927. Three Equal Parts

Hard Array Math

Leetcode Link

Problem Description

In this combinatorial problem, we are given an array `arr` consisting only of 0s and 1s. The goal is to divide this array into three contiguous, non-empty parts, such that each part represents an identical binary value when converted to a decimal number. If such a tripartition is possible, we return the indices `[i, j]` which separate the three parts:

- The first part spans from `arr[0]` to `arr[i]`,
- The second part spans from `arr[i+1]` to `arr[j-1]`, and
- The third part spans from `arr[j]` to `arr[arr.length - 1]`.

It's important to note that all three parts must represent the same binary value. The representation includes leading zeros, which means that sequences like `[0, 1, 1]` and `[1, 1]` are considered equivalent. If division into such three parts is not feasible, we must return `[-1, -1]`.

Intuition

The approach to solving this problem involves understanding that if we can divide the array into three partitions with equal binary value, the number of 1's in each part must be equal, because binary numbers are represented by the sequence of 1's they contain, and leading zeros do not change their value.

Following this logic:

1. We first count the total number of 1's in the array. If this number is not divisible by 3, we cannot possibly divide the array into three parts with an equal number of 1's, and we return `[-1, -1]`.
2. If the total number of 1's is zero, that means the array contains only 0's, and we can return any valid partition such as `[0, n - 1]` where `n` is the length of the array.
3. In other cases, we settle for an iterative approach to find the actual partition:
   - We find the index of the 1st `1` in each of the three parts. These will be the start points for comparing the parts.
   - Since `ont` is the total count of 1's in each partition, we find:
     - The 1st `1` at `find(0)`,
     - The 1st `1` in the second part at `find(ont + 1)`
     - The 1st `1` in the third part at `find(ont * 2 + 1)`
4. Once we have these starting points, we compare each subsequent digit of the three parts to ensure they match. If they do, we move all indices to the next position.
5. We continue this until we've compared all digits or find a mismatch. If we reach the end of the array and all digits have matched, we return `[i - 1, j]`, which represents the required partition. If there's a mismatch, the arrangement is not possible, and we return `[-1, -1]`.

This approach utilizes two key ideas: equal number of 1's in each partition and the rule that leading zeros do not alter the binary number's value for the solution.

Solution Approach

The provided Python solution begins with a helper function `find(x)` used to locate the index of the `x`-th `1` within the array. This function iterates over the `arr` and keeps a count `c` of the number of 1's seen so far. When `c` equals `x`, the function returns the current index, effectively marking the boundary of a part.

Here is the step-by-step breakdown of the algorithm:

1. **Counting 1's and Pre-validation**: The total number of 1's in the array is counted using `sum(arr)`. The algorithm then checks if this sum is divisible by 3. If not, the parts cannot have equal binary values, and the function returns `[-1, -1]`. If there are no 1's, the array can be split anywhere to create equal parts of zeros, and thus `[0, n - 1]` is returned.
2. **Finding the Initial Indices for Comparison**: The algorithm finds the indices `i`, `j`, and `k` which point to the first 1 in each of the three parts we want to compare. This is accomplished by calling `find()` for the first 1, `find(cnt + 1)` for the first 1 in the second part, and `find(cnt * 2 + 1)` for the first 1 in the third part, where `cnt` is the count of 1's in any part.
3. **Comparing the Three Parts**: By initializing `i`, `j`, and `k` to the start of each part, the algorithm enters a `while` loop which continues until the index `k` reaches the end of the array. Inside the loop, it checks whether the current bits at `arr[i]`, `arr[j]`, and `arr[k]` are equal. If they are equal, it means the parts are still matching, and it increments `i`, `j`, and `k` one position forward to compare the next set of bits.
4. **Determining the Correct Split Point**: The loop terminates when the end of the array is reached (`k == n`) or a mismatch is found. If the end of the array is reached without encountering a mismatch, it means the three parts match and the function returns the split points `[i - 1, j]`. If the while loop exits due to a mismatch, it means equal partitioning is not possible, and thus `[-1, -1]` is returned.

