LeetCode Training Day 3 Prefix Sum

When we are asked to calculate sum in a subarray in an array, we will use the trick of prefix sum. If we do brute-force we have to use two loops, which get all combination of subarray then do the calculation. This will give us a O(N\*N) time complexity which is not optimal. What we do is we calculate the sum dp[i] as the sum from 0 to (i), so if we asked to get the sub array sum from s to t, we simply do dp[t] – dp[s].

If we want to calculate the maximum subarray sum (assume the array contains negative), we just need to capture the minimum prefix sum seen before the current position. If we want to capture the subarray sum with multiple of K, we just need to record all the value of arr[i] % k.

If the array become circular, we can do it with 2 \* length.

## 53. Maximum Subarray

Easy

Given an integer array nums, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.

**Example:**

**Input:** [-2,1,-3,4,-1,2,1,-5,4],

**Output:** 6

**Explanation:** [4,-1,2,1] has the largest sum = 6.

**Follow up:**

If you have figured out the O(*n*) solution, try coding another solution using the divide and conquer approach, which is more subtle.

### Analysis:

we calculate the accumulated sum at every position and also record the minimum sum previously appeared, so the max difference between the current accumulated sum and the previous minimum sum is the maximum sum of the subarray.

/// <summary>

/// Leet code #53. Maximum Subarray

///

/// Given an integer array nums, find the contiguous subarray (containing

/// at least one number) which has the largest sum and return its sum.

///

/// Example:

///

/// Input: [-2,1,-3,4,-1,2,1,-5,4],

/// Output: 6

/// Explanation: [4,-1,2,1] has the largest sum = 6.

///

/// Follow up:

/// If you have figured out the O(n) solution, try coding another solution

/// using the divide and conquer approach, which is more subtle.

/// </summary>

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

{

int min\_sum = 0;

int max\_sum = INT\_MIN;

int sum = 0;

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

{

sum += nums[i];

// calculate max\_sum first because min\_sum is for previous ones.

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

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

}

return max\_sum;

}

## 918. Maximum Sum Circular Subarray

Medium

Given a **circular array** **C** of integers represented by A, find the maximum possible sum of a non-empty subarray of **C**.

Here, a *circular array* means the end of the array connects to the beginning of the array.  (Formally, C[i] = A[i] when 0 <= i < A.length, and C[i+A.length] = C[i] when i >= 0.)

Also, a subarray may only include each element of the fixed buffer A at most once.  (Formally, for a subarray C[i], C[i+1], ..., C[j], there does not exist i <= k1, k2 <= j with k1 % A.length = k2 % A.length.)

**Example 1:**

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

**Output:** 3

**Explanation:** Subarray [3] has maximum sum 3

**Example 2:**

**Input:** [5,-3,5]

**Output:** 10

**Explanation:** Subarray [5,5] has maximum sum 5 + 5 = 10

**Example 3:**

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

**Output:** 4

**Explanation:** Subarray [2,-1,3] has maximum sum 2 + (-1) + 3 = 4

**Example 4:**

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

**Output:** 3

**Explanation:** Subarray [3] and [3,-2,2] both have maximum sum 3

**Example 5:**

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

**Output:** -1

**Explanation:** Subarray [-1] has maximum sum -1

**Note:**

1. -30000 <= A[i] <= 30000
2. 1 <= A.length <= 30000

### Analysis:

We track the maximum left sum or minimum left sum to calculate max\_sum and min\_sum for any subarray. The max\_sum must come from max\_sum, or sum – min\_sum.

/// <summary>

/// Leet code #918. Maximum Sum Circular Subarray

///

/// Given a circular array C of integers represented by A, find the maximum

/// possible sum of a non-empty subarray of C.

///

/// Here, a circular array means the end of the array connects to the

/// beginning of the array.  (Formally, C[i] = A[i] when 0 <= i < A.length,

/// and C[i+A.length] = C[i] when i >= 0.)

///

/// Also, a subarray may only include each element of the fixed buffer A at

/// most once.  (Formally, for a subarray C[i], C[i+1], ..., C[j], there does

/// not exist i <= k1, k2 <= j with k1 % A.length = k2 % A.length.)

