Longest Subarray with equal number of 0s and 1s

Given an array containing only 0s and 1s, find the largest subarray which contains equal no of 0s and 1s. The expected time complexity is O(n).

Examples:

```
Input: arr[] = {1, 0, 1, 1, 1, 0, 0}
Output: 1 to 6
(Starting and Ending indexes of output subarray)

Input: arr[] = {1, 1, 1, 1}
Output: No such subarray

Input: arr[] = {0, 0, 1, 1, 0}
Output: 0 to 3 Or 1 to 4
```

Method 1: Brute Force.

Approach: The brute force approach in these type of questions is to generate all the possible sub-arrays. Then firstly check whether the sub-array has equal number of **0's** and **1's** or not. To make this process easy take **cumulative sum** of the sub-arrays taking **0's as -1** and **1's** as it is. The point where **cumulative sum = 0** will signify that the sub-array from starting till that point has equal number of **0's** and **1's**. Now as this is a valid sub-array, compare it's size with the maximum size of such sub-array found till now.

Algorithm :

1. Use a starting a pointer which signifies the starting point of the sub-array.
2. Take a variable **sum=0** which will take the cumulative sum of all the sub-array elements.
3. Initialize it with value **1 if** the **value at starting point=1** else initialize it with **-1**.
4. Now start an inner loop and start taking the cumulative sum of elements following the same logic.
5. If the cumulative sum **(value of sum)=0** it signifies that the sub-array has equal number of **0's and 1's**.
6. Now compare its size with the size of the largest sub-array if it is greater store the first index of such sub-array in a variable and update the value of size.
7. Print the sub-array with the **starting index and size** returned by the above algorithm.

C++ Java

```
// A simple C++ program to find the largest
// subarray with equal number of 0s and 1s
#include <bits/stdc++.h>

using namespace std;

// This function Prints the starting and ending
// indexes of the largest subarray with equal
// number of 0s and 1s. Also returns the size
// of such subarray.

int findSubArray(int arr[], int n)
{
    int sum = 0;
    int maxsize = -1, startindex;
```

```
    // Pick a starting point as i
    for (int i = 0; i < n - 1; i++) {
        sum = (arr[i] == 0) ? -1 : 1;

        // Consider all subarrays starting from i
        for (int j = i + 1; j < n; j++) {
            (arr[j] == 0) ? (sum += -1) : (sum += 1);

            // If this is a 0 sum subarray, then
            // compare it with maximum size subarray
            // calculated so far
            if (sum == 0 && maxsize < j - i + 1) {
                maxsize = j - i + 1;
                startindex = i;
            }
        }
    }
    if (maxsize == -1)
        cout << "No such subarray";
    else
        cout << startindex << " to "
             << startindex + maxsize - 1;

    return maxsize;
}

/* Driver code*/
int main()
{
```

```
    int arr[] = { 1, 0, 0, 1, 0, 1, 1 };
    int size = sizeof(arr) / sizeof(arr[0]);

    findSubArray(arr, size);
    return 0;
}

// This code is contributed by rathbhupendra
```

Output:

```
0 to 5
```

Complexity Analysis:

- **Time Complexity:** O(n^2).
  As all the possible sub-arrays are generated using a pair of nested loops.
- **Auxiliary Space:** O(1).
  As no extra data structure is used which takes auxiliary space.

Method 2: Hashmap.

Approach: The concept of taking **cumulative sum, taking 0's as -1** will help us in optimizing the approach. While taking the cumulative sum, there are two cases when there can be a sub-array with equal number of 0's and 1's.

1. When **cumulative sum=0**, which signifies that sub-array from index **(0)** till present index has equal number of **0's and 1's**.
2. When we encounter a cumulative sum value which we have already encountered before, which means that sub-array from the **previous index+1** till the **present index** has equal number of 0's and 1's as they give a

In a nutshell this problem is equivalent to finding two indexes **i & j in array[] such that array[i] = array[j] and (j-i) is maximum**. To store the first occurrence of each unique cumulative sum value we use a **hash_map** wherein if we get that value again we can find the sub-array size and compare it with the maximum size found till now.

Algorithm :

1. Let input array be arr[] of size n and max_size be the size of output sub-array.
2. Create a temporary array sumleft[] of size n. Store the sum of all elements from arr[0] to arr[i] in sumleft[i].
3. There are two cases, the output sub-array may start from 0th index or may start from some other index. We will return the max of the values obtained by two cases.
4. To find the maximum length sub-array starting from 0th index, scan the sumleft[] and find the maximum i where sumleft[i] = 0.
5. Now, we need to find the subarray where subarray sum is 0 and start index is not 0. This problem is equivalent to finding two indexes i & j in sumleft[] such that sumleft[i] = sumleft[j] and j-i is maximum. To solve this, we create a hash table with size = max-min+1 where min is the minimum value in the sumleft[] and max is the maximum value in the sumleft[]. Hash the leftmost occurrences of all different values in sumleft[]. The size of hash is chosen as max-min+1 because there can be these many different possible values in sumleft[]. Initialize all values in hash as -1.
6. To fill and use hash[], traverse sumleft[] from 0 to n-1. If a value is not present in hash[], then store its index in hash. If the value is present, then calculate the difference of current index of sumleft[] and previously stored value in hash[]. If this difference is more than maxsize, then update the maxsize.
7. To handle corner cases (all 1s and all 0s), we initialize maxsize as -1. If the maxsize remains -1, then print there is no such subarray.

C++ Java

```
import java.util.HashMap;

class LargestSubArray1 {

    // Returns largest subarray with
    // equal number of 0s and 1s

    int maxLen(int arr[], int n)
    {
        // Creates an empty hashMap hM

        HashMap<Integer, Integer> hM
            = new HashMap<Integer, Integer>();

        // Initialize sum of elements
        int sum = 0;

        // Initialize result
        int max_len = 0;
        int ending_index = -1;
        int start_index = 0;

        for (int i = 0; i < n; i++) {
            arr[i] = (arr[i] == 0) ? -1 : 1;
        }

        // Traverse through the given array
```

Dash
All
Articles
Videos
Problems
Quiz
Contest

```
for (int i = 0; i < n; i++) {
    // Add current element to sum

    sum += arr[i];

    // To handle sum=0 at last index

    if (sum == 0) {
        max_len = i + 1;
        ending_index = i;
    }

    // If this sum is seen before,
    // then update max_len if required
    if (hM.containsKey(sum)) {
        if (max_len < i - hM.get(sum)) {
            max_len = i - hM.get(sum);
            ending_index = i;
        }
    }
    else // Else put this sum in hash table
        hM.put(sum, i);
}

for (int i = 0; i < n; i++) {
    arr[i] = (arr[i] == -1) ? 0 : 1;
}
```

```
            int end = ending_index - max_len + 1;
            System.out.println(end + " to " + ending_index);

            return max_len;
        }

        /* Driver program to test the above functions */
        public static void main(String[] args)
        {
            LargestSubArray1 sub = new LargestSubArray1();
            int arr[] = { 1, 0, 0, 1, 0, 1, 1 };
            int n = arr.length;

            sub.maxLen(arr, n);
        }
    }

    // This code has been by Mayank Jaiswal(mayank_24)
```

Output:

```
0 to 5
```

Complexity Analysis:

- **Time Complexity:** O(n).
  As the given array is traversed only once.

- **Auxiliary Space:** O(n).
  As **hash_map** has been used which takes extra space.