2601. Prime Subtraction Operation

Medium Greedy Array Math Binary Search **Number Theory** Leetcode Link

Problem Description

You are provided with an array of integers, and the goal is to determine if you can transform this array into a strictly increasing array through a specific operation. The operation allows you to pick an index in the array that has not been picked before and then choose a prime number less than the value at that index. You subtract this prime number from the value at the chosen index. An array is considered strictly increasing if every element is larger than the element before it. You are allowed to perform this operation as many times as needed, as long as each index is only picked once. The main challenge is to find out whether it is possible to make the array strictly increasing through these operations and return `true` if it is possible, or `false` if it is not.

Intuition

The key insight to solving this problem is understanding that each non-increasing pair of elements in the array needs to be dealt with by subtracting a prime number in such a way as to make the current element less than the next. To accomplish this, we must consider the primes smaller than the current element.

Because we are ensuring strictly increasing order, we need to find primes that are just small enough to decrease the current number, but not so much that we can't maintain the strictly increasing property. We approach this by starting from the second-to-last element and moving backward, checking if the current number needs adjustment with respect to its subsequent number.

To optimize prime number finding and selection, we first pre-generate a list of all prime numbers smaller than the maximum number in the original array. We then use binary search to find the smallest prime number that we can subtract from the current number to make it smaller than the subsequent number.

The algorithm is designed to find the optimal prime for each element from back to front. If, at any point, we cannot find a suitable prime, we know it isn't possible to achieve a strictly increasing array, and we return `false`. If, after processing all elements, we haven't encountered this issue, we return `true`, as we've successfully transformed the array into a strictly increasing one.

Solution Approach

The implementation of the solution uses two main algorithms: prime number generation and binary search, and it employs the list data structure to store the prime numbers and to manipulate the `nums` array.

Prime Number Generation

The prime number generation part of the algorithm involves creating a list of prime numbers up to the largest element in the `nums` array. We initialize an empty list `p` to store the prime numbers. We then iterate through numbers from `2` to the maximum number found in `nums`. For each number `i`, we test whether it is divisible by any previously identified prime numbers in the `p` list. If `i` is divisible by any of these primes, it is not prime and we skip to the next number. If no divisors are found, `i` is prime and we append it to the `p` list.

Binary Search for Prime Subtraction

Once we have our list of prime numbers, we then iterate through the `nums` array in reverse order, starting from the second-to-last element. For each element `nums[i]`, we need to find a prime number `p[j]` that, when subtracted from `nums[i]`, would make `nums[i]` less than `nums[i+1]`. This ensures the strictly increasing property for these two elements.

We use `bisect_right` from Python's `bisect` module to perform a binary search for the proper prime number that can be subtracted. `bisect_right` finds the index `j` in the prime list such that all the primes up to index `j-1` are strictly less than `nums[i] - nums[i+1]`. If `j` is equal to the length of `p` or `p[j]` is greater than or equal to `nums[i]`, it indicates there is no suitable prime that can be subtracted from `nums[i]` without making it negative or non-increasing with respect to `nums[i+1]`. In this case, we return `false` because it is impossible to make the array strictly increasing.

After the loop, if we have successfully found primes for all required elements, we return `true`, indicating it is possible to form a strictly increasing array with the specified operations.

To summarize, this approach cleverly combines efficient prime generation with binary search to minimize the number of elements that need to be checked and to quickly find the smallest appropriate prime for each subtraction operation. This results in an efficient algorithm suitable for processing arrays to determine if they can be made strictly increasing.

Example Walkthrough

Let's illustrate the solution approach with a small example. Consider the array `nums = [10, 5, 15]`. We want to determine if we can subtract prime numbers from each element (only once per element) to make the array strictly increasing.

First, we generate the list of prime numbers up to the largest number in `nums`, which is 15. So our list of primes `p` up to 15 would be `[2, 3, 5, 7, 11, 13]`.

Following the solution approach:

1. We start from the second-to-last element in `nums`, which is 5 in this case.
2. We see that 5 is not less than 15 (the subsequent element), so we need to check if we can subtract a prime number from the element `[10]` to make it less than 5.
3. From the list `p`, we search for the smallest prime number that would make the element at index 0 smaller than 5 (the next element) when subtracted. Here, we can subtract 7 from 10 (10 - 7 = 3), and 3 is less than 5.
4. The array now looks like `[3, 5, 15]`.
5. Upon iteratively checking all elements, we find that no more operations are needed because the array is already strictly increasing.
6. Since we were able to apply the operations to the array to make it strictly increasing, the final return value would be `true`.

To summarize the example, we navigated through each element of the array against its subsequent element, using binary search to quickly find a suitable prime to subtract from `nums[i]` to achieve the strictly increasing property. This process demonstrated the algorithm's strategy of prime subtraction and how it can effectively transform an array while adhering to the constraints of the problem.

