

## Contiguous Array New

Given a binary array, find the maximum length of a contiguous subarray with equal number of 0 and 1.

### Example 1:

**Input:** [0,1]

**Output:** 2

**Explanation:** [0, 1] is the longest contiguous subarray with equal number of 0 and 1.

### Example 2:

**Input:** [0,1,0]

**Output:** 2

**Explanation:** [0, 1] (or [1, 0]) is a longest contiguous subarray with equal number of 0 and 1.

**Note:** The length of the given binary array will not exceed 50,000.

## Solution 1

The idea is to change `0` in the original array to `-1`. Thus, if we find `SUM[i, j] == 0` then we know there are even number of `-1` and `1` between index `i` and `j`. Also put the `sum` to `index` mapping to a HashMap to make search faster.

```
public class Solution {
    public int findMaxLength(int[] nums) {
        for (int i = 0; i < nums.length; i++) {
            if (nums[i] == 0) nums[i] = -1;
        }

        Map<Integer, Integer> sumToIndex = new HashMap<>();
        sumToIndex.put(0, -1);
        int sum = 0, max = 0;

        for (int i = 0; i < nums.length; i++) {
            sum += nums[i];
            if (sumToIndex.containsKey(sum)) {
                max = Math.max(max, i - sumToIndex.get(sum));
            }
            else {
                sumToIndex.put(sum, i);
            }
        }

        return max;
    }
}
```

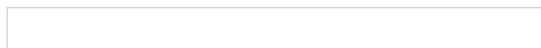
written by [shawngao](#) original link [here](#)

## Solution 2

My idea is very similar to others, but let me try to explain it more visually. My thought was inspired by 121. Best Time to Buy and Sell Stock.

Let's have a variable `count` initially equals 0 and traverse through `nums`. Every time we meet a 0, we decrease `count` by 1, and increase `count` by 1 when we meet 1. It's pretty easy to conclude that we have a contiguous subarray with equal number of 0 and 1 when `count` equals 0.

What if we have a sequence `[0, 0, 0, 0, 1, 1]`? the maximum length is 4, the `count` starting from 0, will equal -1, -2, -3, -4, -3, -2, and won't go back to 0 again. But wait, the longest subarray with equal number of 0 and 1 started and ended when `count` equals -2. We can plot the changes of `count` on a graph, as shown below. Point (0,0) indicates the initial value of `count` is 0, so we count the sequence starting from index 1. The longest subarray is from index 2 to 6.



From above illustration, we can easily understand that two points with the same y-axis value indicates the sequence between these two points has equal number of 0 and 1.

Another example, sequence `[0, 0, 1, 0, 0, 0, 1, 1]`, as shown below,



There are 3 points have the same y-axis value -2. So subarray from index 2 to 4 has equal number of 0 and 1, and subarray from index 4 to 8 has equal number of 0 and 1. We can add them up to form the longest subarray from index 2 to 8, so the maximum length of the subarray is  $8 - 2 = 6$ .

Yet another example, sequence `[0, 1, 1, 0, 1, 1, 1, 0]`, as shown below. The longest subarray has the y-axis value of 0.



To find the maximum length, we need a dict to store the value of `count` (as the key) and its associated index (as the value). We only need to save a `count` value and its index at the first time, when the same `count` values appear again, we use the new index subtracting the old index to calculate the length of a subarray. A variable `max_length` is used to keep track of the current maximum length.

```
class Solution(object):
    def findMaxLength(self, nums):
        count = 0
        max_length=0
        table = {0: 0}
        for index, num in enumerate(nums, 1):
            if num == 0:
                count -= 1
            else:
                count += 1

            if count in table:
                max_length = max(max_length, index - table[count])
            else:
                table[count] = index

        return max_length
```

written by [NoAnyLove](#) original link [here](#)

## Solution 3

Keeping track of the balance (number of ones minus number of zeros) and storing the first index where each balance occurred.

---

### Python

Keeping the balance in units of 0.5 which makes the update expression short (not that `num * 2 - 1` or `1 if num else -1` would be terribly long):

```
def findMaxLength(self, nums):
    index = {0: -1}
    balance = maxlen = 0
    for i, num in enumerate(nums):
        balance += num - 0.5
        maxlen = max(maxlen, i - index.setdefault(balance, i))
    return maxlen
```

Just for fun as an ugly one-liner:

```
def findMaxLength(self, nums):
    return reduce(lambda f, b, m, (i, x): (f, b+x-.5, max(m, i-f.setdefault(b+x-.5, i))),
        enumerate(nums), ({0:-1}, 0, 0))[2]
```

---

### Java

Using `putIfAbsent` so I only need one map function call per number.

```
public int findMaxLength(int[] nums) {
    Map<Integer, Integer> index = new HashMap<>();
    index.put(0, -1);
    int balance = 0, maxlen = 0;
    for (int i = 0; i < nums.length; i++) {
        balance += nums[i] * 2 - 1;
        Integer first = index.putIfAbsent(balance, i);
        if (first != null)
            maxlen = Math.max(maxlen, i - first);
    }
    return maxlen;
}
```

Could avoid using `Math.max` like this:

```
if (first != null && i - first > maxlen)
    maxlen = i - first;
```

written by [StefanPochmann](#) original link [here](#)

