1186. Maximum Subarray Sum with One Deletion

```
Medium
               Dynamic Programming
        Array
```

## **Problem Description**

one element from the subarray. The subarray must remain non-empty even after a deletion, if any. Thus, the final subarray can either be a contiguous array without deletions or one where a single element has been removed. It's important to optimize the sum even if that means including negative numbers, as long as they maximally contribute to the overall sum.

The task is to find the maximum sum of a contiguous subarray from a given array of integers with the option to delete at most

Intuition

To tackle this problem, we utilize dynamic programming to consider each subarray scenario and the effect of an element deletion. The intuition behind the solution is to calculate, for every index in the array, the maximum subarray sum ending at that index (left) and starting at that index (right). We construct two arrays to keep track of these sums. The left array is filled by

traversing the array from left to right, and the right array is filled by traversing from right to left. By calculating the left maximum sums, we have the maximum subarray sum possible up to every index, without any deletion.

The right array provides us the same, but starting at each index and moving to the end. Then, for every index, assuming that index is the deleted element, we can sum the maximum totals from the left and right arrays—essentially the sums just before and just after the deleted element. This way, we consider the maximum sums possible with one deletion for every index. We also take into account the possibility of no deletion being optimal by keeping the maximum sum found while calculating the

Solution Approach The solution uses dynamic programming to find the maximum sum of a subarray that can optionally have one element deleted.

The left array is used to store the maximum subarray sum ending at each index when no deletion occurs. We traverse the array

left array. Finally, we simply return the largest sum found, which will either include no deletions or one deletion, whichever yields

## from left to right, updating the current sum s by comparing it with 0 - we never want the sum to drop below 0 because a negative

ith element.

a larger sum.

sum would decrease the total achievable sum. For each element x in the array at index i, we calculate s = max(s, 0) + x and then store that in left[i].

Following that, we initialize the right array, which stores the maximum subarray sum starting at each index and again with no deletions allowed. The traversal this time is right to left. Similar to the left array calculation, we update s for each element in a reverse manner, i.e., s = max(s, 0) + arr[i]. Now, these two arrays combined give us the maximum subarray sums before and after each index. The candidate solutions for the maximum sum with one deletion at each index i would be the sum of left[i - 1] + right[i + 1], effectively skipping the

We then compute the maximum value from the left array, which represents the maximum subarray sum without any deletion. Also, we iterate through the array to find the maximum sum if one deletion occurs, using the method described above. The returned result is the maximum sum found, ans, taking into account both scenarios - with and without a single deletion. This

approach ensures that all possible subarrays are accounted for, and the optimal solution is determined accurately.

Here's code execution visualized step by step: 1. Initialize two arrays left and right of size n with all zeroes. 2. Set a temporary sum s to 0.

4. Reset s to 0 and iterate over arr in reverse while updating right: For each element arr[i], update s to max(s, 0) + arr[i] and assign right[i] to s. 5. Find the maximum value in left, which represents the best case without deletion.

The algorithm has a linear time complexity O(n) due to the single pass made to fill both the left and right arrays.

Set s = 0. Traverse arr from left to right to fill left array:

 $\circ$  i = 5:s = max(2, 0) + 6 = 8, left[5] = 8 Now, left = [1, 0, 3, 7, 2, 8].

 $\circ$  i = 0:s = max(6, 0) + 1 = 7, right[0] = 7 Now, right = [7, 6, 8, 5, 1, 6].

 $\circ$  i = 0:s = max(0, 0) + 1 = 1, left[0] = 1

 $\circ$  i = 3:s = max(3, 0) + 4 = 7, left[3] = 7

 $\circ$  i = 4:s = max(7, 0) - 5 = 2, left[4] = 2

 $\circ$  i = 4:s = max(6, 0) - 5 = 1, right[4] = 1

 $\circ$  i = 1:s = max(8, 0) - 2 = 6, right[1] = 6

 $\circ$  i = 4: ans = max(9, 7 + 6) = 13 (deleting -5)

contiguous subarray with at most one deletion.

