Maximum Sum Circular Subarray

Solution

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`

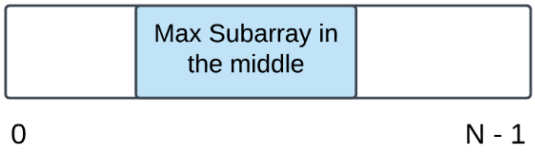
Hide Hint #1

For those of you who are familiar with the **Kadane's algorithm**, think in terms of that. For the newbies, Kadane's algorithm is used to finding the maximum sum subarray from a given array. This problem is a twist on that idea and it is advisable to read up on that algorithm first before starting this problem. Unless you already have a great algorithm brewing up in your mind in which case, go right ahead!

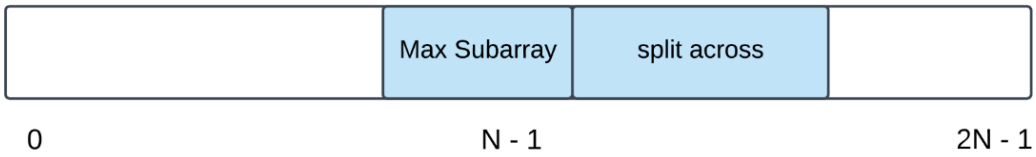
Hide Hint #2

What is an alternate way of representing a circular array so that it appears to be a straight array? Essentially, there are two cases of this problem that we need to take care of. Let's look at the figure below to understand those two cases:

Case 1



Case 2



Hide Hint #3

The first case can be handled by the good old Kadane's algorithm. However, is there a smarter way of going about handling the second case as well?

C++

```
class Solution {
public:
    int maxSubarraySumCircular(vector<int>& nums) {

    }
};
```

Solution

Notes and A Primer on Kadane's Algorithm

About the Approaches

In both Approach 1 and Approach 2, "grindy" solutions are presented that require less insight, but may be more intuitive to those with a solid grasp of the techniques in those approaches. Without prior experience, these approaches would be very challenging to emulate.

Approaches 3 and 4 are much easier to implement, but require some insight.

Explanation of Kadane's Algorithm

To understand the solutions in this article, we need some familiarity with Kadane's algorithm. In this section, we will explain the core idea behind it.

For a given array `A`, Kadane's algorithm can be used to find the maximum sum of the subarrays of `A`. Here, we only consider non-empty subarrays.

Kadane's algorithm is based on dynamic programming. Let `dp[j]` be the maximum sum of a subarray that ends in `A[j]`. That is,

$$\mathrm{dp}[j] = \max_i(A[i] + A[i+1] + \cdots + A[j])$$

Then, a subarray ending in `j+1` (such as `A[i], A[i+1] + ... + A[j+1]`) maximizes the `A[i] + ... + A[j]` part of the sum by being equal to `dp[j]` if it is non-empty, and `0` if it is. Thus, we have the recurrence:

$$\mathrm{dp}[j+1] = A[j+1] + \max(\mathrm{dp}[j], 0)$$

Since a subarray must end somewhere, $\max_j dp[j]$ must be the desired answer.

To compute `dp` efficiently, Kadane's algorithm is usually written in the form that reduces space complexity. We maintain two variables: `ans` as $\max_j dp[j]$, and `cur` as $dp[j]$; and update them as j iterates from 0 to $A.\mathrm{length} - 1$.

Then, Kadane's algorithm is given by the following psuedocode:

```
#Kadane's algorithm
ans = cur = None
for x in A:
    cur = x + max(cur, 0)
    ans = max(ans, cur)
return ans
```

Approach 1: Next Array

Intuition and Algorithm

Subarrays of circular arrays can be classified as either as *one-interval* subarrays, or *two-interval* subarrays, depending on how many intervals of the fixed-size buffer `A` are required to represent them.

For example, if `A = [0, 1, 2, 3, 4, 5, 6]` is the underlying buffer of our circular array, we could represent the subarray `[2, 3, 4]` as one interval $[2, 4]$, but we would represent the subarray `[5, 6, 0, 1]` as two intervals $[5, 6], [0, 1]$.

Using Kadane's algorithm, we know how to get the maximum of *one-interval* subarrays, so it only remains to consider *two-interval* subarrays.

