2552. Count Increasing Quadruplets

Hard | Binary Indexed Tree | Array | Dynamic Programming | Enumeration | Prefix Sum | Leetcode Link

Problem Description

The problem asks us to find the number of increasing quadruplets in an array `nums` of size `n`. A quadruplet (i, j, k, l) is considered increasing if:

1. The indexes follow the relationship `0 <= i < j < k < l < n`
2. The values at those indices follow the relationship `nums[i] < nums[k] < nums[j] < nums[l]`

Basically, we need to count how many combinations of four different indices `(i, j, k, l)` in the array satisfy the conditions where the elements are strictly increasing with respect to those indices.

Intuition

To solve this problem, the intuitive approach would be to try every possible quadruplet and check if it satisfies the condition. However, this would result in an inefficient solution with a time complexity of O(n^4) which is not practical for large arrays.

Observing the problem, we can cleverly reduce the problem's complexity by dividing it into subproblems:

1. For each pair `(j, k)` where `j < k`, count how many `l` (where `l > k`) exist such that `nums[j] < nums[l]`. Store this count since it will be common for all `i` that come before `j`. We'll call this array `f`.
2. Similarly, for each pair `(j, k)` where `j < k`, count how many `i` (where `i < j`) exist such that `nums[i] < nums[k]`. We need this to avoid recounting for different `i` with the same `j` and `k`. We'll call this array `g`.

Now, for each pair `(j, k)`, the number of valid quadruplets with indices `(i, j, k, l)` is `f[j][k] * g[j][k]`. By summing these products for all pairs `(j, k)`, we get the total count of increasing quadruplets.

This approach significantly reduces the time complexity to O(n^3) as we perform three nested loops to count elements and use additional arrays `f` and `g` to store intermediate counts and avoid recomputation.

Solution Approach

The solution implements a dynamic programming approach to solve the problem efficiently. Two auxiliary matrices `f` and `g` are used to store the counts of valid `l` indices for the second half of the quadruplet and valid `i` indices for the first half of the quadruplet. The solution involves three main steps:

1. Counting the number of `l` elements for each `(j, k)` pair where `j < k`, such that `nums[j] < nums[l]`. This count is stored in the `f[j][k]` matrix. To avoid recomputation, we iterate backward from the penultimate index down to `1` and update the count dynamically.

2. Counting the number of `i` elements for each `(j, k)` pair where `j < k`, such that `nums[i] < nums[k]`. This count is stored in the `g[j][k]` matrix. Similarly, we iterate through the array and update the count as we move.

3. Finally, for each `(j, k)` pair found in the previous steps, we multiply the values at `f[j][k]` and `g[j][k]` to get the number of valid quadruplets with indexes `(i, j, k, l)`. We sum these products to derive the total count.

The code uses three nested loops: The outer two are for iterating over all possible `(j, k)` pairs, and the inner ones are for calculating the counts that get stored in `f` and `g`. It's notable that relying on previously computed values and only updating them when necessary is a hallmark of dynamic programming which reduces time complexity.

Here's a more detailed step-by-step breakdown of what each part of the code is doing:

- We initialize the matrices `f` and `g` as nxn matrices filled with 0s.
- We iterate over variable `j` from 1 to `n-3` for the first loop.
  - We calculate the count of `l` values greater than `nums[j]` using a list comprehension and store it in `cnt`.
  - Then we iterate over `k` from `j+1` to `n-2` and populate the `f[j][k]` matrix using `cnt`. If `nums[j] > nums[k]`, the count remains unchanged; else, we decrease `cnt` by 1 as that `k` is not valid.
- We iterate over variable `k` from 2 to `n-2` for the second loop.
  - We calculate the count of `i` values smaller than `nums[k]` using a list comprehension and store in `cnt`.
  - Then we iterate backward over `j` from `k-1` to 1, populating the `g[j][k]` matrix using `cnt`. If `nums[j] > nums[k]`, the count remains unchanged; else, we decrease `cnt` by 1 for the same reason as above.
- We calculate the sum of products of corresponding elements in `f` and `g` to get the final count of increasing quadruplets.

