# LeetCode\_Day\_28 Subarray Sum

It is quite common that you need to calculate the sum of a subarray. Generally you need to scan from the start position to the end position and sum up every element in the array, but what is you need to calculate the sum of the subarray frequently? Do you need to scan every time? Or you can build some mechanism which just scan the whole array once and then calculate the sum of subarray in any range.

The solution is that you record the accumulated sum at every index, say sum[i] is the total sum until the index i, in this case if you need to calculate the sum between i and j, inclusive, the answer should be sum[j] - sum[i-1].

Such pattern appears frequently in array and DP problem.

## 134. Gas Station

Medium

There are *N* gas stations along a circular route, where the amount of gas at station *i* is gas[i].

You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from station *i* to its next station (*i*+1). You begin the journey with an empty tank at one of the gas stations.

Return the starting gas station's index if you can travel around the circuit once in the clockwise direction, otherwise return -1.

**Note:**

* If there exists a solution, it is guaranteed to be unique.
* Both input arrays are non-empty and have the same length.
* Each element in the input arrays is a non-negative integer.

**Example 1:**

**Input:**

gas = [1,2,3,4,5]

cost = [3,4,5,1,2]

**Output:** 3

**Explanation:**

Start at station 3 (index 3) and fill up with 4 unit of gas. Your tank = 0 + 4 = 4

Travel to station 4. Your tank = 4 - 1 + 5 = 8

Travel to station 0. Your tank = 8 - 2 + 1 = 7

Travel to station 1. Your tank = 7 - 3 + 2 = 6

Travel to station 2. Your tank = 6 - 4 + 3 = 5

Travel to station 3. The cost is 5. Your gas is just enough to travel back to station 3.

Therefore, return 3 as the starting index.

**Example 2:**

**Input:**

gas = [2,3,4]

cost = [3,4,3]

**Output:** -1

**Explanation:**

You can't start at station 0 or 1, as there is not enough gas to travel to the next station.

Let's start at station 2 and fill up with 4 unit of gas. Your tank = 0 + 4 = 4

Travel to station 0. Your tank = 4 - 3 + 2 = 3

Travel to station 1. Your tank = 3 - 3 + 3 = 3

You cannot travel back to station 2, as it requires 4 unit of gas but you only have 3.

Therefore, you can't travel around the circuit once no matter where you start.

### Analysis:

On every station if you deduct the cost from gas you will get a gain of loss at that station. Starting from the station next to the biggest loss with 0 gas and travel in a loop, if on station if you get another negative gas then you cannot complete trip.

/// <summary>

/// Leet code #134. Gas Station

///

/// There are N gas stations along a circular route, where the amount of gas

/// at station i is gas[i].

///

/// You have a car with an unlimited gas tank and it costs cost[i] of gas to

/// travel from station i to its next station (i+1). You begin the journey

/// with an empty tank at one of the gas stations.

///

/// Return the starting gas station's index if you can travel around the

/// circuit once in the clockwise direction, otherwise return -1.

///

/// Note:

///

/// If there exists a solution, it is guaranteed to be unique.

/// Both input arrays are non-empty and have the same length.

/// Each element in the input arrays is a non-negative integer.

///

/// Example 1:

/// Input:

/// gas  = [1,2,3,4,5]

/// cost = [3,4,5,1,2]

/// Output: 3

///

/// Explanation:

/// 1. Start at station 3 (index 3) and fill up with 4 unit of gas.

///    Your tank = 0 + 4 = 4

/// 2. Travel to station 4. Your tank = 4 - 1 + 5 = 8

/// 3. Travel to station 0. Your tank = 8 - 2 + 1 = 7

/// 4. Travel to station 1. Your tank = 7 - 3 + 2 = 6

/// 5. Travel to station 2. Your tank = 6 - 4 + 3 = 5

/// 6. Travel to station 3. The cost is 5. Your gas is just enough to travel

///    back to station 3.

/// 7. Therefore, return 3 as the starting index.

///

/// Example 2:

/// Input:

/// gas  = [2,3,4]

/// cost = [3,4,3]

/// Output: -1

///

/// Explanation:

/// 1. You can't start at station 0 or 1, as there is not enough gas to travel

///    to the next station.

/// 2. Let's start at station 2 and fill up with 4 unit of gas. Your

///    tank = 0 + 4 = 4

/// 3. Travel to station 0. Your tank = 4 - 3 + 2 = 3

/// 4. Travel to station 1. Your tank = 3 - 3 + 3 = 3

/// 5. You cannot travel back to station 2, as it requires 4 unit of gas but

///    you only have 3.

/// 6. Therefore, you can't travel around the circuit once no matter where

///    you start.

