2809. Minimum Time to Make Array Sum At Most x

Hard Array Dynamic Programming Sorting Leetcode Link

Problem Description

Given two arrays `nums1` and `nums2` that are both 0-indexed and of the same length, the task is to find the minimum time it takes to reduce the sum of all elements in `nums1` to a value less than or equal to a given integer `x`. Every second, each element `nums1[i]` is increased by its corresponding element `nums2[i]`. Once this increment has happened, you have the choice to reset any `nums1[i]` to 0 exactly once. If you cannot make the sum of `nums1` less than or equal to `x`, return -1.

Intuition

To solve this problem, the intuition is to determine for each second what the best outcome would be if we reset one of the `nums1` elements to 0 at that exact moment. The outcome to maximize is the sum of all `nums1` elements plus all the increments that have been made up to that second.

To approach this solution:

1. The `nums1` and `nums2` items are first zipped together and sorted by the `nums2` values since they indicate how much each corresponding `nums1` value grows per second. We're interested in maximizing the rate of growth per second, which is why sorting based on `nums2` is sensible.

2. A dynamic programming list `f` is prepared which is initialized to 0 and has a length of `n + 1`, where `n` is the length of the arrays. This list will help keep track of the best outcomes at each second for the corresponding prefix of the array.

3. The main idea applied here is, for a current moment denoted by `j`, we want to determine whether it would be beneficial to perform the reset operation now or to have done it one second earlier. This is computed by three values:
   - The value if we choose not to reset at this moment (`f[j]`).
   - The new sum we would get if we reset `f[j-1]` one second earlier and added the current `a + (b * j)`, which denotes current incremented value plus its growth over `j` seconds.
4. This leads to the updating of `f[j]` to the maximum of these options.

5. Finally, loop through each possible second and calculate if the total growth plus the sum of elements is less than or equal to `x` after resetting the best element at that second. If it is, return the current second as the minimum time. If no such second is found, return -1.

This solution tries to balance the gain from allowing the `nums1` values to grow and the benefit of resetting a value to zero at the right time to minimize the sum to meet the condition with respect to `x`.

Solution Approach

The solution uses a combination of sorting, dynamic programming, and greedy strategy.

1. **Sorting**: First off, the `zip` function combines the elements from `nums1` and `nums2` into pairs, and then sorts them by the second value (`nums2[i]`), which represents the growth rate or increment per second. This is done using the `sorted` function with the key being the second element of the pair (`z[1]`).

   By sorting, we prioritize the handling of the elements in `nums1` which are paired with the largest increment factors first, allowing us to benefit from their rapid growth when considering when to reset them to zero.

2. **Dynamic Programming (DP)**: A dynamic programming array `f` is initialized as a list of zeros, with a size of `n+1`. Here, `n` is the length of `nums1`. The dynamic programming array `f[j]` represents the maximum sum that can be achieved by applying the reset operation `j` times, up until the current point of our iteration over the sorted pairs.

3. **Greedy Iteration**: After sorting, a loop is started which iterates over each paired element from the sorted list. For each such element, an inner loop runs in reverse over the range from `n` down to `1`. The reverse iteration allows us to consider the scenarios of having consumed one less reset opportunity at each stage.

   In the inner loop, `a` represents the current value of `nums1[i]`, and `b` is the increment value of `nums2[i]`. The inner loop updates the max sum for `j` resets by comparing the current `f[j]` and the sum of `f[j-1]` (the max sum if the reset was done one second earlier) plus the value of the current index (after having been incremented for `j` seconds).

   The condition `f[j] = max(f[j], f[j - 1] + a + b * j)` ensures that for each possible number of resets `j`, we track the best sum possible at that moment.

4. **Calculate Required Seconds**: After the dynamic programming step, two sums are calculated: the sum of elements in `nums1` (`s1`) and the sum of chosen increments in `nums2` (`s2`). The final check loops through `j` from `0` to `n`, checking if the condition `(s1 + s2 * j - f[j]) <= x` is met. This is checking if the total growth (assuming maximum growth for each second) minus the maximum sum when `j` resets have been used is less than or equal to `x`. If this is the case, it indicates that the sum of `nums1` can be maintained below or equal to `x` in `j` seconds after resetting the most beneficial elements, and `j` is returned as the solution.

   If the loop completes without finding a satisfactory `j`, then the function returns `-1`, indicating that it's not possible to reduce the sum of `nums1` below or equal to `x`.

The solution capitalizes on the greedy strategy of sorting by growth rate to consider the most impactful elements first and uses dynamic programming to keep track of the best reset decisions as the array elements grow.

Example Walkthrough

Let's consider a small example to illustrate the solution approach.

Let `nums1 = [4, 3, 8]`, `nums2 = [2, 1, 3]`, and `x = 13`.

