:

```
// Calculate sum of candies for Alice
for (int candySize : aliceSizes) {
    sumAlice += candySize;
// Calculate sum of candies for Bob and populate the set with Bob's candy sizes
for (int candySize : bobSizes) {
    bobCandies.add(candySize);
    sumBob += candySize;
// Compute the difference to be balanced, divided by 2
int balanceDiff = (sumAlice - sumBob) >> 1;
// Iterate through Alice's candies to find the fair swap
for (int aliceCandySize : aliceSizes) {
    int targetSize = aliceCandySize - balanceDiff;
    // Check if Bob has the candy size that would balance the swap
    if (bobCandies.contains(targetSize)) {
        // Return the pair that represents the fair swap
        return new int[]{aliceCandySize, targetSize};
// If no fair swap is possible, return null
return null;
```

```
};
```

```
return result; // Return the pair representing a fair swap.
TypeScript
function fairCandySwap(aliceCandySizes: number[], bobCandySizes: number[]): number[] {
    // Calculate the sum of candies for both Alice and Bob.
    let sumAlice = aliceCandySizes.reduce((accumulated, current) => accumulated + current, 0);
    let sumBob = bobCandySizes.reduce((accumulated, current) => accumulated + current, 0);
    // Calculate the difference in candies between Alice and Bob, divided by 2.
    let halfDiff = (sumAlice - sumBob) >> 1;
    // Create a set from Bob's candy sizes for constant-time lookups.
    let bobSizesSet = new Set(bobCandySizes);
    // Loop through each of Alice's candy sizes to find a fair swap.
    for (let aliceCandy of aliceCandySizes) {
        // Calculate the target size for Bob that would equalize the sum.
        let targetBobCandy = aliceCandy - halfDiff;
        // Check if the target candy size exists in Bob's collection.
        if (bobSizesSet.has(targetBobCandy)) {
            // If found, return the pair of candy sizes for Alice and Bob.
            return [aliceCandy, targetBobCandy];
    // If no fair swap is found, return an empty array.
    // (The problem statement assures a solution always exists, so this line is never expected to be reached in practice.)
    return [];
class Solution:
    def fairCandySwap(self, alice sizes: List[int], bob sizes: List[int]) -> List[int]:
        # Compute the difference in candy sizes between Alice and Bob, divided by 2
        # because any swap should compensate for half of the total size difference.
        size_difference = (sum(alice_sizes) - sum(bob_sizes)) // 2
```

because the question implies there is always a valid swap. Time and Space Complexity

bob sizes set = set(bob sizes)

for candy size alice in alice sizes:

Time Complexity: The time complexity of the given code is O(A + B), where A is the number of elements in aliceSizes and B is the number of elements in bobSizes. Here's the breakdown: calculating the sum of both arrays takes O(A) and O(B) time respectively; creating the set s of bobSizes takes O(B) time; and the for loop runs for every element in aliceSizes, giving another O(A). Since set look-up is O(1) on average, checking if target is in s does not significantly add to the complexity. Thus, the overall time complexity is the sum of these operations, which simplifies to O(A + B).

Space Complexity: The space complexity of the code is O(B). This is because we create a set s consisting of all elements in bobSizes, which takes up space proportional to the number of elements in bobSizes. No other significant space is used as the rest of the variables use constant space.