# Subarray Sum 1D

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.

## 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;

}

## 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 #446. Arithmetic Slices II - Subsequence

///

/// A sequence of numbers is called arithmetic if it consists of at

/// least three elements and if the difference between any two

/// consecutive elements is the same.

///

/// For example, these are arithmetic sequences:

/// 1, 3, 5, 7, 9

/// 7, 7, 7, 7

/// 3, -1, -5, -9

///

/// The following sequence is not arithmetic.

/// 1, 1, 2, 5, 7

/// A zero-indexed array A consisting of N numbers is given. A

/// subsequence slice of that array is any sequence of integers

/// (P0, P1, ..., Pk) such that 0 ≤ P0 < P1 < ... < Pk < N.

///

/// A subsequence slice (P0, P1, ..., Pk) of array A is called

/// arithmetic if the sequence A[P0], A[P1], ..., A[Pk-1], A[Pk]

/// is arithmetic. In particular, this means that k ≥ 2.

///

/// The function should return the number of arithmetic subsequence

/// slices in the array A.

///

/// The input contains N integers. Every integer is in the range of -2^31

/// and 2^31-1 and 0 ≤ N ≤ 1000. The output is guaranteed to be less than

/// 2^31-1.

///

/// Example:

/// Input: [2, 4, 6, 8, 10]

/// Output: 7

///

/// Explanation:

/// All arithmetic subsequence slices are:

/// [2,4,6]

/// [4,6,8]

/// [6,8,10]

/// [2,4,6,8]

/// [4,6,8,10]

/// [2,4,6,8,10]

/// [2,6,10]

/// </summary>

int LeetCode::numberOfArithmeticSlicesII(vector<int>& A)

{

int count = 0;

vector<unordered\_map<int, int>> arithmeticCount(A.size());

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

{

for (int j = i - 1; j >= 0; j--)

{

int diff = A[i] - A[j];

arithmeticCount[i][diff] += 1;

if (arithmeticCount[j].count(diff) > 0)

{

arithmeticCount[i][diff] += arithmeticCount[j][diff];

count += arithmeticCount[j][diff];

}

}

}

return count;

}

## 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;

}

## 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;

}

## 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;

}

## 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;

}

## 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;

}