(5)找到和等于给定值的最长子数组

给定一个数组，它有n个元素，元素类型为整形。给定一个值，如果能找出1到多个子数组，使子数组中所有元素的和等于给定值。请求出这个多个子数组中最长的那个子数组。

例如，给定如下数组和值：
A[] = { 5, 6, -5, 5, 3, 5, 3, -2, 0 }
Sum = 8

那么，和为8的子数组有3个，分别是：
{ -5, 5, 3, 5 }
{ 3, 5 }
{ 5, 3 }
其中，最长的子数组是
{ -5, 5, 3, 5 }，它的长度为4。

一种最直接的方法就是遍历所有的子数组，并求出它们的和。如果子数组的和等于给定值，那么我们就更新最长子数组的值。这种方法的时间复杂度是O(n3)，因为有n2个子数组，按后还要花O(n)时间求出它所有元素的和。当然，如果把计算子数组的和优化成O(1)，那么这种方法的时间复杂度可以优化成O(n2)。

C++语言实现

```
#include <iostream>
#include <unordered_map>
using namespace std;

// Naive function to find maximum length sub-array with sum S present
// in the given array
void maxLengthSubArray(int arr[], int n, int S)
{
    // len stores the maximum length of sub-array with sum S
    int len = 0;

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

```
    // consider all sub-arrays starting from i
    for (int i = 0; i < n; i++)
    {
        int sum = 0;

        // consider all sub-arrays ending at j
        for (int j = i; j < n; j++)
        {
            // sum of elements in current sub-array
            sum += arr[j];

            // if we have found a sub-array with sum S
            if (sum == S)
            {
                // update length and ending index of max length sub-array
                if (len < j - i + 1)
                {
                    len =  j - i + 1;
                    ending_index = j;
                }
            }
        }
    }

    // print the sub-array
    cout << "[" << (ending_index - len + 1) << "," << ending_index << "]";
}

// main function
int main()
{
    int arr[] = { 5, 6, -5, 5, 3, 5, 3, -2, 0 };
    int sum = 8;

    int n = sizeof(arr)/sizeof(arr[0]);
    maxLengthSubArray(arr, n, sum);

    return 0;
}
```

Java语言实现：

```
class MaxLengthSubArray
{
   // Naive function to find maximum length sub-array with sum S present
   // in the given array
   public static void maxLengthSubArray(int arr[], int S)
   {
      // n is length of the array
      int n = arr.length;

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

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

      // consider all sub-arrays starting from i
      for (int i = 0; i < n; i++)
      {
         int sum = 0;

         // consider all sub-arrays ending at j
         for (int j = i; j < n; j++)
         {
            // sum of elements in current sub-array
            sum += arr[j];

            // if we have found a sub-array with sum S
            if (sum == S)
            {
               // update length & ending index of max length subarray
               if (len < j - i + 1)
               {
                  len =  j - i + 1;
                  ending_index = j;
               }
            }
         }
      }
```

```
        // print the sub-array
        System.out.println("[" + (ending_index - len + 1) + ", "
                           + ending_index + "]");
    }

    // main function
    public static void main (String[] args)
    {
        int arr[] = { 5, 6, -5, 5, 3, 5, 3, -2, 0 };
        int sum = 8;

        maxLengthSubArray(arr, sum);
    }
}
```

输出:
[2,5]

2

我们还可以使用map来解决这个问题，使时间复杂度降低到O(n)。这种方法的思想是：先创建一个空Map，然后用它存储所有第一个和(设为Sum)等于给定值的子数组的下标。我们遍历数组，然后维护便利过的元素的Sum。如果和是第一次碰到，然后把Sum与它的下标插入到数组中。如果(Sum-S)在之前碰到过，意味着存在一个子数组，它的和（Sum）等于给定值，它的结束下标就是当前下标。如果当前数组的长度更长，那么我就需要更新和等于S的最大子数组的长度。

C++语言实现：

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

// Function to find maximum length sub-array with sum S present
// in the given array
void maxLengthSubArray(int arr[], int n, int S)
{
    // 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 S starts from index 0
    map[0] = -1;

    int sum = 0;

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

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

    // traverse the given array
    for (int i = 0; i < n; i++)
    {
        // sum of elements so far
        sum += arr[i];

        // if sum is seen for first time, insert sum with its index
        // into the map
        if (map.find(sum) == map.end())
            map[sum] = i;

        // update length and ending index of maximum length sub-array
        // having sum S
        if (map.find(sum - S) != map.end() && len < i - map[sum - S])
        {
            len =  i - map[sum - S];
            ending_index = i;
        }
    }

    // print the sub-array
    cout << "[" << (ending_index - len + 1) << "," << ending_index << "]";
}

int main()
{
    int arr[] = { 5, 6, -5, 5, 3, 5, 3, -2, 0 };
    int sum = 8;
```

```
    int n = sizeof(arr) / sizeof(arr[0]);
    maxLengthSubArray(arr, n, sum);

    return 0;
}
```

java语言实现：

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

class MaxLengthSubArray
{
    // Naive function to find maximum length sub-array with sum S present
    // in the given array
    public static void maxLengthSubArray(int arr[], int S)
    {
        // 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 S starts from index 0
        map.put(0, -1);

        int sum = 0;

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

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

        // traverse the given array
        for (int i = 0; i < n; i++)
        {
            // sum of elements so far
```

```
            sum += arr[i];

            // if sum is seen for first time, insert sum with its index
            // into the map
            if (!map.containsKey(sum))
                map.put(sum, i);

            // update length and ending index of maximum length sub-array
            // having sum S
            if (map.containsKey(sum - S) && len < i - map.get(sum - S))
            {
                len =  i - map.get(sum - S);
                ending_index = i;
            }
        }

        // print the sub-array
        System.out.println("[" + (ending_index - len + 1) + ", "
                           + ending_index + "]");
    }

    // main function
    public static void main (String[] args)
    {
        int arr[] = { 5, 6, -5, 5, 3, 5, 3, -2, 0 };
        int sum = 8;

        maxLengthSubArray(arr, sum);
    }
}
```

输出: [2,5]

以上方案的时间复杂度和空间复杂度都是o(n)。