53. Maximum Subarray

Medium | Array | Divide and Conquer | Dynamic Programming

Problem Description

The problem gives us an array of integers called `nums`. Our task is to find a contiguous subarray within this array that adds up to the maximum sum possible and then return this maximum sum. The term "subarray" here refers to a sequence of consecutive elements from the original array. It's important to note that the array may contain both positive and negative numbers, and the subarray with the largest sum must contain at least one element.

Intuition

To solve this problem, we use a well-known algorithm called Kadane's algorithm. The intuition behind this approach is to iterate through each element in the array while keeping track of two important variables: `f` and `ans`. Here, `f` tracks the maximum sum of the subarray ending at the current position, and `ans` keeps the overall maximum sum found so far.

At each step of the iteration, we decide whether to "start fresh" with the current element (if the sum up to this point is negative, since it would only reduce the sum of the subarray) or to add it to the current running sum `f`. We do this by comparing `f` with `0` (essentially dropping the subarray if `f` is negative) and then add the current element to `f`.

Then, we update `ans` to be the maximum of `ans` or the new sum `f`. By the end of the iteration, we would have examined every subarray and `ans` holds the value of the largest subarray sum.

The key idea is to keep adding elements to the current sum if it contributes positively and start a new subarray sum whenever the running sum becomes negative.

Solution Approach

The implementation of the solution is straightforward once the intuition behind the problem is clear. The solution uses no additional data structures other than simple variables for tracking the current sum and the maximum sum.

The algorithm initializes `ans` and `f` with the first element of the array. It assumes that the best subarray could at least be the first element itself. Then it begins to iterate from the second element of the array all the way to the last element.

For each element `x` in the array:

1. We update `f` to be the maximum of `f` + `x` or `0` + `x`. The reason we compare with `0` is to decide whether to start a new subarray from the current element (in case the previous `f` was negative and thus, doesn't help in increasing the sum).

   This is implemented as:

   ```
   f = max(f, 0) + x
   ```

2. We update `ans` to be the maximum of `ans` or the new `f`. This way, we ensure `ans` always holds the maximum sum found so far.

   This is implemented as:

   ```
   ans = max(ans, f)
   ```

3. After the loop terminates, `ans` will hold the maximum subarray sum that we are looking for, which gets returned as the result.

This approach only requires O(1) extra space for the tracking variables and O(n) time complexity, as it passes through the array only once. It's a prime example of an efficient algorithm that combines simple ideas to solve a problem that might seem complex at first glance.

Example Walkthrough

Let's walk through an example to illustrate the solution approach. Consider the following array of integers:

```
nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
```

We want to find a contiguous subarray that has the maximum sum. According to Kadane's algorithm, we initialize our tracking variables:

```
f = nums[0] = -2
ans = nums[0] = -2
```

Now, let's iterate through the array starting from the second element.

1. At index 1, `nums[1] = 1`
   - We calculate `f = max(f + nums[1], nums[1]) = max(-2 + 1, 1) = 1`
   - We then update `ans = max(ans, f) = max(-2, 1) = 1`
2. At index 2, `nums[2] = -3`
   - Update `f = max(f + nums[2], nums[2]) = max(1 + (-3), -3) = max(-2, -3) = -2`
   - Since `f` is less than `ans`, we don't update `ans`. `ans` remains 1.
3. At index 3, `nums[3] = 4`
   - Update `f = max(f + nums[3], nums[3]) = max(-2 + 4, 4) = 4`
   - Update `ans = max(ans, f) = max(1, 4) = 4`
4. At index 4, `nums[4] = -1`
   - Update `f = max(f + nums[4], nums[4]) = max(4 + (-1), -1) = 3`
   - `ans` remains the same as `f` is less than `ans`.
5. At index 5, `nums[5] = 2`
   - Update `f = max(f + nums[5], nums[5]) = max(3 + 2, 2) = 5`
   - Update `ans = max(ans, f) = max(4, 5) = 5`
6. At index 6, `nums[6] = 1`
   - Update `f = max(f + nums[6], nums[6]) = max(5 + 1, 1) = 6`
   - Update `ans = max(ans, f) = max(5, 6) = 6`
7. At index 7, `nums[7] = -5`
   - Update `f = max(f + nums[7], nums[7]) = max(6 + (-5), -5) = 1`
   - `ans` remains 6.
8. Finally, at index 8, `nums[8] = 4`
   - Update `f = max(f + nums[8], nums[8]) = max(1 + 4, 4) = 5`
   - Update `ans = max(ans, f) = max(6, 5) = 6`

