1131. Maximum of Absolute Value Expression

Medium Array Math Leetcode Link

Problem Description

Given two integer arrays `arr1` and `arr2` of the same length, the task is to calculate the maximum value of a specific expression for all pairs of indices `(i, j)` where `0 <= i, j < arr1.length`. The expression to maximize is:

```
|arr1[i] - arr1[j]| + |arr2[i] - arr2[j]| + |i - j|
```

The `| |` denotes the absolute value, and the goal is to find the maximum result possible by choosing different values of `i` and `j`.

Intuition

The intuitive approach to this problem would be to consider all possible pairs of indices `(i, j)` and calculate the value of the expression for each pair, but this would lead to a rather inefficient solution with a quadratic runtime complexity. To optimize this, one must find a smarter way to ascertain the maximum value without directly examining every pair.

The important realization is that the expression can be rewritten and analyzed in terms of its components. Notice that the expression comprises three terms: the absolute difference between `arr1` elements, the absolute difference between `arr2` elements, and the absolute difference between indices.

We can reformulate each term like this:

```
|arr1[i] - arr1[j]| can be either (arr1[i] - arr1[j]) or -(arr1[i] - arr1[j])
|arr2[i] - arr2[j]| can be either (arr2[i] - arr2[j]) or -(arr2[i] - arr2[j])
|i - j| can be either (i - j) or -(i - j)
```

Each of these can have two possible signs (+ or -), which gives us a total of 2 * 2 * 2 = 8 combinations. Each combination can be thought of representing a different "direction" or case. However, for this solution, we take into account that `|i - j|` is always added to the expression, thus we only need to consider four combinations for the terms from the arrays `arr1` and `arr2`.

The code uses a `pairwise` function in combination with a `dirs` tuple to go through these four combinations (-1 and 1 represent the possible signs). For each case, it initializes minimum (`mi`) and maximum (`mx`) values that could be yielded by the expression when evaluating for all indices `(i, j)`. At each step in the loop, the code updates the `mx` and `mi` to find the maximum possible value for the current combination and eventually computes the maximum possible value for the overall expression by comparing it against the maximum (`ans`) of previous cases.

Thus, instead of comparing all pairs of `(i, j)`, we keep track of the max and min value of `a * arr1[i] + b * arr2[i] + i` for each direction and use these values to calculate and update the running maximum value over the entire array.

Solution Approach

The solution utilizes the fact that every absolute difference between two numbers can be represented as either a positive or a negative difference. Therefore, for each element in the arrays, we can apply either a positive or a negative sign, leading to different cases. The goal is to calculate the maximum value of our expression within each of these cases.

Since we have two arrays and we need to consider positive and negative contributions of their elements independently, we need to evaluate four cases. The `dirs` tuple `(1, -1, -1, 1)` along with `pairwise` is used to iterate over these cases. The `pairwise` function is not built into Python, but it would simply return pairs of elements from `dirs`, e.g., `(1, -1), (-1, -1), (-1, 1)`. This means we evaluate the following cases for each index `i`:

1. `arr1[i] - arr2[i] + i`
2. `-arr1[i] - arr2[i] + i`
3. `-arr1[i] + arr2[i] + i`
4. `arr1[i] + arr2[i] + i`

For each of the four cases, we use a loop to iterate over all indices `i` of our input arrays.

Within each loop iteration, we apply the current case's signs to elements of `arr1[i]` and `arr2[i]` and add the index `i`. We update `mx` to be the maximum value seen so far and `mi` to be the minimum value seen so far.

```
mx = max(mx, a * arr1[i] + b * arr2[i] + i)
mi = min(mi, a * arr1[i] + b * arr2[i] + i)
```

The maximum value for this case is then `mx - mi`, which represents the widest range we've seen between the expression values for any pair of indices within this particular case.

`ans = max(ans, mx - mi)`

This line updates the overall maximum value `ans` with the maximum value obtained in the current case. By the end of the for-loop that iterates over the `pairwise` elements, we've considered all possible signs for differences and added the absolute difference of indices, yielding the maximum value of the original expression.

Finally, we return `ans`, which represents the maximum value of the expression across all pairs `(i, j)`.

Example Walkthrough

Let's illustrate the solution approach with an example. Suppose we have the following arrays:

```
arr1 = [1, 2, 3]
arr2 = [4, 5, 6]
```

For these arrays, we want to maximize the expression:

```
|arr1[i] - arr1[j]| + |arr2[i] - arr2[j]| + |i - j|
```

First, let's manually calculate the expression for some pairs to understand the problem:

- For `i = 0` and `j = 1`:
  - The expression would be `|1 - 2| + |4 - 5| + |0 - 1| = 1 + 1 + 1 = 3`
- For `i = 0` and `j = 2`:
  - The expression would be `|1 - 3| + |4 - 6| + |0 - 2| = 2 + 2 + 2 = 6`

We notice that as `i` and `j` get farther apart, the expression tends to increase because of the `|i - j|` term, but the first two absolute terms can also impact the result.