///

///

/// Example 1:

/// Input: [1,-2,3,-2]

/// Output: 3

/// Explanation: Subarray [3] has maximum sum 3

///

/// Example 2:

/// Input: [5,-3,5]

/// Output: 10

/// Explanation: Subarray [5,5] has maximum sum 5 + 5 = 10

///

/// Example 3:

/// Input: [3,-1,2,-1]

/// Output: 4

/// Explanation: Subarray [2,-1,3] has maximum sum 2 + (-1) + 3 = 4

///

/// Example 4:

/// Input: [3,-2,2,-3]

/// Output: 3

/// Explanation: Subarray [3] and [3,-2,2] both have maximum sum 3

//

/// Example 5:

/// Input: [-2,-3,-1]

/// Output: -1

/// Explanation: Subarray [-1] has maximum sum -1

///

///

/// Note:

///

/// 1. -30000 <= A[i] <= 30000

/// 2. 1 <= A.length <= 30000

///

/// </summary>

int LeetCodeArray::maxSubarraySumCircular(vector<int>& A)

{

    int sum = 0;

    int min\_left = 0;

    int max\_left = 0;

    int max\_sum = INT\_MIN;

    int min\_sum = INT\_MAX;

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

    {

        sum += A[i];

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

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

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

        max\_left = max(max\_left, sum);

    }

    int result = max\_sum;

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

    return result;

}

## 325. Maximum Size Subarray Sum Equals k

Medium

Given an array *nums* and a target value *k*, find the maximum length of a subarray that sums to *k*. If there isn't one, return 0 instead.

**Note:**  
The sum of the entire *nums* array is guaranteed to fit within the 32-bit signed integer range.

**Example 1:**

**Input:** *nums* = [1, -1, 5, -2, 3], *k* = 3

**Output:** 4

**Explanation:** The subarray [1, -1, 5, -2] sums to 3 and is the longest.

**Example 2:**

**Input:** *nums* = [-2, -1, 2, 1], *k* = 1

**Output:** 2

**Explanation:** The subarray [-1, 2] sums to 1 and is the longest.

**Follow Up:**  
Can you do it in O(*n*) time?

### Analysis:

The sum of subarray can be calculated on the accumulated sum on the current position deduct some previous sum, so if you expect the result is K you will have a previous sum which is sum[i] – K, for any current index as i, you can record the previous sum in the hash table. To get the longest subarray, we only need to record the first accumulate sum as s.

/// <summary>

/// Leet code #325. Maximum Size Subarray Sum Equals k

///

/// Given an array nums and a target value k, find the maximum length of a

/// subarray that sums to k.

/// If there isn't one, return 0 instead.

/// Note:

/// The sum of the entire nums array is guaranteed to fit within the 32-bit

/// signed integer range.

///

/// Example 1:

/// Given nums = [1, -1, 5, -2, 3], k = 3,

/// return 4. (because the subarray [1, -1, 5, -2] sums to 3 and is the longest)

///

/// Example 2:

/// Given nums = [-2, -1, 2, 1], k = 1,

/// return 2. (because the subarray [-1, 2] sums to 1 and is the longest)

///

/// Follow Up:

/// Can you do it in O(n) time?

/// </summary>

int LeetCodeArray::maxSubArrayLen(vector<int>& nums, int k)

{

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

int sum = 0;

int max\_length = 0;

sum\_map[0] = -1;

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

{

sum += nums[i];

if (sum\_map.find(sum - k) != sum\_map.end())

{

max\_length = max(max\_length, i - sum\_map[sum - k]);

}

if (sum\_map.find(sum) == sum\_map.end())

{

sum\_map[sum] = i;

}

}

return max\_length;

}

## 523. Continuous Subarray Sum

Medium

Given a list of **non-negative** numbers and a target **integer** k, write a function to check if the array has a continuous subarray of size at least 2 that sums up to a multiple of **k**, that is, sums up to n\*k where n is also an **integer**.

**Example 1:**

**Input:** [23, 2, 4, 6, 7], k=6

**Output:** True

**Explanation:** Because [2, 4] is a continuous subarray of size 2 and sums up to 6.

**Example 2:**

**Input:** [23, 2, 6, 4, 7], k=6

**Output:** True

**Explanation:** Because [23, 2, 6, 4, 7] is an continuous subarray of size 5 and sums up to 42.

**Note:**

1. The length of the array won't exceed 10,000.
2. You may assume the sum of all the numbers is in the range of a signed 32-bit integer.

### Analysis:

Similar to LC#325, the subarray sum is calculated by current accumulated sum and the previous accumulated sum, but you need to mod the sum to K and see if there is same remainder before.

Just watch K = 0, where % operation is invalid.

/// <summary>

/// Leet code #523. Continuous Subarray Sum

///

/// Given a list of non-negative numbers and a target integer k, write a

/// function to check if the array has a continuous subarray of size at

/// least 2 that sums up to the multiple of k, that is, sums up to n\*k

/// where n is also an integer.

///

/// Example 1:

/// Input: [23, 2, 4, 6, 7], k=6

/// Output: True

/// Explanation: Because [2, 4] is a continuous subarray of size 2 and

/// sums up to 6.

/// Example 2:

/// Input: [23, 2, 6, 4, 7], k=42

/// Output: True

/// Explanation: Because [23, 2, 6, 4, 7] is an continuous subarray of

/// size 5 and sums up to 42.