After iterating through all elements, we find that the maximum sum of a contiguous subarray in `nums` is `6`, and the contiguous subarray that gives this sum is `[4, -1, 2, 1]`. Thus our function would return `6` as the final result.

Python Solution

```python
from typing import List

class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        # Initialize the maximum subarray sum with the first element.
        max_sum = current_sum = nums[0]

        # Iterate through the remaining elements in the list starting from the second element.
        for num in nums[1:]:
            # Update the current subarray sum. Add the current number to the current sum,
            # or reset it to the current number if the current sum is negative.
            current_sum = max(current_sum + num, num)

            # Update the maximum subarray sum if the current subarray sum is greater.
            max_sum = max(max_sum, current_sum)

        # Return the maximum subarray sum found.
        return max_sum

```

Java Solution

```java
class Solution {
    public int maxSubArray(int[] nums) {
        // `maxSoFar` holds the maximum subarray sum found so far
        int maxSoFar = nums[0];
        // `currentMax` holds the maximum sum of the subarray ending at the current position
        int currentMax = nums[0];

        // Loop through the array starting from the second element
        for (int i = 1; i < nums.length; ++i) {
            // Update `currentMax` to be the maximum of `currentMax` + current element or 0 + current element
            // This is the essence of the Kadane's algorithm which decides whether to start a new subarray or continue with the curre
            currentMax = Math.max(currentMax, 0) + nums[i];

            // If the current computed `currentMax` is greater than `maxSoFar`, update `maxSoFar`
            maxSoFar = Math.max(maxSoFar, currentMax);
        }
        // Return the largest sum
        return maxSoFar;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // for std::max

class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        // Initialize current max to the first element of the vector
        int currentMax = nums[0];
        // Initialize global max with the same value
        int globalMax = nums[0];

        // Loop through the elements starting from the second element
        for (int i = 1; i < nums.size(); ++i) {
            // Update current max; if it becomes negative, reset it to zero
            currentMax = std::max(currentMax, 0) + nums[i];
            // Update global max with the maximum value between current and global max
            globalMax = std::max(globalMax, currentMax);
        }

        // Final answer which is the maximum subarray sum
        return globalMax;
    }
};

```

Typescript Solution

```typescript
/**
 * Finds the contiguous subarray within an array (containing at least one number)
 * which has the largest sum and returns that sum.
 * @param nums The array of numbers.
 * @return The maximum subarray sum.
 */
function maxSubArray(nums: number[]): number {
    // Initialize the answer and the running sum with the first element of the array.
    let maxSum = nums[0];
    let currentSum = nums[0];

    // Iterate over the array starting from the second element.
    for (let i = 1; i < nums.length; ++i) {
        // Update the current sum to be the maximum between the current sum with zero (to discard negative sums)
        // and then add the current element to include it in the subarray.
        currentSum = Math.max(currentSum, 0) + nums[i];

        // Update the maximum sum if the current sum is greater.
        maxSum = Math.max(maxSum, currentSum);
    }

    // Return the final maximum sum found.
    return maxSum;
}

```

Time and Space Complexity

Time Complexity

The given code snippet consists of a single loop that iterates through the list `nums`. The loop starts from the second element and goes till the last element, performing constant time operations in each iteration. The `max` function is also O(1). Therefore, the time complexity is O(n), where `n` is the number of elements in the input list `nums`.

Space Complexity

The space complexity of the algorithm is O(1). It only uses a fixed amount of extra space: two integer variables `ans` and `f` to store the maximum sum and the current sum, respectively. These do not depend on the size of the input list, thus the algorithm uses constant extra space.