Solution

Approach 1: Enumerate prefix and suffix sums

Intuition

As a circular array, the maximum subarray sum can be either the maximum "normal sum" which is the maximum sum of the ordinary array or a "special sum" which is the maximum sum of a prefix sum and a suffix sum of the ordinary array where the prefix and suffix don't overlap.

The normal sum is the Maximum Subarray problem and can be solved with Kadane's algorithm. Please familiarize yourself with this solution if you haven't already. In this article, we will be using Kadane's algorithm, but we will not explain it to save time.

We can calculate both the normal sum and the special sum and return the larger one.

Assuming we already have the normal sum (it's just the solution to Maximum Subarray), let's focus on how to find the special sum.

Assume the input array is called `nums` whose length is `n`. To calculate the special sum, we need to find the maximum sum of a prefix sum and a non-overlapping suffix sum of `nums`. Our idea is to enumerate a prefix with its sum and add the maximum suffix sum that starts after the prefix so that the prefix and suffix don't overlap.

Imagine an array `suffixSum` where `suffixSum[i]` represents the suffix sum starting from index `i`, namely `suffixSum[i]` = `nums[i]` + `nums[i + 1]` + ... + `nums[n - 1]`. We can construct an array `rightMax` where `rightMax[i] = max(suffixSum[i], suffixSum[i + 1], ...suffixSum[n - 1])`.

Namely, `rightMax[i]` is the largest suffix sum of `nums` that comes at or after `i`.

With `rightMax`, we can then calculate the special sum by looking at all prefixes. We can easily accumulate the prefix while iterating over the input, and at each index `i`, we can check `rightMax[i + 1]` to find the maximum suffix that won't overlap with the current prefix.

Algorithm

The algorithm works as follows:

- Create an integer array `rightMax` of length `n`.
- Set `rightMax[n - 1]` to `nums[n - 1]`, set `suffixSum` to `nums[n - 1]`.
- Iterate over `i` from `n-2` to `0`
    - Increase `suffixSum` by `nums[i]`
    - Update `rightMax[i]` to `max(rightMax[i + 1], suffixSum)`
- Calculate the normal sum `maxSum` using Kadane's algorithm.
- Set `specialSum` to `nums[0]`, set `sum` to `0`.
- Iterate over `i` from `0` to `n - 2`
    - Increase `prefixSum` by `nums[i]`
    - Update `specialSum` to `max(specialSum, prefixSum + rightMax[i + 1])`.
- Return `max(maxSum, specialSum)`

Implementation

```
class Solution {
public:
    int maxSubarraySumCircular(vector<int>& nums) {
        const int n = nums.size();
        vector<int> right_max(n);
        right_max[n - 1] = nums[n - 1];
        for (int suffix_sum = nums[n - 1], i = n - 2; i >= 0; --i) {
            suffix_sum += nums[i];
            right_max[i] = max(right_max[i + 1], suffix_sum);
        }
        int max_sum = nums[0];
        int special_sum = nums[0];
        for (int i = 0, suffix_sum = 0, curMax = 0; i < n; ++i) {
            curMax = max(curMax, 0) + nums[i];
            // This is Kadane's algorithm.
            max_sum = max(max_sum, curMax);
            suffix_sum += nums[i];
            if (i + 1 < n) {
                special_sum = max(special_sum, suffix_sum + right_max[i +
1]);
            }
        }
        return max(max_sum, special_sum);
    }
};
```

Complexity Analysis

Here, NNN is the length of the input array.

- Time complexity: O(N)O(N)$O(N)$.

The algorithm iterates over all elements in the array to calculate the `rightMax` array, and then to find the answer. These both take linear time.

- Space complexity: O(N)O(N)$O(N)$.

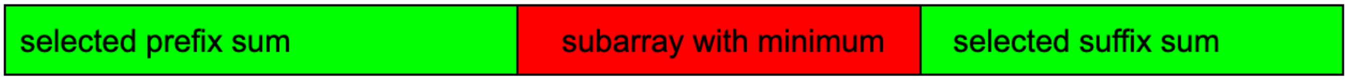
This is the space to save the `rightMax` array.

Approach 2: Calculate the "Minimum Subarray"

Intuition

As mentioned before, we know that the maximum "normal sum" is the Maximum Subarray problem which can be found with Kadane's. As such, we can focus on finding the "special sum".

Instead of thinking about the "special sum" as the sum of a prefix and a suffix, we can think about it as the sum of all elements, minus a subarray in the middle. In this case, we want to minimize this middle subarray's sum, which we can calculate using Kadane's algorithm as well.



Details to consider:

1. The minimum subarray contains at least one element which means the "special sum" never contains all elements. This is fine since the "normal sum" already takes the whole array as a

candidate.

2. If the minimum subarray contains all elements, the "special sum" will be an empty array which is invalid. In this case, all prefix or suffix sums are non-positive (otherwise, we could remove the prefix or suffix from the elements to obtain a subarray with a lower sum). So any "special sum" is non-positive. Consider 2 sub-cases:

2.1. The "normal sum" is non-negative, in this case, it's always the final answer since it's no less than the "special sum" which is 0 (because in this case, the "special sum" would be the sum of an empty array).

2.2. The "normal sum" is negative, recall that the "normal sum" takes any single element as its candidate too, this means all the elements are negative and the "normal sum" is the overall largest element, which is the final answer.

As you can see, in both cases when the minimum subarray contains all elements, the final answer is the "normal sum". We can tell if the minimum subarray contains all elements by also calculating the total sum of the array.

Algorithm

- Calculate the maximum subarray `maxSum` using Kadane's algorithm.
- Calculate the minimum subarray `minSum` using Kadane's algorithm, by using `Math.min()` instead of `Math.max()`.
- Calculate the sum of all the elements in `nums`, `sum`
- If `minSum` == `sum` return `maxSum`, otherwise return max(`maxSum`, `sum` - `minSum`).

Implementation

```
class Solution {
public:
    int maxSubarraySumCircular(vector<int>& nums) {
        int cur_max = 0, cur_min = 0, sum = 0, max_sum = nums[0], min_sum =
nums[0];
        for (int num : nums) {
            cur_max = max(cur_max, 0) + num;
            max_sum = max(max_sum, cur_max);
            cur_min = min(cur_min, 0) + num;
            min_sum = min(min_sum, cur_min);
            sum += num;
        }
        return sum == min_sum ? max_sum : max(max_sum, sum - min_sum);
    }
};
```

Complexity Analysis

Here, NNN is the length of the input array.

- Time complexity: O(N)O(N)$O(N)$.

The algorithm iterates over all elements to calculate the `maxSum`, `minSum`, and `sum` which takes O(N)O(N)$O(N)$ time.

- Space complexity: O(1)O(1)$O(1)$.

The algorithm doesn't use extra space other than several integer variables.