/// Note:

/// The length of the array won't exceed 10,000.

/// You may assume the sum of all the numbers is in the range of a

/// signed 32-bit integer.

/// </summary>

bool LeetCode::checkSubarraySum(vector<int>& nums, int k)

{

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

int sum = 0;

num\_map[sum] = -1;

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

{

sum += nums[i];

if (k == 0)

{

if ((num\_map.count(-sum) > 0) && (num\_map[-sum] + 1 < (int)i))

{

return true;

}

if (num\_map.count(sum) == 0) num\_map[sum] = i;

}

else

{

if ((num\_map.count(sum % k) > 0) && (num\_map[sum % k] + 1 < (int)i))

{

return true;

}

if (num\_map.count(sum % k) == 0) num\_map[sum % k] = i;

}

}

return false;

}

## 525. Contiguous Array

Medium

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.

### Analysis:

If you consider 0 as -1 and 1 as +1 in the array, you just need to calculate the longest subarray with the sum of zero.

/// <summary>

/// Leet code #525. Contiguous Array

///

/// 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.

/// </summary>

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

{

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

// set the initial equal point before the array

pos\_map[0] = -1;

vector<int> num\_count(nums.size());

int max\_length = 0;

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

{

if (nums[i] == 0)

{

num\_count[i] = -1;

}

else

{

num\_count[i] = 1;

}

if (i > 0) num\_count[i] += num\_count[i - 1];

if (pos\_map.count(num\_count[i]) == 0)

{

pos\_map[num\_count[i]] = i;

}

else

{

max\_length = max(max\_length, (int)(i - pos\_map[num\_count[i]]));

}

}

return max\_length;

}

## 560. Subarray Sum Equals K

Medium

Given an array of integers and an integer **k**, you need to find the total number of continuous subarrays whose sum equals to **k**.

**Example 1:**

**Input:**nums = [1,1,1], k = 2

**Output:** 2

**Note:**

1. The length of the array is in range [1, 20,000].
2. The range of numbers in the array is [-1000, 1000] and the range of the integer **k** is [-1e7, 1e7].

### Analysis:

You need to count all the previous sum in the hash table so you can find the subarray sum as K and calculate the count.

/// <summary>

/// Leet code #560. Subarray Sum Equals K

///

/// Given an array of integers and an integer k, you need to find the

/// total number of continuous subarrays whose sum equals to k.

/// Example 1:

/// Input:nums = [1,1,1], k = 2

/// Output: 2

///

/// Note:

/// The length of the array is in range [1, 20,000].

/// The range of numbers in the array is [-1000, 1000] and the range of

/// the integer k is [-1e7, 1e7].

/// </summary>

int LeetCodeArray::subarraySum(vector<int>& nums, int k)

{

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

int sum = 0;

int count = 0;

sum\_map[0] = 1;

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

{

sum += nums[i];

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

{

count += sum\_map[sum - k];

}

sum\_map[sum]++;

}

return count;

}

## 930. Binary Subarrays With Sum

Medium

In an array A of 0s and 1s, how many **non-empty** subarrays have sum S?

**Example 1:**

**Input:** A = [1,0,1,0,1], S = 2

**Output:** 4

**Explanation:**

The 4 subarrays are bolded below:

[**1,0,1**,0,1]

[**1,0,1,0**,1]

[1,**0,1,0,1**]

[1,0,**1,0,1**]

**Note:**

1. A.length <= 30000
2. 0 <= S <= A.length
3. A[i] is either 0 or 1.

### Analysis:

You need to record all the 1s in the array, may keep only the length of S + 1 if you use deque, count the 0 after 1, and check every position with the previous sum = sum - S.

/// <summary>

/// Leet code #930. Binary Subarrays With Sum

///

/// In an array A of 0s and 1s, how many non-empty subarrays have sum S?

///

/// Example 1:

/// Input: A = [1,0,1,0,1], S = 2

/// Output: 4

/// Explanation:

/// The 4 subarrays are bolded below:

/// [1,0,1,x,x]

/// [1,0,1,0,x]

/// [x,0,1,0,1]

/// [x,x,1,0,1]

///

/// Note:

///

/// 1. A.length <= 30000

/// 2. 0 <= S <= A.length

/// 3. A[i] is either 0 or 1.

/// </summary>

int LeetCodeArray::numSubarraysWithSum(vector<int>& A, int S)

{

int result = 0;

vector<int> dp;

dp.push\_back(0);

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

{

if (A[i] == 1) dp.push\_back(0);

else dp[dp.size() - 1]++;

if (dp.size() < (size\_t)(S + 1)) continue;

if (S == 0)

{

result += dp[dp.size() - 1];

}

else

{

result += dp[dp.size() - 1 - S] + 1;

}

}

return result;

}

## 974. Subarray Sums Divisible by K

Medium

Given an array A of integers, return the number of (contiguous, non-empty) subarrays that have a sum divisible by K.