This solution employs a single pass to count `1`s and a single traversal for comparison. It uses no additional data structures and solely relies on array indexing. The solution pattern hinges on the continual comparison of triplet indices, moving forward collectively to maintain the synchrony and alignment of the three parts. This is an example of a two-pointer technique, where multiple pointers are used to iterate over the data structure, updating their positions based on certain conditions.

Example Walkthrough

Let's consider an example `arr = [1, 0, 1, 0, 1, 0]` to illustrate the solution approach.

1. **Counting 1's and Pre-validation**: We start by counting the number of `1`s in the array. There are a total of `3` `1`s, which is divisible by 3. That means we can potentially divide `arr` into three parts with an equal number of `1`s in each part.
2. **Finding the Initial Indices for Comparison**: We want to find the indices for the first `1` in all three parts.
   - The first `1` appears at index `0`. We set `i` to this value.
   - The second `1` is part of the second segment. Since there's one `1` in the first segment, we search for the `(1+1)`-th `1`, which is at index `2`. We set `j` to point to the following position, which is `3`.
   - The third `1` is part of the third segment. We search for the `(2*1+1)`-th `1`, which is at index `4`. We set `k` to point to this index.

By the end of this step, we have `i = 0`, `j = 3`, and `k = 4`.

3. **Comparing the Three Parts**: Now we begin comparing the elements pointed by `i`, `j`, and `k`.
   - At our starting indices the values at `arr[i]`, `arr[j]`, and `arr[k]` are `1`, `0`, and `1`, respectively. They are not all equal, so we move the `i` index forward to try to match the values in the next iteration.
   - After incrementing, we have `i = 1`, `j = 3`, and `k = 4`. The values are `0`, `0`, and `1`. The values at `j` and `k` are different, so we increment `i` to try to match in the next iteration.
   - With `i = 1`, `j = 4`, and `k = 4`, now we have `arr[i]`, `arr[j]`, and `arr[k]` all at `0`. We increment all indices to find the next value.
   - With `i = 2`, `j = 5`, and `k = 5`, the values are `1`, `0`, and `0`. Now, since `arr[j]` and `arr[k]` are equal but different from `arr[i]`, we increment `i` until `arr[i]` matches.
   - This process continues until `i`, `j`, and `k` move past the end of `arr` or until we find a mismatch that we can't resolve by incrementing the pointers.
4. **Determining the Correct Split Point**: In this example, we see that `i` can move to `2`, making `arr[i]` a `1`, which gives us `arr[i]`, `arr[j]`, and `arr[k]` values of `1`, `0`, and `0`. There seems to be a mismatch, however, note that the parts don't need to be equal only the binary value they represent need to be equal. Since leading zeros can be ignored in the binary value representation, in our current positions (`i = 2`, `j = 5`, `k = 5`), the partition yields `arr[0]` to `arr[i]` as `[1,0]`, `arr[i+1]` to `arr[j-1]` as `[0]` and the last part `arr[j]` to `arr[arr.length - 1]` as an empty array which after considering leading zeros could also be `[0]`. The binary value all these parts corresponding to is `1` in decimal, so the array can indeed be partitioned. The correct split points are `[i, j]`, and after decrementing `i` by one to exclude it from the second part, we have `arr[0..1]`, `arr[2..4]`, and `arr[5..5]`. Thus, we return `[1, 5]`.

In this case, the binary value for all three parts is `1`, which is equal for each partition. Hence, we can return `[1, 5]` which are the required indices to divide the array into three parts of equal binary value.