/// </summary>

int LeetCodeArray::canCompleteCircuit(vector<int>& gas, vector<int>& cost)

{

    vector<int> sum(gas.size());

    int start\_index = -1;

    int min\_sum = INT\_MAX;

    for (size\_t i = 0; i < gas.size(); i++)

    {

        if (i == 0)

        {

            sum[i] = gas[i] - cost[i];

        }

        else

        {

            sum[i] = sum[i - 1] + gas[i] - cost[i];

        }

        if (sum[i] < min\_sum)

        {

            min\_sum = sum[i];

            start\_index = (i + 1 == gas.size()) ? 0 : i + 1;

        }

    }

    if (sum[gas.size() - 1] >= 0)

    {

        return start\_index;

    }

    else

    {

        return -1;

    }

}

## 152. Maximum Product Subarray

Medium

Given an integer array nums, find the contiguous subarray within an array (containing at least one number) which has the largest product.

**Example 1:**

**Input:** [2,3,-2,4]

**Output:** 6

**Explanation:** [2,3] has the largest product 6.

**Example 2:**

**Input:** [-2,0,-1]

**Output:** 0

**Explanation:** The result cannot be 2, because [-2,-1] is not a subarray.

### Analysis:

Because that we have the negative numbers and 0, the maximum product must come from the previous minim product or the maximum product times the current value, and then compare the current value to get the current minimum or maximum product.

Please note that you can not use divide because there is a zero in the array, which will cause extra code.

/// <summary>

/// Leet code #152. Maximum Product Subarray

///

/// Given an integer array nums, find the contiguous subarray within an array

/// (containing at least one number) which has the largest product.

///

/// Example 1:

///

/// Input: [2,3,-2,4]

/// Output: 6

/// Explanation: [2,3] has the largest product 6.

///

/// Example 2:

///

/// Input: [-2,0,-1]

/// Output: 0

/// Explanation: The result cannot be 2, because [-2,-1] is not a subarray.

/// </summary>

int LeetCodeArray::maxProduct(vector<int>& nums)

{

int result = INT\_MIN;

int min\_product = 1;

int max\_product = 1;

for (size\_t i = 0; i < nums.size(); i++)

{

int curr\_min = min(min(nums[i] \* min\_product, nums[i] \* max\_product), nums[i]);

int curr\_max = max(max(nums[i] \* min\_product, nums[i] \* max\_product), nums[i]);

min\_product = curr\_min;

max\_product = curr\_max;

result = max(result, max\_product);

}

return result;

}

## 238. Product of Array Except Self

Medium

Given an array nums of *n* integers where *n* > 1,  return an array output such that output[i] is equal to the product of all the elements of nums except nums[i].

**Example:**

**Input:** [1,2,3,4]

**Output:** [24,12,8,6]

**Constraint:** It's guaranteed that the product of the elements of any prefix or suffix of the array (including the whole array) fits in a 32 bit integer.

**Note:**Please solve it **without division** and in O(*n*).

**Follow up:**  
Could you solve it with constant space complexity? (The output array **does not** count as extra space for the purpose of space complexity analysis.)

### Analysis:

For any product which is except itself, which means the product on left times the product on right, so you first calculate the left product put it in result and then do the right product.

/// <summary>

/// Leet code #238. Product of Array Except Self

/// Given an array of n integers where n > 1, nums, return an array output such

/// that output[i] is equal to the product of all the elements of nums except

/// nums[i].

///

/// Solve it without division and in O(n).

/// For example, given [1,2,3,4], return [24,12,8,6].

///

/// Follow up:

/// Could you solve it with constant space complexity?

/// (Note: The output array does not count as extra space for the purpose of

/// space complexity analysis.)

/// </summary>

vector<int> LeetCodeArray::productExceptSelf(vector<int>& nums)

{

    vector<int> result(nums.size());

    int product = 1;

    for (size\_t i = 0; i < nums.size(); i++)

    {

        result[i] = product;

        product = product \* nums[i];

    }

    product = 1;

    for (int i = nums.size() - 1; i >= 0; i--)

    {

        result[i] = result[i] \* product;

        product \*= nums[i];

    }

    return result;

}

## 1124. Longest Well-Performing Interval

Medium

We are given hours, a list of the number of hours worked per day for a given employee.

A day is considered to be a *tiring day* if and only if the number of hours worked is (strictly) greater than 8.

A *well-performing interval* is an interval of days for which the number of tiring days is strictly larger than the number of non-tiring days.

Return the length of the longest well-performing interval.

**Example 1:**

**Input:** hours = [9,9,6,0,6,6,9]

**Output:** 3

**Explanation:** The longest well-performing interval is [9,9,6].

**Constraints:**

* 1 <= hours.length <= 10000
* 0 <= hours[i] <= 16

### Analysis:

Accumulate the tiring days, remember every negative accumulated number at its first occurrence, for any positive sum, we count whole left subarray, for any negative value check distance between sum and sum - 1.