Let's now use the optimized approach described in the solution:

1. We will consider four cases by applying the `pairwise` concept to the `dirs` tuple.
   - We don't actually need the `pairwise` function here because we know our cases in advance:

```
Case 1: arr1[i] - arr2[i] + i
Case 2: -arr1[i] - arr2[i] + i
Case 3: -arr1[i] + arr2[i] + i
Case 4: arr1[i] + arr2[i] + i
```

2. We will iterate over each element `i` of `arr1` and `arr2` and calculate the expressions for each of the four cases, updating the maximum (`mx`) and minimum (`mi`) as we go. We then use these to compute the range `mx - mi` in each case which is a potential maximum for our final answer.
3. Let's start with Case 1 (`arr1[i] - arr2[i] + i`)
   - For `i = 0`: `1 - 4 + 0 = -3`
   - For `i = 1`: `2 - 5 + 1 = -2`
   - For `i = 2`: `3 - 6 + 2 = -1`
   - `mx` of Case 1 is `-1`, `mi` is `-3`, and the range `mx - mi` is `-1 - (-3) = 2`
4. Next is Case 2 (`-arr1[i] - arr2[i] + i`)
   - For `i = 0`: `-1 - 4 + 0 = -5`
   - For `i = 1`: `-2 - 5 + 1 = -6`
   - For `i = 2`: `-3 - 6 + 2 = -7`
   - `mx` of Case 2 is `-5`, `mi` is `-7`, the range `mx - mi` is `-5 - (-7) = 2`
5. Proceed similarly for Case 3 and Case 4, updating `mx` and `mi` for each case, then compute the range `mx - mi` to see if the result is higher than the previous cases.
6. After evaluating all four cases, the final answer `ans` would be the maximum range `mx - mi` found among all cases.

For our example, the ranges were the same (`2`) for Case 1 and Case 2, and if we calculate it for Cases 3 and 4, we would get `10` and `8` respectively. Hence, the maximum value for the expression `|arr1[i] - arr1[j]| + |arr2[i] - arr2[j]| + |i - j|` across all `i, j` with `0 <= i, j < arr1.length` would be `10`.

This example shows how to compute the maximum value effectively without checking each combination of `i` and `j` individually. By considering the varied outcomes of applying both the positive and negative signs, we manage to find the optimal result with a more efficient approach.

Python Solution

```python
from itertools import product
from math import inf

class Solution:
    def maxAbsValExpr(self, arr1, arr2):
        # Initialize maximum absolute value expression to a very small number
        max_absolute_value_expr = -inf

        # Iterate over all combinations of directions for arr1 and arr2.
        # Each combination represents multiplying arr1 and arr2 with
        # 1 or -1, effectively checking all possible cases.
        for dir1, dir2 in product((1, -1), repeat=2):
            # Initialize maximum and minimum values observed
            # for the current direction combination
            max_val, min_val = -inf, inf

            # Iterate over the pairs (i, (x, y)) where i is the index,
            # x is the element from arr1, and y is the element from arr2
            for i, (x, y) in enumerate(zip(arr1, arr2)):
                # Calculate the current value using the expression given by the problem
                current_value = dir1 * x + dir2 * y + i

                # Update the maximum and minimum observed values
                max_val = max(max_val, current_value)
                min_val = min(min_val, current_value)

            # Update the maximum absolute value expression with the
            # maximum difference between observed values
            max_absolute_value_expr = max(max_absolute_value_expr, max_val - min_val)

        # Return the maximum absolute value expression found
        return max_absolute_value_expr