Python Solution

```python
class Solution:
    def threeEqualParts(self, arr):
        # Helper function to find the index of the partition
        def find(x, start=0, target_sum=1):
            running_sum = 0
            for index, value in enumerate(arr):
                running_sum += value
                if running_sum == target_sum:
                    return index

        n = len(arr) # Length of the array
        total_ones, remainder = divmod(sum(arr), 3)  # Total ones and remainder when divided by 3

        # If the number of 1's is not divisible by 3, return [-1, -1]
        if remainder != 0:
            return [-1, -1]

        # If there are no ones, any index can be the partition, choose 0 and n-1
        if total_ones == 0:
            return [0, n - 1]

        # Find the start index of each part
        part1_index = find(0)  # Start of the first part
        part2_index = find(0, partition=total_ones + 1)
        part3_index = find(0, partition=2 * total_ones + 1)

        # Match bits of the three parts
        while part3_index < n and arr[part1_index] == arr[part2_index] == arr[part3_index]:
            part1_index += 1
            part2_index += 1
            part3_index += 1

        # If part3_index reached the end, we found a valid partition
        if part3_index == n:
            return [part1_index - 1, part2_index]

        # Otherwise, no valid partition was found
        return [-1, -1]
```

Java Solution

```java
class Solution {
    private int[] binaryArray; // Renamed array for clarity

    public int[] threeEqualParts(int[] arr) {
        binaryArray = arr; // More descriptive naming
        int countOfOnes = 0, n = arr.length; // More descriptive naming
        int arrayLength = arr.length;

        // Count the number of 1s in the array
        for (int value : arr) {
            countOfOnes += value;
        }

        // If count of 1s is not divisible by three, solution is impossible
        if (countOfOnes % 3 != 0) {
            return new int[] {-1, -1};
        }

        // If array has no 1s, any split is valid (all zeros)
        if (countOfOnes == 0) {
            return new int[] {0, arrayLength - 1};
        }

        // Each part must contain an equal number of 1s
        countOfOnes /= 3;

        // Find the index of three subarrays with an equal number of 1s
        int first = findNthOne(1);
        int second = findNthOne(countOfOnes + 1);
        int third = findNthOne(countOfOnes * 2 + 1);

        // Check if all three parts are equal
        while (third < arrayLength) {
            if (binaryArray[first] == binaryArray[second] &&
                binaryArray[second] == binaryArray[third]) {
                ++first;
                ++second;
                ++third;
            } else {
                break;
            }
        }

        // If end of the array is reached, return valid split positions, else return {-1, -1}
        return third == arrayLength ?
                new int[] {first - 1, second} :
                new int[] {-1, -1};
    }

    // Helper method to find the start index of the n-th one in the binary array
    private int findNthOne(int nthOne) {
        int sum = 0;
        for (int i = 0; i < binaryArray.length; ++i) {
            sum += binaryArray[i];
            if (sum == nthOne) {
                return i;
            }
        }
        return -1; // Return -1 if the nth one is not found
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <numeric> // For std::accumulate

class Solution {
public:
    vector<int> threeEqualParts(vector<int>& arr) {
        int totalOnes = accumulate(arr.begin(), arr.end(), 0); // Count total number of 1's in the array

        // If total number of 1's is not divisible by 3, we can't partition the array into three parts with equal no of 1's.
        if (totalOnes % 3 != 0) return {-1, -1};

        // If there are no 1's, any partition is valid (since all are zeros)
        if (totalOnes == 0) return {0, static_cast<int>(arr.size()) - 1};

        int onesPerPart = totalOnes / 3; // Number of 1's each part must have

        // Lambda function to find the first index where the cumulative number of 1's equals 'targetOnes'
        auto findIndexWithCumulativeOnes = [&](int targetOnes) {
            int cumulativeOnes = 0;
            for (int i = 0; i < arr.size(); ++i) {
                cumulativeOnes += arr[i];
                if (cumulativeOnes == targetOnes) return i;
            }
            return -1; // Return -1 if target is not found (should not happen if input is valid)
        };

        // Find the ending indexes of parts with 1/3, 2/3, and all of the 1's
        int firstThirdEnd = findIndexWithCumulativeOnes(1);
        int secondPartEnd = findIndexWithCumulativeOnes(onesPerPart + 1);
        int thirdThirdEnd = findIndexWithCumulativeOnes(2 * onesPerPart + 1);

        // Ensure each corresponding position in the three parts contains the same value
        while (thirdThirdEnd < arr.size() && arr[firstThirdEnd] == arr[secondPartEnd] && arr[secondPartEnd] == arr[thirdThirdEnd]) {
            ++firstThirdEnd;
            ++secondPartEnd;
            ++thirdThirdEnd;
        }

        // If end of array is reached, return valid partition points.
        // Otherwise, return {-1, -1} since parts are not equal.
        return thirdThirdEnd == arr.size() ? vector<int>{firstThirdEnd - 1, secondPartEnd} : vector<int>{-1, -1};
    }
};
```