/// <summary>

/// Leet code #1124. Longest Well-Performing Interval

///

/// We are given hours, a list of the number of hours worked per day for a

/// given employee.

/// A day is considered to be a tiring day if and only if the number of hours

/// worked is (strictly) greater than 8.

/// A well-performing interval is an interval of days for which the number of

/// tiring days is strictly larger than the number of non-tiring days.

/// Return the length of the longest well-performing interval.

///

/// Example 1:

/// Input: hours = [9,9,6,0,6,6,9]

/// Output: 3

/// Explanation: The longest well-performing interval is [9,9,6].

///

/// Constraints:

/// 1. 1 <= hours.length <= 10000

/// 2. 0 <= hours[i] <= 16

/// </summary>

int LeetCode::longestWPI(vector<int>& hours)

{

int n = hours.size();

vector<int> dp(n, -1);

int sum = 0;

int result = 0;

for (int i = 0; i < (int)hours.size(); i++)

{

if (hours[i] <= 8)

{

sum--;

}

else

{

sum++;

}

if (sum > 0) result = i + 1;

else if (sum == -n) continue;

else

{

// for zero we do not need to store the position

if (sum < 0) if (dp[sum + n] == -1) dp[sum + n] = i;

if (dp[sum + n - 1] != -1)

{

result = max(result, i - dp[sum + n - 1]);

}

}

}

return result;

}

## 1191. K-Concatenation Maximum Sum

Medium

Given an integer array arr and an integer k, modify the array by repeating it k times.

For example, if arr = [1, 2] and k = 3 then the modified array will be [1, 2, 1, 2, 1, 2].

Return the maximum sub-array sum in the modified array. Note that the length of the sub-array can be 0 and its sum in that case is 0.

As the answer can be very large, return the answer **modulo** 10^9 + 7.

**Example 1:**

**Input:** arr = [1,2], k = 3

**Output:** 9

**Example 2:**

**Input:** arr = [1,-2,1], k = 5

**Output:** 2

**Example 3:**

**Input:** arr = [-1,-2], k = 7

**Output:** 0

**Constraints:**

* 1 <= arr.length <= 10^5
* 1 <= k <= 10^5
* -10^4 <= arr[i] <= 10^4

### Analysis:

We iterate array at most twice and calculate maximum subarray sum, and if the sum of array is positive then add (k-2) \* sum of array.

/// <summary>

/// Leet code #1191. K-Concatenation Maximum Sum

///

/// Given an integer array arr and an integer k, modify the array by repeating

/// it k times.

/// For example, if arr = [1, 2] and k = 3 then the modified array will

/// be [1, 2, 1, 2, 1, 2].

/// Return the maximum sub-array sum in the modified array. Note that the

/// length of the sub-array can be 0 and its sum in that case is 0.

///

/// As the answer can be very large, return the answer modulo 10^9 + 7.

///

/// Example 1:

/// Input: arr = [1,2], k = 3

/// Output: 9

///

/// Example 2:

/// Input: arr = [1,-2,1], k = 5

/// Output: 2

/// Example 3:

/// Input: arr = [-1,-2], k = 7

/// Output: 0

///

/// Constraints:

/// 1. 1 <= arr.length <= 10^5

/// 2. 1 <= k <= 10^5

/// 3. -10^4 <= arr[i] <= 10^4

/// </summary>

/// <summary>

/// Leet code #1191. K-Concatenation Maximum Sum

///

/// Given an integer array arr and an integer k, modify the array by repeating

/// it k times.

/// For example, if arr = [1, 2] and k = 3 then the modified array will

/// be [1, 2, 1, 2, 1, 2].

/// Return the maximum sub-array sum in the modified array. Note that the

/// length of the sub-array can be 0 and its sum in that case is 0.

///

/// As the answer can be very large, return the answer modulo 10^9 + 7.

///

/// Example 1:

/// Input: arr = [1,2], k = 3

/// Output: 9

///

/// Example 2:

/// Input: arr = [1,-2,1], k = 5

/// Output: 2

/// Example 3:

/// Input: arr = [-1,-2], k = 7

/// Output: 0

///

/// Constraints:

/// 1. 1 <= arr.length <= 10^5

/// 2. 1 <= k <= 10^5

/// 3. -10^4 <= arr[i] <= 10^4

/// </summary>

int LeetCodeArray::kConcatenationMaxSum(vector<int>& arr, int k)

{

int M = 1000000007;

// use 64 bits so we do not worry about overflow

long long sum = 0;

long long min\_sum = 0;

long long result = 0;

int n = arr.size();

for (size\_t i = 0; i < min(k, 2) \* arr.size(); i++)

{

sum += arr[i % n];

result = max(result, sum - min\_sum);

min\_sum = min(min\_sum, sum);

}

if (sum > 0 && k > 2)

{

result += (sum / 2) \* (k - 2);

}

result = result % M;

return (int)result;

}

## 1477. Find Two Non-overlapping Sub-arrays Each With Target Sum

Medium

Given an array of integers arr and an integer target.