By storing these intermediate results, the algorithm effectively avoids repetitive calculations and achieves a lower time complexity than the naive approach would permit.

Example Walkthrough

Let's consider a small example array `nums` with elements [3, 1, 4, 2, 5].

Following the solution approach to count the number of increasing quadruplets:

1. **Count number of `l` elements for each `(j, k)` pair**:
   - Starting from `j = 1` (the second element):
     - We have `(j, k)` as (1, 2) i.e., elements `1` and `4`. There's one element greater than 4 in the remaining list `[2, 5]`, which is `5`. So, `f[1][2] = 1`.
     - For `(j, k)` as (1, 3), there are no elements greater than `2` in our remaining list `[5]`, so `f[1][3] = 0`.
   - Next with `j = 2` (the third element):
     - We only have one `(j, k)` pair (2, 3) since `k` should be greater than `j`. There is one element greater than `2` which is `5`, so `f[2][3] = 1`.
2. **Count number of `i` elements for each `(j, k)` pair**:
   - Starting from `k = 2` (the third element):
     - Consider `i` values less than `4` when `k = 2`. We have one `i` at index 1 (element `3`), so `g[1][2] = 1`.
   - Next with `k = 3` (the fourth element):
     - The values less than `2` when `k = 3` are `3` and `1` from indices 0 and 1, respectively. Thus, `g[0][3] = 1` and `g[1][3] = 1`.
3. **Calculate the total count of increasing quadruplets**:
   - We only consider pairs where `j < k`, obtaining the products of corresponding `f[j][k]` and `g[j][k]`.
   - For `(j, k)` being (1, 2), we multiply `f[1][2]` and `g[1][2]` which gives us `1 * 1 = 1`.
   - For `(j, k)` being (1, 3), the multiplication is `f[1][3] * g[1][3] = 0 * 1 = 0`.
   - For `(j, k)` being (2, 3), the multiplication is `f[2][3] * g[0][3] = 1 * 1 = 1`.

By summing these up, we have `1 + 0 + 1 = 2` increasing quadruplets in the array.

As illustrated in this example, we utilized dynamic programming to systematically break down and solve the problem of finding all increasing quadruplets in an array, rather than naively comparing all possible combinations. Thanks to the intermediate storage of the count of `l` and `i` indices, we significantly reduce unnecessary recalculations and optimize the entire process.

Python Solution

```python
from typing import List

class Solution:
    def countQuadruplets(self, nums: List[int]) -> int:
        # Determine the number of elements in the list.
        length = len(nums)

        # Initialize two matrices to store frequency counts.
        freq_greater = [[0] * length for _ in range(length)]
        freq_smaller = [[0] * length for _ in range(length)]

        # Calculate the frequencies where nums[j] is greater than nums[k].
        for j in range(1, length - 2):
            # Count numbers greater than nums[j] starting from j+1
            count_greater = sum(nums[elem] > nums[j] for elem in range(j + 1, length))
            for k in range(j + 1, length - 1):
                # If nums[j] is greater than nums[k], store the count.
                if nums[j] > nums[k]:
                    freq_greater[j][k] = count_greater
                # Otherwise, decrease the count as we move to the next k.
                else:
                    count_greater -= 1

        # Calculate the frequencies where nums[j] is smaller than nums[k].
        for k in range(2, length - 1):
            # Count numbers smaller than nums[k] up to index k
            count_smaller = sum(nums[i] < nums[k] for i in range(k))
            for j in range(k - 1, 0, -1):
                # If nums[j] is greater than nums[k], store the count.
                if nums[j] > nums[k]:
                    freq_smaller[j][k] = count_smaller
                # Otherwise, decrease the count as we move to the next j.
                else:
                    count_smaller -= 1

        # Calculate the total number of quadruplets by multiplying the corresponding frequencies.
        total_quadruplets = sum(
            freq_greater[j][k] * freq_smaller[j][k] for j in range(1, length - 2) for k in range(j + 1, length - 1)
        )

        return total_quadruplets