**Example 1:**

**Input:** A = [4,5,0,-2,-3,1], K = 5

**Output:** 7

**Explanation:** There are 7 subarrays with a sum divisible by K = 5:

[4, 5, 0, -2, -3, 1], [5], [5, 0], [5, 0, -2, -3], [0], [0, -2, -3], [-2, -3]

**Note:**

1. 1 <= A.length <= 30000
2. -10000 <= A[i] <= 10000
3. 2 <= K <= 10000

### Analysis:

Calculate the accumulated sum and mod by K and add the count of previous accumulated sum with same mod value (remainder). Remember and remainder 0 by default has a count of 1.

/// <summary>

/// Leet code #974. Subarray Sums Divisible by K

///

/// Given an array A of integers, return the number of (contiguous, non-empty)

/// subarrays that have a sum divisible by K.

///

/// Example 1:

///

/// Input: A = [4,5,0,-2,-3,1], K = 5

/// Output: 7

/// Explanation: There are 7 subarrays with a sum divisible by K = 5:

/// [4, 5, 0, -2, -3, 1], [5], [5, 0], [5, 0, -2, -3], [0], [0, -2, -3],

/// [-2, -3]

///

/// Note:

///

/// 1. 1 <= A.length <= 30000

/// 2. -10000 <= A[i] <= 10000

/// 3. 2 <= K <= 10000

/// </summary>

int LeetCodeArray::subarraysDivByK(vector<int>& A, int K)

{

vector<int> remainder(K);

remainder[0]++;

int sum = 0;

int result = 0;

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

{

sum += A[i];

int mod = (sum % K + K) % K;

result += remainder[mod];

remainder[mod]++;

}

return result;

}

## 1151. Minimum Swaps to Group All 1's Together

Medium

Given a binary array data, return the minimum number of swaps required to group all 1’s present in the array together in **any place** in the array.

**Example 1:**

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

**Output:** 1

**Explanation:**

There are 3 ways to group all 1's together:

[1,1,1,0,0] using 1 swap.

[0,1,1,1,0] using 2 swaps.

[0,0,1,1,1] using 1 swap.

The minimum is 1.

**Example 2:**

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

**Output:** 0

**Explanation:**

Since there is only one 1 in the array, no swaps needed.

**Example 3:**

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

**Output:** 3

**Explanation:**

One possible solution that uses 3 swaps is [0,0,0,0,0,1,1,1,1,1,1].

**Note:**

1. 1 <= data.length <= 10^5
2. 0 <= data[i] <= 1

### Analysis:

To make all the one together, we need to calculate maximum sum for subarray of K, when we have K of 1s.

/// <summary>

/// Leet code #1151. Minimum Swaps to Group All 1's Together

///

/// Given a binary array data, return the minimum number of swaps required

/// to group all 1’s present in the array together in any place in the array.

///

/// Example 1:

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

/// Output: 1

/// Explanation:

/// There are 3 ways to group all 1's together:

/// [1,1,1,0,0] using 1 swap.

/// [0,1,1,1,0] using 2 swaps.

/// [0,0,1,1,1] using 1 swap.

/// The minimum is 1.

///

/// Example 2:

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

/// Output: 0

/// Explanation:

/// Since there is only one 1 in the array, no swaps needed.

///

/// Example 3:

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

/// Output: 3

/// Explanation:

/// One possible solution that uses 3 swaps is [0,0,0,0,0,1,1,1,1,1,1].

///

/// Note:

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

/// 2. 0 <= data[i] <= 1

/// </summary>

int LeetCodeArray::minSwaps(vector<int>& data)

{

vector<int> sum(data.size());

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

{

if (i == 0) sum[0] = data[0];

else (sum[i] = sum[i - 1] + data[i]);

}

int length = sum[sum.size() - 1];

int result = 0;

for (size\_t i = length - 1; i < sum.size(); i++)

{

int count = 0;

if (i == length - 1) count = sum[i];

else count = sum[i] - sum[i - length];

result = max(result, count);

}

return length - result;

}

## 548. Split Array with Equal Sum

Medium

Given an array with n integers, you need to find if there are triplets (i, j, k) which satisfies following conditions:

1. 0 < i, i + 1 < j, j + 1 < k < n - 1
2. Sum of subarrays (0, i - 1), (i + 1, j - 1), (j + 1, k - 1) and (k + 1, n - 1) should be equal.

where we define that subarray (L, R) represents a slice of the original array starting from the element indexed L to the element indexed R.

