1014. Best Sightseeing Pair

Medium Array Dynamic Programming Leetcode Link

Problem Description

In this problem, you are given an array called `values`, where each element `values[i]` signifies the value of the i-th sightseeing spot. Sightseeing spots have a score that can be calculated based on their values and their distances from each other. Specifically, if you pick two spots at indices `i` and `j` such that `i < j`, the score is determined by the formula `values[i] + values[j] + i - j`. This score accounts for both the value of each spot and the cost of distance between them, since a higher distance decreases the overall score.

The challenge is to find the maximum possible score that can be obtained from any pair of sightseeing spots (`i`, `j`).

Intuition

The obvious solution might seem to try all possible pairs and compute the score for each, but this approach is not efficient for large arrays. Instead, we can optimize by recognizing that for any spot `j`, we want the highest possible value of `values[i] + i` from all previous spots `i` (since `i` is less than `j`, this keeps the distance cost minimal). That is because the score `values[i] + values[j] + i - j` can be rewritten as `(values[i] + i) + (values[j] - j)`.

The solution proceeds by keeping track of the maximum value of `values[i] + i` encountered so far as it loops from left to right across the elements of the array. For each spot `j`, it calculates the potential score by adding `values[j] - j` to this maximum. This potential score is compared to the maximum score found so far, and if it's higher, it becomes the new maximum score.

By keeping track of these two quantities — the maximum score so far, and the maximum value of `values[i] + i` — the solution efficiently arrives at the maximum score possible for any pair of sightseeing spots.

Solution Approach

The solution approach implements a single pass algorithm with a time complexity of O(n), where n is the length of the `values` array. It utilizes a simple pattern that leverages the relationship between indices and their corresponding values, as described in the intuition part.

Here's the step-by-step description of the algorithm:

1. Initialize two variables: `ans` to store the maximum score found so far and `mx` to store the maximum value of `values[i] + i` seen as the array is traversed. `mx` is initially set to `values[0]` since it's the first value plus its index.

2. Iterate over the array starting from index 1, since the first element is already considered as part of the initial maximum value (`mx`).

3. For each sightseeing spot `j` (from index 1 to the end):

   a. Calculate the current potential score as `values[j] - j + mx`. This utilizes the previously mentioned formula of the best pair score `(values[i] + i) + (values[j] - j)`. Here, `mx` represents the best `values[i] + i` found so far, for some `i < j`.

   b. Update `ans` to the greater value between itself and the current potential score. This ensures that at the end of the iteration, `ans` holds the maximum score of all pairs.

   c. Update `mx` to the greater value between itself and `values[j] + j`. This is crucial because, as we move rightward through the array, we might find a new sightseeing spot `j` with a higher value of `values[j] + j` that could contribute to a higher pair score in the following steps.

4. After the loop, `ans` holds the maximum score for a pair that could be achieved and is returned as the result.

This single-pass algorithm is elegant and efficient as it avoids the need for a nested loop, which would result in a less efficient O(n^2) complexity. Instead, by updating `mx` and `ans` on-the-fly, it computes the maximum score in linear time, thus making it suitable for large datasets.

The python code implementing this algorithm is straightforward:

```python
class Solution:
    def maxScoreSightseeingPair(self, values: List[int]) -> int:
        ans, mx = 0, values[0]  # Initialize `ans` and `mx`
        for j in range(1, len(values)):  # Iterate through the array, starting from index 1
            ans = max(ans, values[j] - j + mx)  # Update the maximum score `ans`
            mx = max(mx, values[j] + j)  # Update the running maximum of `values[i] + i`
        return ans  # After iterating through the array, return the maximum score `ans`
```

This solution does not use any complex data structures; it only requires two variables to keep track of the scores, making it space-efficient with O(1) additional space complexity.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Suppose we have the following `values` array:

```
values = [8, 1, 5, 2, 6]
```

We want to find the maximum score sightseeing pair using the formula `values[i] + values[j] + i - j`. Following the steps from the solution approach:

1. We initialize `ans` to 0 and `mx` to `values[0]`, so initially `ans = 0` and `mx = 8`.

2. We iterate from the second element (since we already have `mx` starting with the first element). Now we start with `j = 1`, and `values[1] = 1`.

3. We perform the following actions for each `j`:

   a. For `j = 1` (value is 1), we calculate the potential score: `1 - 1 + mx = 1 - 1 + 8 = 8`. Now we compare `ans` with this score and update `ans`: `ans = max(0, 8) = 8`.

   b. Next, we update `mx`: `mx = max(mx, values[j] + j) = max(8, 1 + 1) = 8`. It remains the same since the new `values[j] + j` is not greater than `mx`.

   c. Moving to `j = 2` (value is 5): The potential score is `5 - 2 + mx = 5 - 2 + 8 = 11`, update `ans`: `ans = max(8, 11) = 11`, and update `mx` : `mx = max(8, 5 + 2) = 8`. Again, `mx` does not change.

   d. For `j = 3` (value is 2): The potential score is `2 - 3 + mx = 2 - 3 + 8 = 7`, update `ans`: `ans = max(11, 7) = 11`, and update `mx`: `mx = max(8, 2 + 3) = 8`.

   e. Lastly, for `j = 4` (value is 6): The potential score is `6 - 4 + mx = 6 - 4 + 8 = 10`, update `ans`: `ans = max(11, 10) = 11`, and update `mx`: `mx = max(8, 6 + 4) = 10`.