Let's say the intervals are $[0, i], [j, A.\text{length} - 1]$. Let's try to compute the *i-th candidate*: the largest possible sum of a two-interval subarray for a given i. Computing the $[0, i]$ part of the sum is easy. Let's write

$$T_j = A[j] + A[j+1] + \cdots + A[A.\text{length} - 1]$$

and

$$R_j = \max_{k \geq j} T_k$$

so that the desired i-th candidate is:

$$(A[0] + A[1] + \cdots + A[i]) + R_{i+2}$$

Since we can compute T_j and R_j in linear time, the answer is straightforward after this setup.

```
class Solution {
    public int maxSubarraySumCircular(int[] A) {
        int N = A.length;

        int ans = A[0], cur = A[0];
        for (int i = 1; i < N; ++i) {
            cur = A[i] + Math.max(cur, 0);
            ans = Math.max(ans, cur);
        }

        // ans is the answer for 1-interval subarrays.
        // Now, let's consider all 2-interval subarrays.
        // For each i, we want to know
        // the maximum of sum(A[j:]) with j >= i+2

        // rightsums[i] = A[i] + A[i+1] + ... + A[N-1]
        int[] rightsums = new int[N];
        rightsums[N-1] = A[N-1];
        for (int i = N-2; i >= 0; --i)
            rightsums[i] = rightsums[i+1] + A[i];

        // maxright[i] = max_{j >= i} rightsums[j]
```

```
        int[] maxright = new int[N];
        maxright[N-1] = A[N-1];
        for (int i = N-2; i >= 0; --i)
            maxright[i] = Math.max(maxright[i+1], rightsums[i]);

        int leftsum = 0;
        for (int i = 0; i < N-2; ++i) {
            leftsum += A[i];
            ans = Math.max(ans, leftsum + maxright[i+2]);
        }

        return ans;
    }
}
```

Complexity Analysis

- Time Complexity: $O(N)$, where N is the length of `A`.
- Space Complexity: $O(N)$.

Approach 2: Prefix Sums + Monoqueue

Intuition

First, we can frame the problem as a problem on a fixed array.

We can consider any subarray of the circular array with buffer `A`, to be a subarray of the fixed array `A+A`.

For example, if `A = [0,1,2,3,4,5]` represents a circular array, then the subarray `[4,5,0,1]` is also a subarray of fixed array `[0,1,2,3,4,5,0,1,2,3,4,5]`. Let `B = A+A` be this fixed array.

Now say $N = A.\text{length}$, and consider the prefix sums

$$P_k = B[0] + B[1] + \cdots + B[k-1]$$

Then, we want the largest $P_j - P_i$ where $j - i \leq N$.

Now, consider the j-th candidate answer: the best possible $P_j - P_i$ for a fixed j. We want the i so that P_i is smallest, with $j - N \leq i < j$. Let's call this the *optimal i for the j-th candidate answer*. We can use a monoqueue to manage this.

Algorithm

Iterate forwards through j, computing the j-th candidate answer at each step. We'll maintain a `queue` of potentially optimal i's.

The main idea is that if $i_1 < i_2$ and $P_{i_1} \geq P_{i_2}$, then we don't need to remember i_1 anymore.

Please see the inline comments for more algorithmic details about managing the queue.

```
class Solution {
    public int maxSubarraySumCircular(int[] A) {
        int N = A.length;

        // Compute P[j] = B[0] + B[1] + ... + B[j-1]
        // for fixed array B = A+A
        int[] P = new int[2*N+1];
        for (int i = 0; i < 2*N; ++i)
            P[i+1] = P[i] + A[i % N];

        // Want largest P[j] - P[i] with 1 <= j-i <= N
        // For each j, want smallest P[i] with i >= j-N
        int ans = A[0];
        // deque: i's, increasing by P[i]
        Deque<Integer> deque = new ArrayDeque();
        deque.offer(0);

        for (int j = 1; j <= 2*N; ++j) {
            // If the smallest i is too small, remove it.
            if (deque.peekFirst() < j-N)
```

```
            deque.pollFirst();

        // The optimal i is deque[0], for cand. answer P[j] - P[i].
        ans = Math.max(ans, P[j] - P[deque.peekFirst()]);

        // Remove any i1's with P[i2] <= P[i1].
        while (!deque.isEmpty() && P[j] <= P[deque.peekLast()])
            deque.pollLast();

        deque.offerLast(j);
      }

      return ans;
    }
}
```

Complexity Analysis

- Time Complexity: $O(N)$, where N is the length of `A`.
- Space Complexity: $O(N)$.