**Example:**

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

**Output:** True

**Explanation:**

i = 1, j = 3, k = 5.

sum(0, i - 1) = sum(0, 0) = 1

sum(i + 1, j - 1) = sum(2, 2) = 1

sum(j + 1, k - 1) = sum(4, 4) = 1

sum(k + 1, n - 1) = sum(6, 6) = 1

**Note:**

1. 1 <= n <= 2000.
2. Elements in the given array will be in range [-1,000,000, 1,000,000].

### Analysis:

There is no question that we should use prefix, but if we iterate i, then j then k, we will end up with O(N^3). To make it O(N^2), we should iterate j, then iterate I on the left with all the equal split, then iterate k on the right with equal split.

/// <summary>

/// Leet code #548. Split Array with Equal Sum

///

/// Given an array with n integers, you need to find if there are

/// triplets (i, j, k) which satisfies following conditions:

/// 0 < i, i + 1 < j, j + 1 < k < n - 1

/// Sum of subarrays (0, i - 1), (i + 1, j - 1), (j + 1, k - 1)

/// and (k + 1, n - 1) should be equal.

/// where we define that subarray (L, R) represents a slice of the

/// original array starting from the element indexed L to the element

/// indexed R.

/// Example:

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

/// Output: True

/// Explanation:

/// i = 1, j = 3, k = 5.

/// sum(0, i - 1) = sum(0, 0) = 1

/// sum(i + 1, j - 1) = sum(2, 2) = 1

/// sum(j + 1, k - 1) = sum(4, 4) = 1

/// sum(k + 1, n - 1) = sum(6, 6) = 1

/// Note:

/// 1 <= n <= 2000.

/// Elements in the given array will be in range [-1,000,000, 1,000,000].

/// </summary>

bool LeetCodeArray::splitArray(vector<int>& nums)

{

vector<int> sum(nums.size());

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

{

if (i == 0) sum[i] = nums[i];

else (sum[i] = sum[i - 1] + nums[i]);

}

for (int j = 3; j < (int)nums.size() - 3; j++)

{

unordered\_set<int> sum\_set;

for (int i = 1; i < j - 1; i++)

{

if (sum[i - 1] == sum[j - 1] - sum[i])

{

sum\_set.insert(sum[i - 1]);

}

}

for (int k = j + 2; k < (int)nums.size() - 1; k++)

{

if (sum[k - 1] - sum[j] == sum[sum.size() - 1] - sum[k])

{

if (sum\_set.count(sum[k - 1] - sum[j]) > 0)

{

return true;

}

}

}

}

return false;

}

## 696. Count Binary Substrings

Easy

Give a string s, count the number of non-empty (contiguous) substrings that have the same number of 0's and 1's, and all the 0's and all the 1's in these substrings are grouped consecutively.

Substrings that occur multiple times are counted the number of times they occur.

**Example 1:**

**Input:** "00110011"

**Output:** 6

**Explanation:** There are 6 substrings that have equal number of consecutive 1's and 0's: "0011", "01", "1100", "10", "0011", and "01".

Notice that some of these substrings repeat and are counted the number of times they occur.

Also, "00110011" is not a valid substring because **all** the 0's (and 1's) are not grouped together.

**Example 2:**

**Input:** "10101"

**Output:** 4

**Explanation:** There are 4 substrings: "10", "01", "10", "01" that have equal number of consecutive 1's and 0's.

**Note:**

 s.length will be between 1 and 50,000.

 s will only consist of "0" or "1" characters.

### Analysis:

If we count 0 as -1 1 as +1, we just need to count subarray with sum as 0.

/// <summary>

/// Leet code #696. Count Binary Substrings

///

/// Give a string s, count the number of non-empty (contiguous) substrings

/// that have the same number of 0's and 1's, and all the 0's and all the

/// 1's in these substrings are grouped consecutively.

///

/// Substrings that occur multiple times are counted the number of times

/// they occur.

///

/// Example 1:

/// Input: "00110011"

/// Output: 6

/// Explanation: There are 6 substrings that have equal number of

/// consecutive 1's and 0's: "0011", "01", "1100", "10", "0011", and "01".

///

/// Notice that some of these substrings repeat and are counted the number

/// of times they occur.

///

/// Also, "00110011" is not a valid substring because all the 0's

/// (and 1's) are not grouped together.

/// Example 2:

/// Input: "10101"

/// Output: 4

/// Explanation: There are 4 substrings: "10", "01", "10", "01" that have

/// equal number of consecutive 1's and 0's.

/// Note:

///

/// s.length will be between 1 and 50,000.

/// s will only consist of "0" or "1" characters.