```

Java Solution

```java
class Solution {
    public int maxAbsValExpr(int[] arr1, int[] arr2) {
        // Define the multipliers to represent the four possible combinations
        // of adding or subtracting arr1[i] and arr2[i].
        // These multipliers will be applied inside the loop below.
        int[] multipliers = {1, -1, -1, 1};

        // Initialize variable 'maxDifference' to keep track of the maximum absolute value
        // expression found. This value will be returned at the end.
        // Using Integer.MIN_VALUE to handle negative overflow edge case.
        int maxDifference = Integer.MIN_VALUE;

        // Length of the given arrays, assuming both arrays have the same length.
        int n = arr1.length;

        // Iterate over the four possible combinations of expressions represented by multipliers.
        for (int k = 0; k < 4; ++k) {
            // Variables representing the direction for arr1[i] and arr2[i]
            // in the expression based on the current multiplier.
            int a = multipliers[k], b = multipliers[k + 1];

            // Initialize 'maxValue' and 'minValue' to track the maximum and minimum values
            // of the expression for a given set of multipliers.
            // Using Integer.MIN_VALUE and Integer.MAX_VALUE to start with the extreme possible values.
            int maxValue = Integer.MIN_VALUE, minValue = Integer.MAX_VALUE;

            // Iterate through the elements of the arrays to calculate the expressions.
            for (int i = 0; i < n; ++i) {
                // Calculate the current value of the expression.
                int currentValue = a * arr1[i] + b * arr2[i] + i;

                // Update 'maxValue' and 'minValue' if the current value is greater than 'maxValue'
                // or less than 'minValue' respectively.
                maxValue = Math.max(maxValue, currentValue);
                minValue = Math.min(minValue, currentValue);

                // Update 'maxDifference' with the maximum difference found so far.
                maxDifference = Math.max(maxDifference, maxValue - minValue);
            }
        }

        // Return the maximum absolute value expression found.
        return maxDifference;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>
using namespace std;

class Solution {
public:
    int maxAbsValExpr(vector<int>& arr1, vector<int>& arr2) {
        // Directions are the coefficients for arr1 and arr2 in the combinations
        // Last element is duplicated for easy loop termination
        int directions[5] = {1, -1, -1, 1, 1};
        const int INF = 1 << 30; // Define infinity as a large number
        int maxAnswer = -INF; // Initialize the maximum answer to negative infinity
        int arraySize = arr1.size(); // Get the size of the input arrays

        // Iterate over the 4 combinations of signs in the expression
        for (int k = 0; k < 4; ++k) {
            int coefficientA = directions[k], // Coefficient for arr1
                coefficientB = directions[k + 1]; // Coefficient for arr2
            int maxExprValue = -INF, // Initialize max expression value in the current loop
                minExprValue = INF;  // Initialize min expression value in the current loop

            // Loop through each element in the array to find max and min of expression
            for (int i = 0; i < arraySize; ++i) {
                // Find the maximum value expression with the current coefficients
                maxExprValue = max(maxExprValue, coefficientA * arr1[i] + coefficientB * arr2[i] + i);
                // Find the minimum value expression with the current coefficients
                minExprValue = min(minExprValue, coefficientA * arr1[i] + coefficientB * arr2[i] + i);
                // Update the overall maximum answer with the difference between max and min
                maxAnswer = max(maxAnswer, maxExprValue - minExprValue);
            }
        }
        return maxAnswer; // Return the found maximum absolute value of an expression
    }
};

```

Typescript Solution

```typescript
// Calculates the maximum absolute value expression for two arrays
function maxAbsValExpr(arr1: number[], arr2: number[]): number {
    // Define multipliers for different expression scenarios
    const multipliers = [1, -1, -1, 1, 1];

    // Initialize the maximum answer as the smallest integer possible
    let maxAnswer = Number.MIN_SAFE_INTEGER;

    // Iterate through all possible expressions based on multipliers
    for (let expIndex = 0; expIndex < 4; ++expIndex) {
        // Select multipliers for current expression
        const coeffA = multipliers[expIndex];
        const coeffB = multipliers[expIndex + 1];

        // Initialize max and min variables for current expression scenario
        let maxCurrent = Number.MIN_SAFE_INTEGER;
        let minCurrent = Number.MAX_SAFE_INTEGER;

        // Iterate through elements of the arrays to compute expressions
        for (let i = 0; i < arr1.length; ++i) {
            const val1 = arr1[i];
            const val2 = arr2[i];

            // Calculate the expression value with current i
            const expression = coeffA * val1 + coeffB * val2 + i;

            // Update the current maximum and minimum
            maxCurrent = Math.max(maxCurrent, expression);
            minCurrent = Math.min(minCurrent, expression);

            // Update the global maximum answer
            maxAnswer = Math.max(maxAnswer, maxCurrent - minCurrent);
        }
    }

    // Return the calculated maximum absolute value expression
    return maxAnswer;
}

```

Time and Space Complexity

The given Python code calculates the maximum absolute value expression for two arrays `arr1` and `arr2`.

- **Time Complexity:**
  - There are 4 pairs of `(a, b)` which correspond to each combination of `{1, -1}` for each array element.
  - For each of these 4 pairs, we iterate through both `arr1` and `arr2` simultaneously, using `enumerate(zip(arr1, arr2))`, which runs for `n` iterations where `n` is the length of the input arrays.
  - Inside the loop, we perform constant-time operations such as comparisons and arithmetic operations.
  - Therefore, the time complexity is `O(4n)` which simplifies to `O(n)`.
- **Space Complexity:**
  - The additional space used by the algorithm is constant. It only needs a few variables for tracking maximum values, minimum values, and indices (`mx`, `mi`, and `i`), regardless of the input size.
  - As such, the space complexity is `O(1)`.