Maximum Subarray Sum

This section discusses the problem of finding the maximum subarray sum within a given array and explores different approaches to solve it, including brute force, a better approach, and the optimal Kadane's Algorithm.

Understanding the Problem

The problem is: given an array, find the subarray (a contiguous part of the array) that has the largest sum.

> A **subarray** is a contiguous part of an array. This means elements must be adjacent.

For example, in the array `[-2, -3, 4, -1, -2, 1, 5, -3]`, the subarray `[4, -1, -2, 1, 5]` has the largest sum, which is `7`.

Important points to note:

- A single element can be a subarray.
- The entire array can be a subarray.
- A subsequence is NOT a subarray if it has discontinuity.

Brute Force Approach

The initial approach to solving this problem involves trying out all possible subarrays and calculating their sums. The subarray with the maximum sum is then returned.

The algorithm iterates through all possible subarrays using three nested loops:

1. The outer loop (i) stands at the starting element of the subarray and goes from the zeroth index to the last index.
2. The second loop (j) iterates from the current starting element (i) to the end of the array, thus creating all possible subarrays starting from index i.
3. The innermost loop (k) calculates the sum of the elements within the current subarray (from i to j).

   $sum = 0$ for i from 0 to n: for j from i to n: for k from i to j: $sum+ = array[k]$ $maximum = max(sum, maximum)$

- Time Complexity: $O(n^3)$ (approximately, due to the loops not always running exactly n times).
- Space Complexity: $O(1)$ (no extra space is used).

Better Approach

To optimize the brute force approach, we can avoid recalculating the sum of each subarray from scratch. Instead, we can leverage the sum of the previous subarray.

Instead of using a third nested loop to compute the sum of each subarray, the sum is updated as the inner loop progresses. This reduces the time complexity from $O(n^3)$ to $O(n^2)$.

```
$sum = 0$
for i from 0 to n:
    for j from i to n:
        $sum += array[j]$
        $maximum = max(sum, maximum)$
```

- Time Complexity: $O(n^2)$
- Space Complexity: $O(1)$

Kadane's Algorithm (Optimal Approach)

To find the most optimal solution, we will be using something known as the Kadane's algorithm. The main idea behind Kadane's Algorithm is to look for contiguous subarrays with positive sums. If the current sum becomes negative, it makes sense to discard it and start with a new subarray.

The algorithm works as follows:

1. Initialize `maximum` to the lowest possible integer value (`int_min`) and `sum` to `0`.
2. Iterate through the array, adding each element to the current `sum`.
3. After each addition, compare `sum` with `maximum` and update `maximum` if `sum` is greater.
4. If `sum` becomes negative, reset it to `0` because a negative sum will always reduce the sum of any subarray it is part of.

   $maximum = int_min\ sum = 0$ for i from 0 to n: $sum += array[i]$ $maximum = max(sum, maximum)$ if $sum < 0$: $sum = 0$

- Time Complexity: $O(n)$
- Space Complexity: $O(1)$

Kadane's Algorithm: Finding the Maximum Subarray Sum

Kadane's algorithm is an efficient way to find the maximum sum of a contiguous subarray within a given array. It works by iterating through the array, keeping track of the current sum and the maximum sum encountered so far.

How Kadane's Algorithm Works

1. **Initialization**:
   - Start with `current_sum = 0` and `max_so_far = -infinity`.
2. **Iterate Through the Array**:
   - For each element, add it to the `current_sum`.
   - If `current_sum` becomes negative, reset it to `0` (start a new subarray).
   - Update `max_so_far` with the maximum of `max_so_far` and `current_sum`.

Example Walkthrough

Consider the array: `[-2, -3, 4, -1, -2, 1, 5, -3]`

Element	Current Sum	Maximum Sum	Decision
-2	-2	-2	`current_sum` is not > `max_so_far`.
-3	-3	-2	Reset `current_sum` to 0 (since it's negative).
4	4	4	Start a new subarray. `current_sum` becomes 4, which is greater than the initial `max_so_far`.
-1	3	4	Add -1 to `current_sum`.
-2	1	4	Add -2 to `current_sum`.
1	2	4	Add 1 to `current_sum`.
5	7	7	Add 5 to `current_sum`. New maximum found!
-3	4	7	Add -3 to `current_sum`.

The maximum subarray sum is 7.

Code Implementation

Here's a basic implementation of Kadane's algorithm:

```cpp
#include <iostream>
#include <algorithm>
#include <climits> // For INT_MIN

using namespace std;

int kadane(int arr[], int n) {
    long long current_sum = 0;
    long long max_so_far = LLONG_MIN; // Initialize with the smallest poss

    for (int i = 0; i < n; i++) {
        current_sum += arr[i];
        max_so_far = max(max_so_far, current_sum);
        if (current_sum < 0) {
            current_sum = 0;
        }
    }
    return max_so_far;
}

int main() {
    int arr[] = {-2, -3, 4, -1, -2, 1, 5, -3};
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Maximum subarray sum is " << kadane(arr, n) << endl;
    return 0;
}
```

Handling Empty Subarrays

In some problems, you might need to consider the possibility of an empty subarray. If all elements are negative, the maximum subarray sum would be 0 (representing an empty subarray).

```cpp
if (max_so_far < 0) {
    max_so_far = 0;
}
```

Edge Case Example

Consider the array `[-4, -2, -3, -1]`. The algorithm would identify -1 as the "least negative" number. However, if the problem statement specifies that an empty subarray should be considered with a sum of 0, the result should be 0.

Follow-Up: Printing the Subarray

To print the subarray with the maximum sum, keep track of the start and end indices of the subarray.

```cpp
#include <iostream>
#include <algorithm>
#include <climits>

using namespace std;

void kadaneWithSubarray(int arr[], int n) {
    long long current_sum = 0;
    long long max_so_far = LLONG_MIN;
    int start = 0;
    int end = 0;
    int temp_start = 0;

    for (int i = 0; i < n; i++) {
        current_sum += arr[i];

        if (current_sum > max_so_far) {
            max_so_far = current_sum;
            start = temp_start;
            end = i;
        }

        if (current_sum < 0) {
            current_sum = 0;
            temp_start = i + 1;
        }
    }

    cout << "Maximum subarray sum is " << max_so_far << endl;
    cout << "Subarray: ";
    for (int i = start; i <= end; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    int arr[] = {-2, -3, 4, -1, -2, 1, 5, -3};
    int n = sizeof(arr) / sizeof(arr[0]);
    kadaneWithSubarray(arr, n);
    return 0;
}
```

Time and Space Complexity

- **Time Complexity**: $O(n)$ because we iterate through the array once.
- **Space Complexity**: $O(1)$ because we use a constant amount of extra space.

Key Takeaways

- Kadane's algorithm is a **dynamic programming** approach to solve the maximum subarray sum problem efficiently.
- It cleverly uses the idea that a negative sum will only decrease the sum of a future subarray, so it's better to start a new subarray from the next element.
- It can be modified to also return the start and end indices of the maximum sum subarray.