Typescript Solution

```typescript
/**
 * Function to find the split points for the array such that all parts are equal.
 * @param arr - Array of 0s and 1s.
 * @returns An array with two indices that represent the split points, if valid, else returns [-1, -1].
 */
function threeEqualParts(arr: number[]): [number, number] {
    // A helper function to find the end index of the subarray that makes the sum x.
    const findEnd = (x: number, startIndex: number): number => {
        let sum = 0;
        for (let i = startIndex; i < n; ++i) {
            sum += arr[i];
            if (sum === targetSum) {
                return i;
            }
        }
        return -1;
    };

    const n = arr.length;
    let totalOnes = arr.reduce((a, b) => a + b); // Count the total number of 1s in the array.

    // Counting the number of 1's in the array.
    for (const value of arr) {
        totalOnes += value;
    }

    // If total number of 1's is not divisible by 3, then it's not possible to split the array into three equal parts.
    if (totalOnes % 3 !== 0) {
        return [-1, -1];
    }

    // If there are no 1's, we can split anywhere.
    if (totalOnes === 0) {
        return [0, n - 1];
    }

    // The number of 1s in each part.
    const onesPerPart = number = Math.floor(totalOnes / 3);

    // Find the first index in each part of the array.
    let firstEndIndex = find(1, 0);
    let secondEndIndex = find(onesPerPart + 1, firstEndIndex + 1) + 1;
    let thirdEndIndex = find(2 * onesPerPart + 1, secondEndIndex) + 1;

    // Now we just have to check that the remaining parts of the array are the same.
    while(thirdEndIndex < n && arr[firstEndIndex] === arr[secondEndIndex] && arr[secondEndIndex] === arr[thirdEndIndex]) {
        firstEndIndex += 1;
        secondEndIndex += 1;
        thirdEndIndex += 1;
    }

    // If we reached the end of the array, it means we found a valid split.
    if (thirdEndIndex === n) {
        return [firstEndIndex, secondEndIndex];
    }

    // If no valid split is found, return [-1, -1].
    return [-1, -1];
}
```

Time and Space Complexity

The given Python function `threeEqualParts` is intended to solve a problem where, given an array of binary digits, the goal is to split the array into three non-empty parts that all represent the same binary value.

Time Complexity

The time complexity of the code is composed of several parts:

1. `sum(arr)`: This operation traverses the entire array to compute the sum, and has a time complexity of $O(n)$ where n is the length of the array.
2. Three separate calls to `find(x)`: Each call to `find(x)` will, in the worst case, traverse the entire array once, therefore the time complexity for all three calls combined is $O(n)$.
3. The while loop: This loop continues incrementing `i`, `j`, and `k` until `k` reaches the end of the array or a mismatch is found. In the worst case, this leads to traversing the array again, adding another $O(n)$ to the complexity.

The total time complexity is the sum of these parts, leading to $O(n) + O(n) + O(n)$ which simplifies to $O(n)$.

Space Complexity

The space complexity is determined by the amount of additional memory used by the algorithm, which is mainly:

1. Constant extra space for variables `s`, `cnt`, `mod`, `i`, `j`, `k`, and `n`, which does not grow with the input size.
2. No additional data structures that grow with input size are used.

Therefore, the space complexity is $O(1)$, which signifies constant space.

In summary, the time complexity of the function is $O(n)$ and the space complexity is $O(1)$.