(6)找到0和1个数相等的最大子数组

给定一个数组，它的元素是0和1，请找出最大的子数组，使这个子数组的元素中0和1的个数相等。

例如，**输入**：
{ 0, 0, 1, 0, 1, 0, 0 }
输出：
Largest subarray is { 0, 1, 0, 1 } or { 1, 0, 1, 0}

我们马上能想到的一种算法是遍历所有的子数组，对于每个子数组，我们计算它包含的所有0和1的个数。如果子数组中包含的0和1的个数相等，我们就判断此子数组的长度是否和之前记录的最长子数组相等，如果它更大，那么就记录它，直到遍历完所有的子数组。这种方案的时间复杂度是O(n3)，因为有n2个子数组，并且我们必须话费O(n)的时间去遍历每个子数组，以求出它包含的0和1的个数。当然，通过优化，我们可以在O(1)时间内计算出0和1的个数，这样这种方法可以优化到O(n2)的时间复杂度。

还有另一种方法，可以在线性时间内解决问题。这种方案是这样，把数组中的元素0替换成-1，并找出和等于0的最大子数组。为了找到和等于0的最大子数组，我们可以创建一个空的Map用以存储和等于给定值的所有子数组的结束下标。我们遍历整个数组，然后维护到目前为止便利到的所有元素。
 如果和是第一次见到，那么就把和插入到Map中作为Key，index作为Value。

如果和在之前碰到过，那么一定存在一个子数组使它的所有元素之和等于0，它的结束下标就是当前index。如果当前子数组的长度比之前保存的最长子数组长度更长，那么我们就更新它。

c++语言实现：

```
#include <bits/stdc++.h>
using namespace std;

// Function to find maximum length sub-array having equal number
// of 0's and 1's
void maxLenSubarray(int arr[], int n)
{
    // create an empty map to store ending index of first sub-array
```

```
// having some sum
unordered_map<int, int> map;

// insert (0, -1) pair into the set to handle the case when
// sub-array with sum 0 starts from index 0
map[0] = -1;

// len stores the maximum length of sub-array with sum 0
int len = 0;

// stores ending index of maximum length sub-array having sum 0
int ending_index = -1;

int sum = 0;

// Traverse through the given array
for (int i = 0; i < n; i++)
{
    // sum of elements so far (replace 0 with -1)
    sum += (arr[i] == 0)? -1 : 1;

    // if sum is seen before
    if (map.find(sum) != map.end())
    {
        // update length and ending index of maximum length
        // sub-array having sum 0
        if (len < i - map[sum])
        {
            len = i - map[sum];
            ending_index = i;
        }
    }
    // if sum is seen for first time, insert sum with its
    // index into the map
    else
        map[sum] = i;
}

// print the sub-array if present
if (ending_index != -1)
    cout << "[" << ending_index - len + 1 << ", " << ending_index << "]";
```

```
    else
        cout << "No sub-array exists";
}

// main function
int main()
{
    int arr[] = { 0, 0, 1, 0, 1, 0, 0 };
    int n = sizeof(arr) / sizeof(arr[0]);
    maxLenSubarray(arr, n);
    return 0;
}
```

Java语言实现：

```
import java.util.Map;
import java.util.HashMap;

class MaxLengthSubArray
{
    // Function to find maximum length sub-array having equal number
    // of 0's and 1's
    public static void maxLenSubarray(int arr[])
    {
        // n is length of the array
        int n = arr.length;

        // create an empty Hash Map to store ending index of first
        // sub-array having some sum
        Map<Integer, Integer> map = new HashMap<Integer, Integer>();

        // insert (0, -1) pair into the set to handle the case when
        // sub-array with sum 0 starts from index 0
        map.put(0, -1);

        // len stores the maximum length of sub-array with sum 0
        int len = 0;

        // stores ending index of maximum length sub-array having sum 0
        int ending_index = -1;
```

```
        int sum = 0;

        // Traverse through the given array
        for (int i = 0; i < n; i++)
        {
            // sum of elements so far (replace 0 with -1)
            sum += (arr[i] == 0)? -1: 1;

            // if sum is seen before
            if (map.containsKey(sum))
            {
                // update length and ending index of maximum length
                // sub-array having sum 0
                if (len < i - map.get(sum))
                {
                    len = i - map.get(sum);
                    ending_index = i;
                }
            }
            // if sum is seen for first time, insert sum with its
            // index into the map
            else
                map.put(sum, i);
        }

        // print the sub-array if present
        if (ending_index != -1)
            System.out.println("[" + (ending_index - len + 1) + ", " +
                               ending_index + "]");
        else
            System.out.println("No sub-array exists");
    }

    // main function
    public static void main (String[] args)
    {
        int arr[] = { 0, 0, 1, 0, 1, 0, 0 };
        maxLenSubarray(arr);
    }
}
```

输出: [1,4]

以上两种方案的时间复杂度都是O(n)，空间复杂度都是o(n)。