```

Java Solution

```java
class Solution {
    public long countQuadruplets(int[] nums) {
        int len = nums.length;
        // 'greaterThanCount' stores the number of elements greater than nums[j] after the index j.
        int[][] greaterThanCount = new int[len][len];

        // 'lessThanCount' stores the number of elements less than nums[k] before the index k.
        int[][] lessThanCount = new int[len][len];

        // Pre-compute the number of elements that are greater than the current element for future reference.
        for (int j = 1; j < len - 2; ++j) {
            int count = 0;
            for (int l = j + 1; l < len; ++l) {
                if (nums[l] > nums[j]) {
                    ++count;
                }
            }
            for (int k = j + 1; k < len - 1; ++k) {
                if (nums[j] > nums[k]) {
                    greaterThanCount[j][k] = count;
                } else {
                    --count;
                }
            }
        }

        long quadrupletsCount = 0;
        for (int k = 2; k < len - 1; ++k) {
            int count = 0;
            // Count elements less than nums[k] from the start of the array to the index k.
            for (int i = 0; i < k; ++i) {
                if (nums[i] < nums[k]) {
                    ++count;
                }
            }
            for (int j = k - 1; j > 0; --j) {
                if (nums[j] > nums[k]) {
                    lessThanCount[j][k] = count;
                    // Multiply the counts of valid numbers from both sides of 'k,'
                    // to get the number of valid quadruplets that include 'nums[k]' as the third member.
                    quadrupletsCount += (long) greaterThanCount[j][k] * lessThanCount[j][k];
                } else {
                    --count;
                }
            }
        }
        return quadrupletsCount;
    }
}

```

C++ Solution

```cpp
const int MAX_SIZE = 4001;
int forwardCount[MAX_SIZE][MAX_SIZE];
int backwardCount[MAX_SIZE][MAX_SIZE];

class Solution {
public:
    long long countQuadruplets(vector<int>& nums) {
        // Initialize variables.
        int n = nums.size();
        // Reset forward and backward count matrices.
        memset(forwardCount, 0, sizeof forwardCount);
        memset(backwardCount, 0, sizeof backwardCount);

        // Loop to calculate forward counts.
        for (int j = 1; j < n - 2; ++j) {
            int count = 0; // Initialize counter for the number larger than nums[j]
            // Count numbers larger than nums[j] after index j
            for (int l = j + 1; l < n; ++l) {
                if (nums[l] > nums[j]) {
                    ++count; // Increase count if the condition is true
                }
            }
            // Calculate forward count for pairs (j, k)
            for (int k = j + 1; k < n - 1; ++k) {
                if (nums[j] > nums[k]) {
                    forwardCount[j][k] = count;
                } else {
                    --count; // Decrease the count if nums[j] <= nums[k]
                }
            }
        }

        long long ans = 0; // Initialize answer
        // Loop to calculate backward counts and the answer
        for (int k = 2; k < n - 1; ++k) {
            int count = 0; // Initialize counter for the number smaller than nums[k]
            // Count numbers smaller than nums[k] before index k
            for (int i = 0; i < k; ++i) {
                if (nums[i] < nums[k]) {
                    ++count; // Increase count if the condition is true
                }
            }
            // Calculate backward count for pairs (j, k) and update ans
            for (int j = k - 1; j > 0; --j) {
                if (nums[j] > nums[k]) {
                    backwardCount[j][k] = count;
                    // Update the final answer: the number of quadruplets where j < k
                    ans += 1ll * forwardCount[j][k] * backwardCount[j][k];
                } else {
                    --count; // Decrease the count if nums[j] <= nums[k]
                }
            }
        }
        // Return the final count of quadruplets
        return ans;
    }
};