/// </summary>

int LeetCodeArray::countBinarySubstrings(string s)

{

vector<int> count\_array;

int result = 0;

int count = 0;

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

{

if ((i > 0) && (s[i] != s[i - 1]))

{

count\_array.push\_back(count);

count = 1;

}

else

{

count++;

}

if ((!count\_array.empty()) && (count\_array.back() >= count))

{

result++;

}

}

return result;

}

## 1013. Partition Array Into Three Parts With Equal Sum

Easy

Given an array A of integers, return true if and only if we can partition the array into three **non-empty** parts with equal sums.

Formally, we can partition the array if we can find indexes i+1 < j with (A[0] + A[1] + ... + A[i] == A[i+1] + A[i+2] + ... + A[j-1] == A[j] + A[j-1] + ... + A[A.length - 1])

**Example 1:**

**Input:** A = [0,2,1,-6,6,-7,9,1,2,0,1]

**Output:** true

**Explanation:** 0 + 2 + 1 = -6 + 6 - 7 + 9 + 1 = 2 + 0 + 1

**Example 2:**

**Input:** A = [0,2,1,-6,6,7,9,-1,2,0,1]

**Output:** false

**Example 3:**

**Input:** A = [3,3,6,5,-2,2,5,1,-9,4]

**Output:** true

**Explanation:** 3 + 3 = 6 = 5 - 2 + 2 + 5 + 1 - 9 + 4

**Constraints:**

* 3 <= A.length <= 50000
* -10^4 <= A[i] <= 10^4

### Analysis:

We calculate sum of array, then track the 1/3 and 2/3 of sum. These two positions must not be the last element. It also should not be overlapped. We can do it in state transition.

/// <summary>

/// Leet code #1013. Partition Array Into Three Parts With Equal Sum

///

/// Given an array A of integers, return true if and only if we can

/// partition the array into three non-empty parts with equal sums.

///

/// Formally, we can partition the array if we can find indexes i+1 < j with

/// (A[0] + A[1] + ... + A[i] == A[i+1] + A[i+2] + ... + A[j-1] == A[j] +

/// A[j-1] + ... + A[A.length - 1])

///

///

/// Example 1:

///

/// Input: [0,2,1,-6,6,-7,9,1,2,0,1]

/// Output: true

/// Explanation: 0 + 2 + 1 = -6 + 6 - 7 + 9 + 1 = 2 + 0 + 1

/// Example 2:

///

/// Input: [0,2,1,-6,6,7,9,-1,2,0,1]

/// Output: false

/// Example 3:

///

/// Input: [3,3,6,5,-2,2,5,1,-9,4]

/// Output: true

/// Explanation: 3 + 3 = 6 = 5 - 2 + 2 + 5 + 1 - 9 + 4

///

/// Note:

///

/// 1. 3 <= A.length <= 50000

/// 2. -10000 <= A[i] <= 10000

/// </summary>

bool LeetCodeArray::canThreePartsEqualSum(vector<int>& A)

{

    int sum = 0;

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

    {

        sum += A[i];

    }

    if (sum % 3 != 0) return false;

    int first = sum / 3;

    sum = 0;

    int result = 0;

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

    {

        sum += A[i];

        if ((sum == first) && (result == 0))

        {

            result++;

        }

        else if ((sum == first \* 2) && (result == 1) && (i != A.size() - 1))

        {

            result++;

            break;

        }

    }

    if (result == 2) return true;

    else return false;

}

## 1052. Grumpy Bookstore Owner

Medium

Today, the bookstore owner has a store open for customers.length minutes.  Every minute, some number of customers (customers[i]) enter the store, and all those customers leave after the end of that minute.

On some minutes, the bookstore owner is grumpy.  If the bookstore owner is grumpy on the i-th minute, grumpy[i] = 1, otherwise grumpy[i] = 0.  When the bookstore owner is grumpy, the customers of that minute are not satisfied, otherwise they are satisfied.

The bookstore owner knows a secret technique to keep themselves not grumpy for X minutes straight, but can only use it once.

Return the maximum number of customers that can be satisfied throughout the day.

**Example 1:**

**Input:** customers = [1,0,1,2,1,1,7,5], grumpy = [0,1,0,1,0,1,0,1], X = 3

**Output:** 16

**Explanation:** The bookstore owner keeps themselves not grumpy for the last 3 minutes.

The maximum number of customers that can be satisfied = 1 + 1 + 1 + 1 + 7 + 5 = 16.

**Note:**

* 1 <= X <= customers.length == grumpy.length <= 20000
* 0 <= customers[i] <= 1000
* 0 <= grumpy[i] <= 1

### Analysis:

We need to calculate every X minutes, with owner become grumpy how many customers are not satisfied, then apply the secret X minutes for the maximum unsatisfied customer, and the calculate the final result.

/// <summary>

/// Leet code #1052. Grumpy Bookstore Owner

///

/// Today, the bookstore owner has a store open for customers.length minutes.

/// Every minute, some number of customers (customers[i]) enter the store,

/// and all those customers leave after the end of that minute.