Solution Implementation

from typing import List

 $max_sum_left = [0] * n$ 

 $max_sum_right = [0] * n$ 

for i in range(n - 1, -1, -1):

max\_sum = max(max\_sum\_left)

for i in range(1, n - 1):

# Return the maximum sum found

max\_sum\_right[i] = current\_sum

#include <algorithm> // Include algorithm for max\_element

// Initialize vectors to keep track of maximum subarray sums

// Calculate maximum subarray sums from the left.

for (int i = 0, currentSum = 0; i < size; ++i) {

currentSum = max(currentSum, 0) + arr[i];

// Calculate maximum subarray sums from the right.

currentSum = max(currentSum, 0) + arr[i];

for (int i = size - 1, currentSum = 0; i >= 0; --i) {

int answer = \*max\_element(maxLeft.begin(), maxLeft.end());

// The maximum sum with one deletion can be found by

// Calculate maximum subarray sum from the right for each element

// Find the maximum subarray sum already calculated from the left side

maxSum = Math.max(maxSum, dpLeft[i - 1] + dpRight[i + 1]);

for (let i = arrLength - 1, currentSum = 0;  $i \ge 0$ ; --i) {

// Check if any sum can be maximized by removing one element

currentSum = Math.max(currentSum, 0) + arr[i];

answer = max(answer, maxLeft[i - 1] + maxRight[i + 1]);

int maximumSum(vector<int>& arr) {

vector<int> maxLeft(size);

vector<int> maxRight(size);

maxLeft[i] = currentSum;

maxRight[i] = currentSum;

for (int i = 1; i < size - 1; ++i) {

return answer;

dpLeft[i] = currentSum;

dpRight[i] = currentSum;

let maxSum = Math.max(...dpLeft);

// Return the maximum sum found

return maxSum;

for (let i = 1; i < arrLength - 1; ++i) {</pre>

**}**;

// from the left and from the right.

int size = arr.size();

using namespace std;

class Solution {

public:

current\_sum = 0

return max\_sum

class Solution:

**Python** 

○ Update ans to the maximum of ans and left[i - 1] + right[i + 1].

For each element x, update s to max(s, 0) + x and assign left[i] to s.

6. Iterate from the second to the second-last element of the array to check cases with one deletion:

7. Return ans which now contains the maximum subarray sum allowing for at most one deletion.

3. Iterate over the array arr while updating left:

The approach involves constructing two auxiliary arrays: left and right.

 $\circ$  i = 1: s = max(1, 0) - 2 = -1, but since we can't have negative sums reset s = 0, then left[1] = 0  $\circ$  i = 2:s = max(0, 0) + 3 = 3, left[2] = 3

Let's use a simple example to demonstrate the solution approach. Consider the array arr = [1, -2, 3, 4, -5, 6].

Initialize left and right arrays of size 6 (the size of arr) with all zeroes: left = [0, 0, 0, 0, 0, 0], right = [0, 0, 0, 0,

```
Reset s = 0. Traverse arr from right to left to fill right array:
\circ i = 5:s = max(0, 0) + 6 = 6, right[5] = 6
```

**Example Walkthrough** 

0, 0].

 $\circ$  i = 3:s = max(1, 0) + 4 = 5, right[3] = 5  $\circ$  i = 2:s = max(5, 0) + 3 = 8, right[2] = 8

```
Find the maximum value in left, which is 8.
Iterate from the second to the second-last element of the array to check cases with one deletion. We need to consider left[i
-1] + right[i + 1]:
\circ i = 1: ans = max(0, 1 + 8) = 9 (deleting -2)
\circ i = 2: ans = max(9, 1 + 5) = 9 (no change, deletion of 3 not beneficial)
\circ i = 3: ans = max(9, 3 + 1) = 9 (no change, deletion of 4 not beneficial)
```

The final result for this example is a sum of 13, which is achieved by the subarray [1, 3, 4, 6] with the deletion of -5. This

walkthrough illustrates how the dynamic programming solution accounts for all possible scenarios to find the maximum sum of a

def maximumSum(self, arr: List[int]) -> int: # Calculate the length of the input array n = len(arr)

# Traverse the array and find the maximum sum by potentially removing one element

max\_sum = max(max\_sum, max\_sum\_left[i - 1] + max\_sum\_right[i + 1])

# Initialize two lists to store the maximum subarray sum from left and right

# Calculate the maximum subarray sum from the right

current\_sum = max(current\_sum, 0) + arr[i]

# Find the maximum sum of the non-empty subarray

Return ans which is now 13, the maximum subarray sum if at most one deletion is allowed.