```

Typescript Solution

```typescript
const MAX_SIZE = 4001;
let forwardCount: number[][] = new Array(MAX_SIZE).fill(null).map(() => new Array(MAX_SIZE).fill(0));
let backwardCount: number[][] = new Array(MAX_SIZE).fill(null).map(() => new Array(MAX_SIZE).fill(0));

/**
 * Counts the number of quadruplets that satisfy the given condition.
 * @param nums An array of integers.
 * @returns The number of valid quadruplets.
 */
function countQuadruplets(nums: number[]): number {
    let n: number = nums.length;

    // Reset forward and backward count matrices.
    forwardCount = forwardCount.map(row => row.fill(0));
    backwardCount = backwardCount.map(row => row.fill(0));

    // Loop to calculate forward counts.
    for (let j = 1; j < n - 2; ++j) {
        let count: number = 0; // Counter for the number larger than nums[j].
        // Count numbers larger than nums[j] after index j.
        for (let l = j + 1; l < n; ++l) {
            if (nums[l] > nums[j]) {
                ++count; // Increase count if the condition is true.
            }
        }
        // Calculate forward count for pairs (j, k).
        for (let k = j + 1; k < n - 1; ++k) {
            if (nums[j] > nums[k]) {
                forwardCount[j][k] = count;
            } else {
                --count; // Decrease the count if nums[j] <= nums[k].
            }
        }
    }

    let ans: number = 0; // Initialize answer.
    // Loop to calculate backward counts and the answer.
    for (let k = 2; k < n - 1; ++k) {
        let count: number = 0; // Initialize counter for the number smaller than nums[k].
        // Count numbers smaller than nums[k] before index k.
        for (let i = 0; i < k; ++i) {
            if (nums[i] < nums[k]) {
                ++count; // Increase count if the condition is true.
            }
        }
        // Calculate backward count for pairs (j, k) and update ans.
        for (let j = k - 1; j > 0; --j) {
            if (nums[j] > nums[k]) {
                backwardCount[j][k] = count;
                // Update the final answer: the number of quadruplets where j < k.
                ans += forwardCount[j][k] * backwardCount[j][k];
            } else {
                --count; // Decrease the count if nums[j] <= nums[k].
            }
        }
    }
    // Return the final count of quadruplets.
    return ans;
}

```

Time and Space Complexity

Time Complexity

For the given `countQuadruplets` function, let's examine the time complexity step by step:

- The generation of the matrices `f` and `g` each require `O(n^2)` time since they involve creating two two-dimensional lists of size `n x n`.

- In the first loop (for `j`), the internal loop for counting (`sum(nums[l] > nums[j] ...)`) has a complexity of `O(n)` for each `j` iteration since it potentially scans all elements after index `j`. The nested loop for `k` runs within the range of `j+1` to `n-1`, making it also on average `O(n/2)` per `j` iteration. Therefore, the total time complexity of this block is `O(n^2 * n/2)` which simplifies to `O(n^3)`.

- The second loop (for `k`) is similar in structure but runs the counting in reverse order. It has an inner loop for counting (`sum(nums[i] < nums[k] ...)`) which also has `O(n)` complexity for each `k`, and the reversed `j` loop again has an average complexity of `O(n/2)` per `k`. This again leads to `O(n^3)` for this block.

- The final sum operation iterates over the elements of the `f` and `g` matrices, which has `O(n^2)` complexity since it involves a nested loop that traverses most of the matrix.

Adding all of these together, the dominant term is `O(n^3)` since `O(n^3)` is much larger than `O(n^2)` as `n` grows.

Thus, the total time complexity of the `countQuadruplets` method is `O(n^3)`.

Space Complexity

The space complexity is determined by the additional space needed aside from the input:

- Two matrices `f` and `g` each of size `n x n` are created, resulting in `O(n^2)` space for each. Since they exist simultaneously, this amounts to `O(2 * n^2)` space in total, which simplifies to `O(n^2)` space.

- There are no other significant space-consuming objects or recursive calls that use additional space.

Hence, the space complexity of the method `countQuadruplets` is `O(n^2)`.