///

/// On some minutes, the bookstore owner is grumpy.  If the bookstore owner

/// is grumpy on the i-th minute, grumpy[i] = 1, otherwise grumpy[i] = 0.

/// When the bookstore owner is grumpy, the customers of that minute are not

/// satisfied, otherwise they are satisfied.

///

/// The bookstore owner knows a secret technique to keep themselves not grumpy

/// for X minutes straight, but can only use it once.

///

/// Return the maximum number of customers that can be satisfied throughout

/// the day.

///

/// Example 1:

///

/// Input: customers = [1,0,1,2,1,1,7,5], grumpy = [0,1,0,1,0,1,0,1], X = 3

/// Output: 16

/// Explanation: The bookstore owner keeps themselves not grumpy for the last

/// 3 minutes.

/// The maximum number of customers that can be

/// satisfied = 1 + 1 + 1 + 1 + 7 + 5 = 16.

///

/// Note:

///

/// 1. 1 <= X <= customers.length == grumpy.length <= 20000

/// 2. 0 <= customers[i] <= 1000

/// 3. 0 <= grumpy[i] <= 1

/// </summary>

int LeetCode::maxSatisfied(vector<int>& customers, vector<int>& grumpy, int X)

{

    int result = 0, grumpy\_sum = 0, grumpy\_max = 0;

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

    {

        if (grumpy[i] == 0) result += customers[i];

        else

        {

            grumpy\_sum += customers[i];

        }

        if (i >= (size\_t)X)

        {

            if (grumpy[i - X] == 1)

            {

                grumpy\_sum -= customers[i - X];

            }

        }

        grumpy\_max = max(grumpy\_max, grumpy\_sum);

    }

    result += grumpy\_max;

    return result;

}

/// <summary>

/// Leet code #1052. Grumpy Bookstore Owner

///

/// Today, the bookstore owner has a store open for customers.length minutes.

/// Every minute, some number of customers (customers[i]) enter the store,

/// and all those customers leave after the end of that minute.

///

/// On some minutes, the bookstore owner is grumpy.  If the bookstore owner

/// is grumpy on the i-th minute, grumpy[i] = 1, otherwise grumpy[i] = 0.

/// When the bookstore owner is grumpy, the customers of that minute are not

/// satisfied, otherwise they are satisfied.

///

/// The bookstore owner knows a secret technique to keep themselves not grumpy

/// for X minutes straight, but can only use it once.

///

/// Return the maximum number of customers that can be satisfied throughout

/// the day.

///

/// Example 1:

///

/// Input: customers = [1,0,1,2,1,1,7,5], grumpy = [0,1,0,1,0,1,0,1], X = 3

/// Output: 16

/// Explanation: The bookstore owner keeps themselves not grumpy for the last

/// 3 minutes.

/// The maximum number of customers that can be

/// satisfied = 1 + 1 + 1 + 1 + 7 + 5 = 16.

///

/// Note:

///

/// 1. 1 <= X <= customers.length == grumpy.length <= 20000

/// 2. 0 <= customers[i] <= 1000

/// 3. 0 <= grumpy[i] <= 1

/// </summary>

int LeetCodeArray::maxSatisfied(vector<int>& customers, vector<int>& grumpy, int X)

{

    int result = 0, grumpy\_sum = 0, grumpy\_max = 0;

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

    {

        if (grumpy[i] == 0) result += customers[i];

        else

        {

            grumpy\_sum += customers[i];

        }

        if (i >= (size\_t)X)

        {

            if (grumpy[i - X] == 1)

            {

                grumpy\_sum -= customers[i - X];

            }

        }

        grumpy\_max = max(grumpy\_max, grumpy\_sum);

    }

    result += grumpy\_max;

    return result;

}

## 1004. Max Consecutive Ones III

Medium

Given an array A of 0s and 1s, we may change up to K values from 0 to 1.

Return the length of the longest (contiguous) subarray that contains only 1s.

**Example 1:**

**Input:** A = [1,1,1,0,0,0,1,1,1,1,0], K = 2

**Output:** 6

**Explanation:**

[1,1,1,0,0,**1**,1,1,1,1,**1**]

Bolded numbers were flipped from 0 to 1. The longest subarray is underlined.

**Example 2:**

**Input:** A = [0,0,1,1,0,0,1,1,1,0,1,1,0,0,0,1,1,1,1], K = 3

**Output:** 10

**Explanation:**

[0,0,1,1,**1**,**1**,1,1,1,**1**,1,1,0,0,0,1,1,1,1]

Bolded numbers were flipped from 0 to 1. The longest subarray is underlined.

