Previous (/articles/reverse-only-letters/) Next (/articles/binary-tree-postorder-transversal/)

918. Maximum Sub Circular Subarray (/problems/maximum-sum-circular-subarray/)

Oct. 5, 2018 | 29.1K views

Average Rating: 4.41 (46 votes)

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`

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,

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

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:

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

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 $A.\text{length} - 1$.

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.

Java Python Copy

```python
class Solution(object):
    def maxSubarraySumCircular(self, A):
        N = len(A)

        ans = cur = None
        for x in A:
            cur = x + max(cur, 0)
            ans = 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] = sum(A[i:])
        rightsums = [None] * N
        rightsums[-1] = A[-1]
        for i in xrange(N-2, -1, -1):
            rightsums[i] = rightsums[i+1] + A[i]

        # maxright[i] = max_{j >= i} rightsums[j]
        maxright = [None] * N
        maxright[-1] = rightsums[-1]
        for i in xrange(N-2, -1, -1):
            maxright[i] = max(maxright[i+1], rightsums[i])

        leftsum = 0
```

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.

Java Python Copy

```
class Solution(object):
    def maxSubarraySumCircular(self, A):
        N = len(A)

        # Compute P[j] = sum(B[:j]) for the fixed array B = A+A
        P = [0]
        for _ in xrange(2):
            for x in A:
                P.append(P[-1] + x)

        # Want largest P[j] - P[i] with 1 <= j-i <= N
        # For each j, want smallest P[i] with i >= j-N
        ans = A[0]
        deque = collections.deque([0]) # i's, increasing by P[i]
        for j in xrange(1, len(P)):
            # If the smallest i is too small, remove it.
            if deque[0] < j-N:
                deque.popleft()

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

            # Remove any i1's with P[i2] <= P[i1].
            while deque and P[j] <= P[deque[-1]]:
                deque.pop()

            deque.append(j)
```

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.

Java Python Copy

```
class Solution(object):
    def maxSubarraySumCircular(self, A):
        def kadane(gen):
            # Maximum non-empty subarray sum
            ans = cur = None
            for x in gen:
                cur = x + max(cur, 0)
                ans = max(ans, cur)
            return ans

        S = sum(A)
        ans1 = kadane(iter(A))
        ans2 = S + kadane(-A[i] for i in xrange(1, len(A)))
        ans3 = S + kadane(-A[i] for i in xrange(len(A) - 1))
        return max(ans1, ans2, ans3)
```

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]`.

Java | Python | Copy

```python
class Solution(object):
    def maxSubarraySumCircular(self, A):
        # ans1: answer for one-interval subarray
        ans1 = cur = None
        for x in A:
            cur = x + max(cur, 0)
            ans1 = max(ans1, cur)

        # ans2: answer for two-interval subarray, interior in A[1:]
        ans2 = cur = float('inf')
        for i in xrange(1, len(A)):
            cur = A[i] + min(cur, 0)
            ans2 = min(ans2, cur)
        ans2 = sum(A) - ans2

        # ans3: answer for two-interval subarray, interior in A[:-1]
        ans3 = cur = float('inf')
        for i in xrange(len(A)-1):
            cur = A[i] + min(cur, 0)
            ans3 = min(ans3, cur)
        ans3 = sum(A) - ans3

        return max(ans1, ans2, ans3)
```

Complexity Analysis

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

Rate this article:

Previous (/articles/reverse-only-letters/)

Next (/articles/binary-tree-postorder-transversal/)

Comments: 23

Sort By

Type comment here... (Markdown is supported)

Preview

Post

(/waerte)

waerte (/waerte) 17 October 14, 2018 6:49 PM

in Method 3, we can replace ans2 and ans3 with an ans2 as below:

```
int ans2 = S + kadane(A, 1, A.length-2, -1);
```

the idea is that since we use ans2 to track "2 interval" case, we need to keep at least element(0) and element(N-1) . and so these two elements should not be removed from the

Read More

7 Share Reply

SHOW 1 REPLY

(/yukangshen)

YukangShen (/yukangshen) 48 October 7, 2018 1:13 AM

In the last method, isn't there a missing line: ''ans3 = S-ans3''?

7 Share Reply

(/akhiyarov)

akhiyarov (/akhiyarov) 6 November 12, 2018 2:13 AM

This wikipedia article for Kadane algorithm has a much better explanation including the Python code:

https://en.wikipedia.org/wiki/Maximum_subarray_problem#Kadane's_algorithm (https://en.wikipedia.org/wiki/Maximum_subarray_problem#Kadane's_algorithm)

The best coders are not necessarily the best teachers.

6 Share Reply

(/pengwu550)

Pengwu550 (/pengwu550) 66 October 9, 2018 3:51 PM

the one-interval subarray and two interval subarray is hard to understand

3 Share Reply

(/alen_lee)

Alen_Lee (/alen_lee) 4 October 7, 2018 2:17 AM

Should the last approach be added the "ans3 = S - ans3" ?

3 Share Reply

(/ping_pong)

ping_pong (/ping_pong) ★ 656 December 21, 2018 10:56 PM

In approach-I why can't we use ans = Math.max(ans, leftsum + maxright[i+1]); instead of i+2 in last for loop.

2 Share Reply

SHOW 1 REPLY

(/vigorousyd)

vigorousyd (/vigorousyd) ★ 4 May 2, 2019 1:31 PM

Approach 4 - calculating both ans2 and ans3 is redundant. We can actually reduce ans2 and ans3 into one pass - just iterate i from 1 to A.length-2. That's because if A[0] or A[A.length-1] is involved, the sum is one-interval and would have already been covered in ans1. And maybe that's why the answer is missing "ans3 = S - ans3" but is still correct.

1 Share Reply

(/osamamohamed)

osamamohamed (/osamamohamed) ★ 4 October 6, 2018 11:41 PM

We can consider this problem as a variation of `Sliding Window Maximum` problem, we calculate the prefix sum on the array `B` and then apply the sliding window algorithm with length `A.size() - 1`. after that all we need to do is iterating over each element of `B` then subtracting it from the maximum subarray that immediately follows it.

1 Share Reply

(/ramkrish_123)

ramkrish_123 (/ramkrish_123) ★ 0 May 30, 2019 10:07 AM

@awice (https://leetcode.com/awice) can you please explain your intuition for the first approach?

1. I understand you partition logic.
2. But how did you come up with 2nd logic for finding the max sum subarray for each partition?

0 Share Reply

(/chro)

chro (/chro) ★ 0 March 27, 2019 4:19 PM

Does Java code for approach 4: (Kadane min variant) miss inverting the ans3 to max sum at line 32?

```
ans3 = S - ans3;
```

0 Share Reply

SHOW 1 REPLY

‹ 1 2 3 ›

Help Center (/support/) | Terms (/terms/) | Privacy (/privacy/) United States (/region/)