<https://leetcode.com/problems/maximum-sum-circular-subarray/description/>

Given a **circular integer array** nums of length n, return the maximum possible sum of a non-empty **subarray** of nums.

A **circular array** means the end of the array connects to the beginning of the array. Formally, the next element of nums[i] is nums[(i + 1) % n] and the previous element of nums[i] is nums[(i - 1 + n) % n].

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

**Example 1:**

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

**Output:** 3

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

**Example 2:**

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

**Output:** 10

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

**Example 3:**

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

**Output:** -2

**Explanation:** Subarray [-2] has maximum sum -2.

**Constraints:**

n == nums.length

1 <= n <= 3 \* 10^4

-3 \* 10^4 <= nums[i] <= 3 \* 10^4

**Attempt 1: 2024-12-24**

**Solution 1: Kane's algorithm (10 min, similar to** [L53.Maximum Subarray (Ref.L821,L918)](note://872ABD256AD34054B5D1F2E9992E2CAA)**)**

**Style 1: O(N) space**

class Solution {

public int maxSubarraySumCircular(int[] A) {

int len = A.length;

int[] max\_dp = new int[len];

max\_dp[0] = A[0];

int max = max\_dp[0];

int sum = A[0];

for(int i = 1; i < len; i++) {

sum += A[i];

max\_dp[i] = A[i] + (max\_dp[i - 1] > 0 ? max\_dp[i - 1] : 0);

max = Math.max(max, max\_dp[i]);

}

int[] min\_dp = new int[len];

min\_dp[0] = A[0];

int min = min\_dp[0];

for(int i = 1; i < len; i++) {

min\_dp[i] = A[i] + (min\_dp[i - 1] < 0 ? min\_dp[i - 1] : 0);

min = Math.min(min, min\_dp[i]);

}

// Check max > 0 to handle corner case as all elements are negative

return max > 0 ? Math.max(max, sum - min) : max;

}

}

Time Complexity: O(n)

Space Complexity: O(n)

**Style 2: O(1) space**

class Solution {

public int maxSubarraySumCircular(int[] A) {

int len = A.length;

int max\_dp = A[0];

int max = max\_dp;

int sum = A[0];

for(int i = 1; i < len; i++) {

sum += A[i];

max\_dp = A[i] + (max\_dp > 0 ? max\_dp : 0);

max = Math.max(max, max\_dp);

}

int min\_dp = A[0];

int min = min\_dp;

for(int i = 1; i < len; i++) {

min\_dp = A[i] + (min\_dp < 0 ? min\_dp : 0);

min = Math.min(min, min\_dp);

}

// Check max > 0 to handle corner case as all elements are negative

return max > 0 ? Math.max(max, sum - min) : max;

}

}

Time Complexity: O(n)

Space Complexity: O(1)

**Style 3: O(1) space**

class Solution {

    public int maxSubarraySumCircular(int[] A) {

        int total = 0, maxSum = A[0], curMax = 0, minSum = A[0], curMin = 0;

        for (int a : A) {

            curMax = Math.max(curMax + a, a);

            maxSum = Math.max(maxSum, curMax);

            curMin = Math.min(curMin + a, a);

            minSum = Math.min(minSum, curMin);

            total += a;

        }

        return maxSum > 0 ? Math.max(maxSum, total - minSum) : maxSum;

    }

}

Time Complexity: O(n)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/maximum-sum-circular-subarray/solutions/178422/one-pass/>

**Intuition**

I guess you know how to solve max subarray sum (without circular).

If not, you can have a reference here: 53. Maximum Subarray

**Explanation**

So there are two case.

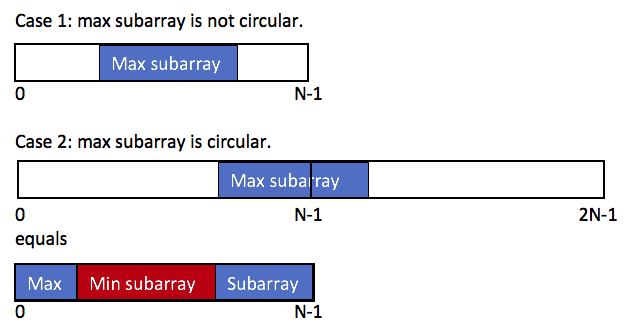
Case 1. The first is that the subarray take only a middle part, and we know how to find the max subarray sum.

Case2. The second is that the subarray take a part of head array and a part of tail array.

We can transfer this case to the first one.

The maximum result equals to the total sum minus the minimum subarray sum.

Here is a diagram by @motorix:



So the max subarray circular sum equals to

max(the max subarray sum, the total sum - the min subarray sum)

**Prove of the second case**

max(prefix+suffix)

= max(total sum - subarray)

= total sum + max(-subarray)

= total sum - min(subarray)

**Corner case**

Just one to pay attention:

If all numbers are negative, maxSum = max(A) and minSum = sum(A).

In this case, max(maxSum, total - minSum) = 0, which means the sum of an empty subarray.

According to the deacription, We need to return the max(A), instead of sum of am empty subarray.

So we return the maxSum to handle this corner case.

**Complexity**

One pass, time O(N)

No extra space, space O(1)

public int maxSubarraySumCircular(int[] A) {

int total = 0, maxSum = A[0], curMax = 0, minSum = A[0], curMin = 0;

for (int a : A) {

curMax = Math.max(curMax + a, a);

maxSum = Math.max(maxSum, curMax);

curMin = Math.min(curMin + a, a);

minSum = Math.min(minSum, curMin);

total += a;

}

return maxSum > 0 ? Math.max(maxSum, total - minSum) : maxSum;

}

**Refer to**

[L53.Maximum Subarray (Ref.L821)](note://872ABD256AD34054B5D1F2E9992E2CAA)

[L1423.Maximum Points You Can Obtain from Cards (Ref.L1658)](note://WEBb70d72cb469c39161ffaeb2d08c64a44)

[L1658.Minimum Operations to Reduce X to Zero (Ref.L918,L1423)](note://WEB2d3951b594e0ba1abab46b9e91d0033c)