4. At the end of the loop, `ans` holds the value 11, which is the maximum score that could be achieved with any pair of sightseeing spots, specifically the pair (0, 2), corresponding to the spots with values 8 and 5, respectively.

So, the result for our example `values = [8, 1, 5, 2, 6]` is `11`, which is the maximum score sightseeing pair (`values[0] + values[2] + 0 - 2`). The solution was able to find this in a single linear pass as opposed to checking every possible pair, saving time and computational resources.

Python Solution

```python
from typing import List

class Solution:
    def maxScoreSightseeingPair(self, values: List[int]) -> int:
        max_score = 0  # Initialize the maximum score to zero
        max_value_plus_index = values[0]  # Initialize to the first value plus its index (0)

        # Iterate over the array, starting from the second element (index 1)
        for i in range(1, len(values)):
            # Update the max score using the current value and the best previous value plus index found
            max_score = max(max_score, values[i] - i + max_value_plus_index)
            # Update the max_value_plus_index with the maximum of the current and the previous
            # while accounting for the increasing index
            max_value_plus_index = max(max_value_plus_index, values[i] + i)

        # Return the maximum score found for any sightseeing pair
        return max_score

```

Java Solution

```java
class Solution {
    public int maxScoreSightseeingPair(int[] values) {
        // Initialize the answer to 0. This will hold the maximum score.
        int maxScore = 0;

        // Initialize the maximum value seen so far, which is the value at the 0th index
        // plus its index (because for the first element, index is 0, so it's just the value).
        int maxValueWithIndex = values[0];

        // Iterate over the array starting from the 1st index since we've already considered the 0th index.
        for (int j = 1; j < values.length; ++j) {
            // Update maxScore with the maximum of the current maxScore and
            // the score of the current sightseeing spot combined with the previous maximum.
            // This score is computed as the value of the current element plus its 'value' score (values[j])
            // subtracted by its distance from the start (j) plus the maxValueWithIndex.
            maxScore = Math.max(maxScore, values[j] - j + maxValueWithIndex);

            // Update the maxValueWithIndex to be the maximum of the current maxValueWithIndex and
            // the 'value' score of current element added to its index (values[j] + j).
            // This accounts for the fact that as we move right, our index increases,
            // which decreases our score, so we need to keep track of the element
            // which will contribute the most to the score including the index.
            maxValueWithIndex = Math.max(maxValueWithIndex, values[j] + j);
        }

        // Return the maximum score found.
        return maxScore;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // For max function
using namespace std;

class Solution {
public:
    int maxScoreSightseeingPair(vector<int>& values) {
        int maxScore = 0; // This will store the maximum score seen so far
        int maxIPlusValue = values[0]; // This keeps track of the maximum (values[i] + i)

        // Start from the second element because we need at least two elements for a pair
        for (int j = 1; j < values.size(); ++j) {
            // Update the max score considering the current element as the second member of the pair
            // values[j] - j is the value of the second member of the pair
            maxScore = max(maxScore, values[j] - j + maxIPlusValue);

            // Update the maxIPlusValue for the next iterations (values[i] + i)
            // since we need to consider this element as the potential first member of the pair
            maxIPlusValue = max(maxIPlusValue, values[j] + j);
        }

        // Return the maximum score found amongst all sightseeing pairs
        return maxScore;
    }
};

```

Typescript Solution

```typescript
/**
 * Calculates the maximum score of a sightseeing pair among all possible pairs.
 * The score of a pair (i, j) is defined as values[i] + values[j] + i - j.
 *
 * @param values - An array representing the values of each sightseeing spot.
 * @returns The maximum score of a sightseeing pair.
 */
function maxScoreSightseeingPair(values: number[]): number {
    // Initialize the answer to zero
    let maxScore = 0;

    // Initialize `maxValueWithIndex` to the value of the first spot,
    // since it will be used to keep track of the best potential score for all previous spots
    let maxValueWithIndex = values[0];

    // Iterate over all the possible sightseeing spots, starting from the second one
    for (let currentIndex = 1; currentIndex < values.length; ++currentIndex) {
        // Calculate the current pair score considering the current index and the max value identified so far
        maxScore = Math.max(maxScore, values[currentIndex] - currentIndex + maxValueWithIndex);

        // Update `maxValueWithIndex` if the current spot, with the addition of its index, is greater than the current `maxValueWithI
        maxValueWithIndex = Math.max(maxValueWithIndex, values[currentIndex] + currentIndex);
    }

    // Return the highest score found among all pairs
    return maxScore;
}

```

Time and Space Complexity

The given code defines a method to find the maximum score of a sightseeing pair where the score is defined by the sum of the values of the pair reduced by the distance between them (i.e., `values[i] + values[j] + i - j` for a pair `(i, j)`).

Time Complexity

The time complexity of the code is `O(n)`, where `n` is the length of the input list `values`.

This is because there is a single for-loop that goes through the array `values` from the second element to the last, doing constant time operations within the loop such as computing the maximum of `ans` and updating the value of `mx`. There are no nested loops or other operations that would increase the time complexity beyond linear time.

Space Complexity

The space complexity of the code is `O(1)`.

There are only a few variables used (`ans`, `mx`, and `j`) and their memory consumption does not depend on the input size, which means that there is a constant amount of extra space used regardless of the size of `values`.