# Calculate the maximum subarray sum from the left current\_sum = 0 for i, value in enumerate(arr): current\_sum = max(current\_sum, 0) + value max\_sum\_left[i] = current\_sum # Reset the current sum for the next loop

```
# Example usage:
# sol = Solution()
# print(sol_maximumSum([1, -2, 0, 3])) # Output: 4
Java
class Solution {
    public int maximumSum(int[] arr) {
        int n = arr.length; // Store the length of the array.
        int[] leftMaxSum = new int[n]; // Maximum subarray sum ending at each index from the left.
        int[] rightMaxSum = new int[n]; // Maximum subarray sum starting at each index from the right.
        int maxSum = Integer.MIN_VALUE; // Initialize maxSum with the smallest possible integer value.
       // Calculate the maximum subarray sum from the left and find the max subarray sum without deletion.
        int currentSum = 0; // Initialize a variable to keep track of the current summation.
        for (int i = 0; i < n; ++i) {
            currentSum = Math.max(currentSum, 0) + arr[i]; // Calculate the sum while resetting if negative.
            leftMaxSum[i] = currentSum; // Store the maximum sum up to the current index from the left.
           maxSum = Math.max(maxSum, leftMaxSum[i]); // Update the maximum subarray sum seen so far.
       // Calculate the maximum subarray sum from the right.
        currentSum = 0; // Reset the currentSum for the right max subarray calculation.
        for (int i = n - 1; i >= 0; --i) {
            currentSum = Math.max(currentSum, 0) + arr[i]; // Calculate the sum while resetting if negative.
            rightMaxSum[i] = currentSum; // Store the maximum sum up to the current index from the right.
       // Check the maximum sum by considering deleting one element from the array.
        for (int i = 1; i < n - 1; ++i) {
           // Sum of subarrays left to i excluding arr[i] and right to i excluding arr[i] provides a sum
           // for the array with arr[i] deleted. Update maxSum if this sum is larger.
           maxSum = Math.max(maxSum, leftMaxSum[i - 1] + rightMaxSum[i + 1]);
       return maxSum; // Return the maximum subarray sum with at most one element deletion.
C++
#include <vector>
```

```
TypeScript
function maximumSum(arr: number[]): number {
   const arrLength = arr.length; // Number of elements in the array
   const dpLeft: number[] = new Array(arrLength); // Dynamic programming array storing max sum from the left
   const dpRight: number[] = new Array(arrLength); // Dynamic programming array storing max sum from the right
   // Calculate maximum subarray sum from the left for each element
   for (let i = 0, currentSum = 0; i < arrLength; ++i) {</pre>
       currentSum = Math.max(currentSum, 0) + arr[i];
```

// Initialize the answer with the maximum subarray sum that contains no deletion.

// adding the maximum subarray sum to the left of the deleted element

// and the maximum subarray sum to the right of the deleted element.

// Iterate through the array to find the maximum sum obtainable by deleting one element.

```
from typing import List
class Solution:
   def maximumSum(self, arr: List[int]) -> int:
       # Calculate the length of the input array
       n = len(arr)
       # Initialize two lists to store the maximum subarray sum from left and right
       \max sum left = [0] * n
       max_sum_right = [0] * n
       # Calculate the maximum subarray sum from the left
        current_sum = 0
        for i, value in enumerate(arr):
            current_sum = max(current_sum, 0) + value
           max sum left[i] = current sum
       # Reset the current sum for the next loop
       current_sum = 0
       # Calculate the maximum subarray sum from the right
        for i in range(n - 1, -1, -1):
```

# Traverse the array and find the maximum sum by potentially removing one element

max\_sum = max(max\_sum, max\_sum\_left[i - 1] + max\_sum\_right[i + 1])

current\_sum = max(current\_sum, 0) + arr[i]

# Find the maximum sum of the non-empty subarray

max\_sum\_right[i] = current\_sum

max\_sum = max(max\_sum\_left)

for i in range(1, n - 1):

Time and Space Complexity

return max\_sum

# Example usage:

# sol = Solution()

# Return the maximum sum found

**Time Complexity** The given Python code has a time complexity of O(n), where n is the length of the input array arr.

Here's the breakdown of the time complexity:

#  $print(sol_maximumSum([1, -2, 0, 3]))$  # Output: 4

iterations exactly. • The second for loop calculates the maximum subarray sums starting at each index (right[]), iterating backward through the array. It also executes n iterations.

Here is the breakdown of the space complexity:

- Computing the maximum value of left[] is done in O(n) time. • Another loop runs through the indices from 1 to n - 2 to find the maximum sum obtained by possibly removing one element. This again is a single pass through the array, which accounts for n operations.
- Each operation within the loops is done in constant time 0(1). Regarding time complexity, we perform a constant amount of work n times, leading to the overall time complexity of O(n).

• The first for loop iterates through the array to compute the maximum subarray sums ending at each index (left[]). This loop runs for n

- **Space Complexity** The space complexity of the code is O(n) due to the additional space used for the left[] and right[] arrays, each of size n.

Since the size of the two arrays scales with the input, the total additional space required by the algorithm is linear with respect to the input size, thus the space complexity is O(n).

• Two arrays of size n (left[] and right[]) are created to store the cumulative sums. • A few variables like n, s, i, x, and ans are also used which take up constant space.