Following the solution approach:

1. The two lists are `zip`ped and sorted by growth values (`nums2`). The resulting list is `[(3, 1), (4, 2), (8, 3)]`.

2. An array `f` is initialized with zeros, of length 4 since `n = 3`. `f = [0, 0, 0, 0]`.

3. Iterating over each element with initialized sums `s1 = 15` (sum of `nums1`) and `s2 = 6` (sum of `nums2`):

   First iteration (element (3, 1)):

   - Inner loop for `j = 1` to `n`, update `f[j]`:
     - `j = 1`: `f[1]` is updated to max(`f[1]`, `f[0] + 3 + 1 * 1`) → `f[1] = 4`.
     - `j = 2`: `f[2]` is updated to max(`f[2]`, `f[1] + 3 + 1 * 2`) → `f[2] = 7`.
     - `j = 3`: `f[3]` is updated to max(`f[3]`, `f[2] + 3 + 1 * 3`) → `f[3] = 10`.

   Second iteration (element (4, 2)):

   - Inner loop:
     - `j = 1`: `f[1]` is updated to max(`f[1]`, `f[0] + 4 + 2 * 1`) → `f[1]` remains `4`.
     - `j = 2`: `f[2]` is updated to max(`f[2]`, `f[1] + 4 + 2 * 2`) → `f[2]` remains `7`.
     - `j = 3`: `f[3]` is updated to max(`f[3]`, `f[2] + 4 + 2 * 3`) → `f[3]` becomes `13`.

   Third iteration (element (8, 3)):

   - Inner loop:
     - `j = 1`: `f[1]` is updated to max(`f[1]`, `f[0] + 8 + 3 * 1`) → `f[1]` becomes `11`.
     - `j = 2`: `f[2]` is updated to max(`f[2]`, `f[1] + 8 + 3 * 2`) → `f[2]` becomes `19`.
     - `j = 3`: `f[3]` is updated to max(`f[3]`, `f[2] + 8 + 3 * 3`) → `f[3]` becomes `28`.

   Now we have `f = [0, 11, 19, 28]`.

4. Finally, loop to find the minimum seconds required:

   Checking for `j` from `0` to `n`:

   - `j = 0`: `(s1 + s2 * 0 - f[0])` → `(15 + 6 * 0 - 0)` = 15 which is greater than `x`.
   - `j = 1`: `(s1 + s2 * 1 - f[1])` → `(15 + 6 * 1 - 11)` = 10 which is less than `x`. So the minimum time needed is 1 second.

In this case, we find that `j = 1` is the point where, after resetting the most significant element to zero, the sum of `nums1` will be less than or equal to `x`. Hence, the answer is 1 second.

Python Solution

```python
from typing import List

class Solution:
    def minimumTime(self, nums1: List[int], nums2: List[int], limit: int) -> int:
        # Determine the number of pairs in nums1 and nums2
        num_pairs = len(nums1)

        # Initialize an array to store the maximum value possible with a given number of pairs used
        max_values = [0] * (num_pairs + 1)

        # Sort the pairs based on the second element of each pair
        sorted_pairs = sorted(zip(nums1, nums2), key=lambda pair: pair[1])

        # Calculate the maximum value for using each possible number of pairs
        for elem1, elem2 in sorted_pairs:
            for j in range(num_pairs, 0, -1):
                max_values[j] = max(max_values[j], max_values[j - 1] + elem1 + elem2 * j)

        # Calculate the sum of elements in nums1 and nums2
        sum_nums1 = sum(nums1)
        sum_nums2 = sum(nums2)

        # Determine the minimum number of  operations needed
        # to bring the sum of warped nums1 and nums2 below 'limit'
        for j in range(num_pairs + 1):
            if sum_nums1 + sum_nums2 * j - max_values[j] <= limit:
                return j

        # If no solution is found, return -1
        return -1

# Example usage:
# solution = Solution()
# result = solution.minimumTime(nums1=[1,2,3], nums2=[3,2,1], limit=50)
# print(result) # Output will depend on the input values

```

Java Solution

```java
import java.util.*;

class Solution {
    public int minimumTime(List<Integer> nums1, List<Integer> nums2, int x) {
        int n = nums1.size(); // The size of the given lists
        int[] dp = new int[n + 1]; // Dynamic programming array to store maximum scores

        // Create an array to store the pairs from nums1 and nums2
        int[][] pairs = new int[n][2];
        for (int i = 0; i < n; ++i) {
            pairs[i] = new int[]{nums1.get(i), nums2.get(i)};
        }

        // Sort the pairs by the second element in ascending order
        Arrays.sort(pairs, Comparator.comparingInt(pair -> pair[1]));

        // Calculate the maximum score for each subsequence of pairs
        for (int[] pair : pairs) {
            int first = pair[0], second = pair[1];
            for (int j = n; j > 0; --j) {
                dp[j] = Math.max(dp[j], dp[j - 1] + first + second * j);
            }
        }

        int sumNums1 = 0, sumNums2 = 0; // Sum of elements from nums1 and nums2
        for (int v : nums1) {
            sumNums1 += v;
        }
        for (int v : nums2) {
            sumNums2 += v;
        }

        // Check for the minimal number of pairs needed to be chosen
        // to satisfy the condition sumNums1 + sumNums2 * j - dp[j] <= x
        for (int j = 0; j <= n; ++j) {
            if (sumNums1 + sumNums2 * j - dp[j] <= x) {
                return j; // Return the minimal number of pairs
            }
        }
        // If no such number of pairs is found, return -1
        return -1;
    }
}