Approach 3: Kadane's (Sign Variant)

Intuition and Algorithm

As in Approach 1, subarrays of circular arrays can be classified as either as *one-interval* subarrays, or *two-interval* subarrays.

Using Kadane's algorithm `kadane` for finding the maximum sum of non-empty subarrays, the answer for one-interval subarrays is `kadane(A)`.

Now, let $N = A.\text{length}$. For a two-interval subarray like:

$$(A_0 + A_1 + \cdots + A_i) + (A_j + A_{j+1} + \cdots + A_{N-1})$$

we can write this as

$$(\sum_{k=0}^{N-1} A_k) - (A_{i+1} + A_{i+2} + \cdots + A_{j-1})$$

For two-interval subarrays, let B be the array A with each element multiplied by -1. Then the answer for two-interval subarrays is $\text{sum}(A) + \text{kadane}(B)$.

Except, this isn't quite true, as if the subarray of B we choose is the entire array, the resulting two interval subarray $[0, i] + [j, N-1]$ would be empty.

We can remedy this problem by doing Kadane twice: once on B with the first element removed, and once on B with the last element removed.

```
class Solution {
    public int maxSubarraySumCircular(int[] A) {
        int S = 0;  // S = sum(A)
        for (int x : A) {
            S += x;
        }

        if (A.length == 1) {
            return S;
        }

        int ans1 = kadane(A, 0, A.length-1, 1);
        int ans2 = S + kadane(A, 1, A.length-1, -1);
        int ans3 = S + kadane(A, 0, A.length-2, -1);
        return Math.max(ans1, Math.max(ans2, ans3));
    }

    public int kadane(int[] A, int i, int j, int sign) {
        // The maximum non-empty subarray for array
        // [sign * A[i], sign * A[i+1], ..., sign * A[j]]
        int ans = Integer.MIN_VALUE;
        int cur = Integer.MIN_VALUE;
```

```
        for (int k = i; k <= j; ++k) {
            cur = sign * A[k] + Math.max(cur, 0);
            ans = Math.max(ans, cur);
        }
        return ans;
    }
}
```

Complexity Analysis

- Time Complexity: $O(N)$, where N is the length of `A`.
- Space Complexity: $O(1)$ in additional space complexity.

Approach 4: Kadane's (Min Variant)

Intuition and Algorithm

As in Approach 3, subarrays of circular arrays can be classified as either as *one-interval* subarrays (which we can use Kadane's algorithm), or *two-interval* subarrays.

We can modify Kadane's algorithm to use `min` instead of `max`. All the math in our explanation of Kadane's algorithm remains the same, but the algorithm lets us find the minimum sum of a subarray instead.

For a two interval subarray written as $(\sum_{k=0}^{N-1} A_k) - (\sum_{k=i+1}^{j-1} A_k)$, we can use our `kadane-min` algorithm to minimize the "interior" $(\sum_{k=i+1}^{j-1} A_k)$ part of the sum.

Again, because the interior $[i+1, j-1]$ must be non-empty, we can break up our search into a search on `A[1:]` and on `A[:-1]`.

```
class Solution {
    public int maxSubarraySumCircular(int[] A) {
        // S: sum of A
        int S = 0;
        for (int x : A) {
            S += x;
        }

        if (A.length == 1) {
            return S;
        }

        // ans1: answer for one-interval subarray
        int ans1 = Integer.MIN_VALUE;
```

```
        int cur = Integer.MIN_VALUE;
        for (int x : A) {
            cur = x + Math.max(cur, 0);
            ans1 = Math.max(ans1, cur);
        }

        // ans2: answer for two-interval subarray, interior in A[1:]
        int ans2 = Integer.MAX_VALUE;
        cur = Integer.MAX_VALUE;
        for (int i = 1; i < A.length; ++i) {
            cur = A[i] + Math.min(cur, 0);
            ans2 = Math.min(ans2, cur);
        }
        ans2 = S - ans2;

        // ans3: answer for two-interval subarray, interior in A[:-1]
        int ans3 = Integer.MAX_VALUE;
        cur = Integer.MAX_VALUE;
        for (int i = 0; i < A.length - 1; ++i) {
            cur = A[i] + Math.min(cur, 0);
            ans3 = Math.min(ans3, cur);
        }

        return Math.max(ans1, Math.max(ans2, ans3));
    }
}
```

Complexity Analysis

- Time Complexity: $O(N)$, where N is the length of `A`.
- Space Complexity: $O(1)$ in additional space complexity.