LEETCODE SOLUTION ARTICLE

1186. Maximum Subarray Sum with One Deletion

Bryce Kille

Full list of author information is available at the end of the article

Abstract

Problem Statement:

Given an array of integers, return the maximum sum for a **non-empty subarray** (contiguous elements) with at most one element deletion. In other words, you want to choose a subarray and optionally delete one element from it so that there is still at least one element left and the sum of the remaining elements is maximum possible.

Note that the subarray needs to be **non-empty subarray** after deleting one element.

Example 1:

```
Input: arr = [1,-2,0,3]
Output: 4
Explanation: Because we can choose [1, -2, 0, 3] and drop
   -2, thus the subarray [1, 0, 3] becomes the maximum
   value.
```

Example 2:

```
Input: arr = [1,-2,-2,3]
Output: 3
Explanation: We just choose [3] and it's the maximum sum.
```

Example 3:

```
Input: arr = [-1,-1,-1,-1]
Output: -1
Explanation: The final subarray needs to be non-empty. You
   can't choose [-1] and delete -1 from it, then get an
   empty subarray to make the sum equals to 0.
```

Solution: By slightly modifying the popular Kadane's algorithm for the Maximum Subarray Problem, we can obtain a simple $O(n)$ time $O(1)$ space solution to the problem.

Keywords: Dynamic Programming

1 Background

Let us first refresh our memory on the simpler version of this problem, where we want to find the maximum subarray sum **without** deleting any elements. For the

remainder of this article, we refer to a subarray with a maximal sum as a maximal subarray.

If we want to solve the simple maximum subarray sum problem using dynamic programming, we need to split the problem up into more manageable chunks. Let's consider the maximal subarray which ends at index i, $0 \leq i < n$ in the array (n is the size of the array). We know that this maximal subarray will have one of the two following characteristics:

- It is a singleton i.e. the maximal subarray ending at index i is simply the the range $[i, i+1)$
- It extends the maximal subarray ending at the position $i-1$.

Now it is clear that at every index i, to compute the maximal subarray ending at i, we simply need to take the maximum of the two cases above:

$$maxSubarray(arr, i) = \max \begin{cases} arr[i] \\ arr[i] + maxSubarray(arr, i-1) \quad i > 0 \end{cases}$$

And we can obtain the solution in linear time by simply starting with $i = 0$ and looping over the entire array, keeping track of the maximum subarray seen as well as the size of the maximum subarray which ends at position $i-1$.

2 Solution

2.1 Intuition

We now move on to the maximum subarray sum with one deletion. The common theme in DP problems is to break down the problem into managable subproblems, for each of which you have a set of properties. In the Maximum Subarray Sum problem, we broke down the problem into maximum subarrays ending at a specific index and realized that they have exactly one of two properties. If we add the option to delete an element in a subarray, how can we incorporate this into our properties and subproblems?

We first realize that we can keep our subproblems the same: The maximum subarray must end at some index i, so keeping track of such values still makes sense. What about the properties? Well, our original two properties still hold, as the solution space still allows for contiguous subarrays. However, now we want to also keep track of "skipped subarrays" i.e. subarrays with one element deleted from them. It remains to determine the properties of a maximal skipped subarray ending at position i. Again, the maximal skipped subarray ending at index i boils down to having one of two properties:

- The maximal skipped subarray ending at index i has already skipped some element, and therefore i is included.
- The maximal skipped subarray ending at index i skips element i and therefore it's value is that of the maximum contiguous subarray ending at $i-1$.

Therefore, we have the following function (which is $-\infty$ for $i = 0$, as no skipped subarray can end at index 0):

$$maxSkippedSubarray(arr, i) = \max \begin{cases} maxSkippedSubarray(arr, i-1) + arr[i] & i > 0 \\ maxSubarray(arr, i-1) & i > 0 \\ -\infty & i = 0 \end{cases}$$

Similar to the previous section, we can obtain the solution in $O(n)$ time $O(1)$ space by setting the base case $i = 0$ and then looping over the rest of the array, computing $maxSubarray(arr, i)$ and $maxSkippedSubarray(arr, i)$ at every index and keeping track of the maximum subarray and maximum skipped subarray seen.

2.2 C++ Implementation

```
class Solution {
    public:
    int maximumSum(vector<int>& arr) {
        // Edge case:
        if (arr.size() == 0) return 0;

        // Base case:
        int max_sum = arr[0];
        int current_sum = arr[0];
        int skipped_current_sum = 0;

        for (int i = 1; i < arr.size(); ++i) {
            // Compute maxSkippedSubarray(arr, i)
            skipped_current_sum = max(
                skipped_current_sum + arr[i],
                current_sum
            );

            // Compute maxSubarray(arr, i)
            current_sum = max(arr[i], current_sum + arr[i]);

            // Keep track of maximum value seen
            max_sum = max(
                max_sum, max(
                    current_sum,
                    skipped_current_sum
                )
            );
        }
        return max_sum;
    }
};
```

2.3 Complexity Analysis

The algorithm above is $O(n)$ time and $O(1)$ space. We do constant work in the base case, as well as at each iteration of the loop while only using a constant number of variables.