927. Three Equal Parts Math

**Leetcode Link** 

## 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

**Problem Description** 

Array

Hard

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],

and leading zeros do not change their value.

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

### 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,

return  $\begin{bmatrix} -1, & -1 \end{bmatrix}$ .

Intuition The approach to solving this problem involves understanding that if we can divide the array into three partitions with equal binary

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]

Following this logic:

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 cnt is the total count of 1's in each partition, we find: The 1st 1 at find(1) The 1st 1 in the second part at find(cnt + 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.

The 1st 1 in the third part at find(cnt \* 2 + 1)

- 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

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

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 s of the number of 1's seen so far. When s 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

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

array can be split anywhere to create equal parts of zeros, and thus [0, n - 1] is returned.

three parts we want to compare. This is accomplished by calling find(1) 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 1s in any part.

returned.

**Example Walkthrough** 

return [-1, -1].

number's value for the solution.

**Solution Approach** 

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

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

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.

This solution employs a single pass to count 1s and a single traversal for comparison. It uses no additional data structures and solely

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.

• 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

1. Counting 1's and Pre-validation: We start by counting the number of 1s in the array. There are a total of 3 1s, which is divisible

by 3. That means we can potentially divide arr into three parts with an equal number of 1s in each part.

#### move the i index forward to try to match the values in the next iteration. $\circ$ 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

[1, 5].

**Python Solution** 

class Solution:

9

10

11

12

13

14

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

37

38

39

40

4

51

52

53

54

55

56

57

58

59

6

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

C++ Solution

#include <vector>

**}**;

4 class Solution {

public:

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

increment j to try to match in the next iteration.

we increment i until arr[i] matches.

n = len(arr) # Length of the array

# Find the start index of each part

part2\_index = find\_partition(total\_ones + 1)

# Otherwise, no valid partition was found

public int[] threeEqualParts(int[] arr) {

sum += binaryArray[i];

return -1; // Return -1 if the nth one is not found

if (sum == nth0ne) {

return i;

#include <numeric> // For std::accumulate

vector<int> threeEqualParts(vector<int>& arr) {

if (totalOnes % 3 != 0) return {-1, -1};

int cumulativeOnes = 0;

++firstPartEnd;

++thirdPartEnd;

++secondPartEnd;

// If there are no 1's, any partition is valid (since all are zeros)

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

// Find the ending indexes of parts with 1/3, 2/3, and all of the 1's

int thirdPartEnd = findIndexWithCumulativeOnes(2 \* onesPerPart + 1);

// Ensure each corresponding position in the three parts contains the same value

int secondPartEnd = findIndexWithCumulativeOnes(onesPerPart + 1);

// If end of array is reached, return valid partition points,

// Otherwise, return  $\{-1, -1\}$  since parts are not equal.

if (totalOnes == 0) return {0, static\_cast<int>(arr.size()) - 1};

auto findIndexWithCumulativeOnes = [&](int targetOnes) {

if (cumulativeOnes == targetOnes) return i;

for (int i = 0; i < arr.size(); ++i) {</pre>

int firstPartEnd = findIndexWithCumulativeOnes(1);

cumulativeOnes += arr[i];

private int[] binaryArray; // Renamed array for clarity

part3\_index = find\_partition(2 \* total\_ones + 1)

part1\_index = find\_partition(1)

# Match bits of the three parts

if remainder != 0:

if total\_ones == 0:

return [-1, -1]

return [0, n - 1]

part1\_index += 1

return [-1, -1]

binaryArray = arr;

Java Solution

class Solution {

# If the number of 1's is not divisible by 3, return [-1, -1]

# If there are no ones, any index can be the partition, choose 0 and n-1

while part3\_index < n and arr[part1\_index] == arr[part2\_index] == arr[part3\_index]:</pre>

incrementing the pointers.

 $\circ$  With i = 1, j = 4, and k = 4, now we have arr[i], arr[j], and arr[k] all 0. We increment all indices to find the next value.  $\circ$  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],

3. Comparing the Three Parts: Now we begin comparing the elements pointed by i, j, and k.

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

arr[j], and arr[k] values of 1, 0, and 0. There seems to be a mismatch, however, we 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 [1] and the

last part arr[j] to arr[arr.length - 1] as an empty array which after considering leading zeros could also be [1]. The binary

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

value all three parts correspond to is 1 in decimal, so the array can indeed be partitioned. The correct split points are [i, j], and

• 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

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],

- 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.
  - def threeEqualParts(self, arr): # Helper function to find the index of the partition def find\_partition(target\_sum): running\_sum = 0 for index, value in enumerate(arr): running\_sum += value if running\_sum == target\_sum: return index return -1

total\_ones, remainder = divmod(sum(arr), 3) # Total ones and remainder when divided by 3