You have to find **two non-overlapping sub-arrays** of arr each with sum equal target. There can be multiple answers so you have to find an answer where the sum of the lengths of the two sub-arrays is **minimum**.

Return *the minimum sum of the lengths* of the two required sub-arrays, or return ***-1*** if you cannot find such two sub-arrays.

**Example 1:**

**Input:** arr = [3,2,2,4,3], target = 3

**Output:** 2

**Explanation:** Only two sub-arrays have sum = 3 ([3] and [3]). The sum of their lengths is 2.

**Example 2:**

**Input:** arr = [7,3,4,7], target = 7

**Output:** 2

**Explanation:** Although we have three non-overlapping sub-arrays of sum = 7 ([7], [3,4] and [7]), but we will choose the first and third sub-arrays as the sum of their lengths is 2.

**Example 3:**

**Input:** arr = [4,3,2,6,2,3,4], target = 6

**Output:** -1

**Explanation:** We have only one sub-array of sum = 6.

**Example 4:**

**Input:** arr = [5,5,4,4,5], target = 3

**Output:** -1

**Explanation:** We cannot find a sub-array of sum = 3.

**Example 5:**

**Input:** arr = [3,1,1,1,5,1,2,1], target = 3

**Output:** 3

**Explanation:** Note that sub-arrays [1,2] and [2,1] cannot be an answer because they overlap.

**Constraints:**

* 1 <= arr.length <= 10^5
* 1 <= arr[i] <= 1000
* 1 <= target <= 10^8

### Analysis:

We can use prefix to calculate prefix sum, for any two subarrays with no overlap, we can track at any position, what is the least subarray until that position.

/// <summary>

/// Leet code #1477. Find Two Non-overlapping Sub-arrays Each With

///                  Target Sum

///

/// Medium

///

/// Given an array of integers arr and an integer target.

///

/// You have to find two non-overlapping sub-arrays of arr each with

/// sum equal target. There can be multiple answers so you have to

/// find an answer where the sum of the lengths of the two sub-arrays

/// is minimum.

///

/// Return the minimum sum of the lengths of the two required

/// sub-arrays, or return -1 if you cannot find such two sub-arrays.

///

/// Example 1:

/// Input: arr = [3,2,2,4,3], target = 3

/// Output: 2

/// Explanation: Only two sub-arrays have sum = 3 ([3] and [3]). The

/// sum of their lengths is 2.

///

/// Example 2:

/// Input: arr = [7,3,4,7], target = 7

/// Output: 2

/// Explanation: Although we have three non-overlapping sub-arrays

/// of sum = 7 ([7], [3,4] and [7]), but we will choose the first

/// and third sub-arrays as the sum of their lengths is 2.

///

/// Example 3:

/// Input: arr = [4,3,2,6,2,3,4], target = 6

/// Output: -1

/// Explanation: We have only one sub-array of sum = 6.

///

/// Example 4:

/// Input: arr = [5,5,4,4,5], target = 3

/// Output: -1

/// Explanation: We cannot find a sub-array of sum = 3.

///

/// Example 5:

/// Input: arr = [3,1,1,1,5,1,2,1], target = 3

/// Output: 3

/// Explanation: Note that sub-arrays [1,2] and [2,1] cannot be an

/// answer because they overlap.

///

/// Constraints:

/// 1. 1 <= arr.length <= 10^5

/// 2. 1 <= arr[i] <= 1000

/// 3. 1 <= target <= 10^8

/// </summary>

int LeetCodeArray::minSumOfLengths(vector<int>& arr, int target)

{

    unordered\_map<int, int> sum\_map;

    sum\_map[0] = -1;

    int sum = 0;

    vector<int> dp(arr.size(), INT\_MAX);

    int result = INT\_MAX;

    for (int i = 0; i < (int)arr.size(); i++)

    {

        sum += arr[i];

        if (sum\_map.count(sum - target) > 0)

        {

            int gap = i - sum\_map[sum - target];

            int pos = i - gap;

            if (pos >= 0 && dp[pos] !=  INT\_MAX)

            {

                result = min(result, gap + dp[pos]);

            }

            dp[i] = gap;

        }

        if (i > 0) dp[i] = min(dp[i], dp[i - 1]);

        sum\_map[sum] = i;

    }

    return result == INT\_MAX ? -1 : result;

}