**Note:**

1. 1 <= A.length <= 20000
2. 0 <= K <= A.length
3. A[i] is 0 or 1

### Analysis:

We can use two pointers to track the subarray to keep the maximum K ‘0’ in the range. Before the number of ‘0’ goes beyond the scope, we can greedily expand the last pointer.

/// <summary>

/// Leet code #1004. Max Consecutive Ones III

///

/// Given an array A of 0s and 1s, we may change up to K values from 0 to 1.

///

/// Return the length of the longest (contiguous) subarray that contains

/// only 1s.

///

/// Example 1:

/// Input: A = [1,1,1,0,0,0,1,1,1,1,0], K = 2

/// Output: 6

/// Explanation:

/// [1,1,1,0,0,1,1,1,1,1,1]

/// Bolded numbers were flipped from 0 to 1.  The longest subarray is

/// underlined.

///

/// Example 2:

/// Input: A = [0,0,1,1,0,0,1,1,1,0,1,1,0,0,0,1,1,1,1], K = 3

/// Output: 10

/// Explanation:

/// [0,0,1,1,1,1,1,1,1,1,1,1,0,0,0,1,1,1,1]

/// Bolded numbers were flipped from 0 to 1.  The longest subarray is

/// underlined.

///

/// Note:

///

/// 1. 1 <= A.length <= 20000

/// 2. 0 <= K <= A.length

/// 3. A[i] is 0 or 1

/// </summary>

int LeetCodeArray::longestOnes(vector<int>& A, int K)

{

    int result = 0;

    int first = -1;

    deque<int> zeros;

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

    {

        if (A[i] == 1 || (int)zeros.size() < K)

        {

            result = max(result, i - first);

        }

        if (A[i] == 0)

        {

            zeros.push\_back(i);

            if ((int)zeros.size() > K)

            {

                first = zeros.front();

                zeros.pop\_front();

            }

        }

    }

    return result;

}

# Advance Problems

## 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;

    }

}

## 1186. Maximum Subarray Sum with One Deletion

Medium

Given an array of integers, return the maximum sum for a **non-empty** subarray (contiguous elements) with at most one element deletion. In other words, you want to choose a subarray and optionally delete one element from it so that there is still at least one element left and the sum of the remaining elements is maximum possible.

Note that the subarray needs to be **non-empty** after deleting one element.

**Example 1:**

**Input:** arr = [1,-2,0,3]

**Output:** 4

**Explanation:** Because we can choose [1, -2, 0, 3] and drop -2, thus the subarray [1, 0, 3] becomes the maximum value.

**Example 2:**

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

**Output:** 3

**Explanation:** We just choose [3] and it's the maximum sum.

**Example 3:**

**Input:** arr = [-1,-1,-1,-1]

**Output:** -1

**Explanation:** The final subarray needs to be non-empty. You can't choose [-1] and delete -1 from it, then get an empty subarray to make the sum equals to 0.

**Constraints:**

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

### Analysis:

Consider delete one element, we can simply add the maximum subarray ending one position before current position and maximum subarray starting one position after current position.

/// <summary>

/// Leet code #1186. Maximum Subarray Sum with One Deletion

///

/// Given an array of integers, return the maximum sum for a non-empty

/// subarray (contiguous elements) with at most one element deletion.

/// In other words, you want to choose a subarray and optionally delete

/// one element from it so that there is still at least one element left

/// and the sum of the remaining elements is maximum possible.

/// Note that the subarray needs to be non-empty after deleting one element.

///

/// Example 1:

/// Input: arr = [1,-2,0,3]

/// Output: 4

/// Explanation: Because we can choose [1, -2, 0, 3] and drop -2, thus the

/// subarray [1, 0, 3] becomes the maximum value.

///

/// Example 2:

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

/// Output: 3

/// Explanation: We just choose [3] and it's the maximum sum.

///

/// Example 3:

/// Input: arr = [-1,-1,-1,-1]

/// Output: -1

/// Explanation: The final subarray needs to be non-empty. You can't

/// choose [-1] and delete -1 from it, then get an empty subarray to make

/// the sum equals to 0.

///

/// Constraints:

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

/// 2. -10^4 <= arr[i] <= 10^4

/// </summary>

int LeetCodeArray::maximumSum(vector<int>& arr)

{

vector<int> dp1(arr.size()), dp2(arr.size());

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

{

dp1[i] = arr[i];

if (i > 0) dp1[i] = max(dp1[i-1] + arr[i], dp1[i]);

}

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

{

dp2[i] = arr[i];

if (i < arr.size() - 1) dp2[i] = max(dp2[i + 1] + arr[i], dp2[i]);

}

int result = INT\_MIN;

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

{

result = max(result, dp1[i]);

if (i > 0 && i < arr.size() - 1)

{

result = max(result, dp1[i - 1] + dp2[i + 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;

}

## 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;

}

## 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;

}

## 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;

}