```

C++ Solution

```cpp
#include <vector>      // Needed for using std::vector
#include <algorithm>   // Needed for std::sort and std::accumulate
#include <cstring>     // Needed for std::memset

class Solution {
public:
    int minimumTime(std::vector<int>& taskTimesA, std::vector<int>& taskTimesB, int maxTime) {
        int tasksCount = taskTimesA.size(); // Number of tasks
        // Pair each task's time from B with the corresponding time from A
        std::vector<std::pair<int, int>> taskPairs;
        for (int i = 0; i < tasksCount; ++i) {
            taskPairs.emplace_back(taskTimesB[i], taskTimesA[i]);
        }

        // Sort the pair array based on times from B task times (ascending)
        std::sort(taskPairs.begin(), taskPairs.end());

        // Initialize the dynamic programming array;
        // f[i] will hold the maximum time saved after completing i tasks
        std::vector<int> dp(tasksCount + 1, 0);

        // Calculate the maximum time saved for each number of tasks
        for (auto& [timeB, timeA] : taskPairs) {
            for (int j = tasksCount; j > 0; --j) {
                dp[j] = std::max(dp[j], dp[j - 1] + timeA + timeB * j);
            }
        }

        // Calculate the sum of the times for both A and B tasks
        int totalTimeA = accumulate(taskTimesA.begin(), taskTimesA.end(), 0);
        int totalTimeB = accumulate(taskTimesB.begin(), taskTimesB.end(), 0);

        // Find the minimum number of tasks needed such that the time limit is not exceeded
        for (int j = 0; j <= tasksCount; ++j) {
            if (totalTimeA + totalTimeB * j - dp[j] <= maxTime) {
                return j; // Minimum number of tasks
            }
        }

        // If no solution is found return -1
        return -1;
    }
};

```

Typescript Solution

```typescript
function minimumTime(nums1: number[], nums2: number[], x: number): number {
    const numPairs = nums1.length;
    const maxValues: number[] = new Array(numPairs + 1).fill(0);
    const sortedPairs: number[][] = [];

    // Combine corresponding elements of nums1 and nums2 into pairs
    for (let i = 0; i < numPairs; i++) {
        sortedPairs.push([nums1[i], nums2[i]]);
    }

    // Sort pairs based on the second element of each pair (nums2 value)
    sortedPairs.sort((pairA, pairB) => pairA[1] - pairB[1]);

    // Populate the maxValues array with maximum scores computed
    for (const [value1, value2] of sortedPairs) {
        for (let j = numPairs; j > 0; j--) {
            maxValues[j] = Math.max(
                maxValues[j],
                maxValues[j - 1] + value1 + value2 * j
            );
        }
    }

    // Calculate total sum of both arrays
    const totalSumNums1 = nums1.reduce((total, num) => total + num, 0);
    const totalSumNums2 = nums2.reduce((total, num) => total + num, 0);

    // Determine the minimum index 'j' where the condition is satisfied
    for (let j = 0; j <= numPairs; j++) {
        const conditionValue = totalSumNums1 + totalSumNums2 * j - maxValues[j];
        if (conditionValue <= x) {
            return j;
        }
    }

    // Return -1 if no such index was found
    return -1;
}

```

Time and Space Complexity

Time Complexity

Let's analyze the time complexity of the given code:

- There is a loop which sorts the `zip` of `nums1` and `nums2` by the second element of the tuple. The sort function in Python uses Timsort, which has a time complexity of `O(n log n)` where `n` is the number of elements being sorted.

- After sorting, there is a nested loop:

  - The outer loop iterates over the sorted list, which is `O(n)`.

  - The inner loop iterates backwards from `n` to `1`, which is also `O(n)` in the worst case.

  - Inside the inner loop, the code updates the list `f` with calculated values, which is an `O(1)` operation.

Combining all the above steps, the overall time complexity of the nested loops is `O(n^2)`.

- The final step of the function iterates through the range `n + 1`, which is `O(n)`.

Thus, the total time complexity is dominated by the sorting and the nested loops, giving us `O(n log n) + O(n^2)`, which simplifies to `O(n^2)` because `n^2` grows faster than `n log n`.

Space Complexity

For space complexity:

- A new list `f` of size `n + 1` is created, which gives us `O(n)`.

- Sorting the zipped lists creates an additional space which also leads to `O(n)`, despite the fact that Python's sort is typically in-place, because the `zip` object is being converted to a list and sorted separately.

- Other variables used are of constant space and do not scale with `n`.

Consequently, the total space complexity is `O(n)`, because we only account for the largest term when calculating space complexity.