31 part2\_index += 1 32 part3\_index += 1 33 34 # If part3\_index reached the end, we found a valid partition 35 ir part3\_index == n: return [part1\_index - 1, part2\_index] 36

```
int countOfOnes = 0; // More descriptive naming
 6
            int arrayLength = arr.length;
 8
            // Count the number of 1s in the array
10
            for (int value : arr) {
                countOfOnes += value;
11
12
13
14
            // If count of 1s is not divisible by three, solution is impossible
            if (countOfOnes % 3 != 0) {
15
                return new int[] {-1, -1};
16
17
18
            // If array has no 1s, any split is valid (all zeros)
19
            if (countOfOnes == 0) {
20
                return new int[] {0, arrayLength - 1};
21
22
23
24
            // Each part must contain an equal number of 1s
25
            countOfOnes /= 3;
26
27
            // Find the start index of three subarrays with an equal number of 1s
            int first = findNth0ne(1),
                second = findNthOne(countOfOnes + 1),
29
30
                third = findNth0ne(2 * countOfOnes + 1);
31
32
            // Check if all three parts are equal
33
            while (third < arrayLength &&</pre>
                   binaryArray[first] == binaryArray[second] &&
34
35
                   binaryArray[second] == binaryArray[third]) {
36
                ++first;
37
                ++second;
38
                ++third;
39
40
            // If end of the array is reached, return valid split positions, else return [-1, -1]
41
42
            return third == arrayLength ?
43
                   new int[] {first - 1, second} :
44
                   new int[] {-1, -1};
45
46
47
        // Helper method to find the start index of the n-th one in the binary array
48
        private int findNthOne(int nthOne) {
            int sum = 0;
49
            for (int i = 0; i < binaryArray.length; ++i) {</pre>
50
```

int totalOnes = accumulate(arr.begin(), arr.end(), 0); // Count total number of 1's in the array

// Lambda function to find the first index where the cumulative number of 1's equals 'targetOnes'

return -1; // Return -1 if target is not found (should not happen if input is valid)

// 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.

while (thirdPartEnd < arr.size() && arr[firstPartEnd] == arr[secondPartEnd] && arr[secondPartEnd] == arr[thirdPartEnd]) {</pre>

return thirdPartEnd == arr.size() ? vector<int>{firstPartEnd - 1, secondPartEnd} : vector<int>{-1, -1};

#### 42 43 **}**; 44

```
Typescript Solution
  1 /**
     * Function to find the split points for the array such that all parts are equal.
     * @param arr An 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 find = (targetSum: number, startIndex: number): number => {
  8
             let sum = 0;
  9
             for (let i = startIndex; i < n; ++i) {</pre>
 10
                 sum += arr[i];
 11
 12
                 if (sum === targetSum) {
 13
                     return i;
 14
 15
 16
             return -1;
 17
         };
 18
 19
         const n: number = arr.length;
 20
         let totalOnes: number = 0; // Count the total number of 1s in the array.
 21
 22
         // Counting the number of 1's in the array.
 23
         for (const value of arr) {
 24
             totalOnes += value;
 25
 26
 27
         // If total number of 1's is not divisible by 3, then it's not possible to split the array into three equal parts.
 28
         if (totalOnes % 3) {
 29
             return [-1, -1];
 30
 31
 32
         // If there are no 1's, we can split anywhere.
 33
         if (totalOnes === 0) {
             return [0, n - 1];
 34
 35
 36
 37
        // The number of 1s in each part.
         const onesPerPart: number = Math.floor(totalOnes / 3);
 38
 39
 40
         // Find the first index in each part of the array.
         let firstIndexEnd = find(1, 0);
 41
         let secondIndexStart = find(onesPerPart + 1, firstIndexEnd + 1) + 1;
 42
 43
         let thirdIndexStart = find(2 * onesPerPart + 1, secondIndexStart) + 1;
 44
 45
         // Now we just have to check that the remaining parts of the array are the same.
        while(thirdIndexStart < n && arr[firstIndexEnd] === arr[secondIndexStart] && arr[secondIndexStart] === arr[thirdIndexStart]) {</pre>
 46
             firstIndexEnd += 1;
 47
             secondIndexStart += 1;
 48
 49
             thirdIndexStart += 1;
 50
 51
 52
         // If we reached the end of the array, it means we found a valid split.
 53
         if (thirdIndexStart === n) {
 54
             return [firstIndexEnd, secondIndexStart];
 55
 56
 57
        // If no valid split is found, return [-1, -1].
 58
         return [-1, -1];
 59
 60
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

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

# **Time Complexity**

of the array.

**Space Complexity** 

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

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

the array into three non-empty parts that all represent the same binary value.

- The total time complexity is the sum of these parts, leading to O(n) + O(n) + O(n) which simplifies to O(n).
  - The space complexity is determined by the amount of additional memory used by the algorithm, which is mainly: 1. Constant extra space for variables n, cnt, mod, i, j, k, and s, which does not grow with the input size.

worst case, this leads to traversing the array again, adding another O(n) to the complexity.

Therefore, the space complexity is 0(1), which signifies constant space.

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

2. No additional data structures that grow with input size are used.