Python Solution

```python
from bisect import bisect_right
from typing import List

class Solution:
    def primeSubOperation(self, nums: List[int]) -> bool:
        # Generate a list of prime numbers less than the maximum element in nums
        primes = []
        max_num = max(nums) + 1  # set limit for sieve as max element in nums + 1
        for current in range(2, max_num):
            # Check divisibility by all previously found primes
            for prime in primes:
                if current % prime == 0:
                    # Not a prime, since divisible by a prime in the list
                    break
            else:  # Loop didn't break, so it's a prime number
                # Append the prime number to the list of primes
                primes.append(current)

        # Get the length of the input list
        num_length = len(nums)

        # Traverse the list from back to front to ensure nums is strictly increasing
        for i in range(num_length - 2, -1, -1):
            # If the current element is already less than the next element, continue
            if nums[i] < nums[i + 1]:
                continue

            # Find the right index to insert difference using bisect_right
            difference = nums[i] - nums[i + 1]
            j = bisect_right(primes, difference)

            # Check if we have a prime number less than the difference
            # If not, we cannot make nums strictly increasing thus return False
            if j == len(primes) or primes[j] >= nums[i]:
                return False

            # Substract the nearest smaller prime from the current element to maintain the condition nums[i] < nums[i + 1]
            nums[i] -= primes[j]

        # If all checks are passed, return True indicating nums can be made strictly increasing
        return True

```

Java Solution

```java
class Solution {

    // Main method to check if a prime subtraction operation can make the array strictly increasing.
    public boolean primeSubOperation(int[] nums) {
        // Generate a list of prime numbers up to 1000
        List<Integer> primes = new ArrayList<>();
        for (int num = 2; num <= 1000; ++num) {
            boolean isPrime = true;
            for (int prime : primes) {
                if (num % prime == 0) {
                    isPrime = false;
                    break;
                }
            }
            if (isPrime) {
                primes.add(num);
            }
        }

        int length = nums.length;

        // Iterate over the array from the end to start
        for (int i = length - 2; i >= 0; --i) {

            // Continue if the current number is less than the next (array is already increasing here)
            if (nums[i] < nums[i + 1]) {
                continue;
            }

            // Use binary search to find the smallest prime such that it's less than the difference
            int index = binarySearch(primes, nums[i] - nums[i + 1]);

            // If the prime is not found or greater than or equal to the current number, return false
            if (index == primes.size() || primes.get(index) >= nums[i]) {
                return false;
            }

            // Subtract the prime number found from the current array element
            nums[i] -= primes.get(index);
        }

        // If no issues found, return true
        return true;
    }

    // Helper method to perform binary search and find the smallest prime number greater than x
    private int binarySearch(List<Integer> primes, int x) {
        int left = 0, right = primes.size();
        while (left < right) {
            int mid = (left + right) >> 1; // Equivalent to (left + right) / 2 but avoids overflow
            if (primes.get(mid) > x) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }
        // Return the index of the smallest prime number greater than x
        return left;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    bool primeSubOperation(vector<int>& nums) {
        vector<int> primes;
        // Generate all prime numbers up to 1000
        for (int i = 2; i <= 1000; ++i) {
            bool isPrime = true;
            // Check if 'i' is divisible by any known prime
            for (int prime : primes) {
                if (i % prime == 0) {
                    isPrime = false;
                    break;
                }
            }
            // If 'i' is prime, add it to the list of primes
            if (isPrime) {
                primes.push_back(i);
            }
        }

        int size = nums.size();
        // Process each number in the array starting from the second-to-last
        for (int i = size - 2; i >= 0; --i) {
            // If the current element is smaller than the next one, move on
            if (nums[i] < nums[i + 1]) {
                continue;
            }

            // Find the smallest prime such that nums[i] - prime is larger than nums[i + 1]
            int idx = upper_bound(primes.begin(), primes.end(), nums[i] - nums[i + 1]) - primes.begin();

            // If no such prime exists or the prime is greater than or equal to nums[i], return false
            if (idx == primes.size() || primes[idx] >= nums[i]) {
                return false;
            }

            // Reduce nums[i] by the found prime number to satisfy the condition
            nums[i] -= primes[idx];
        }

        // If all elements satisfy the condition, return true
        return true;
    }
};

```

Typescript Solution

```typescript
// A function to determine whether a series of subtraction operations can make the array strictly increasing
// by subtracting a prime number from an element if necessary.
function primeSubOperation(nums: number[]): boolean {

    // Store prime numbers in an array
    const primes: number[] = [];

    // Generate prime numbers up to 1000 and store them in the primes array
    for (let i = 2; i <= 1000; ++i) {
        let isPrime = true;
        for (const prime of primes) {
            if (i % prime === 0) {
                isPrime = false;
                break;
            }
        }
        if (isPrime) {
            primes.push(i);
        }
    }

    // Binary search to find the index of the smallest prime number
    // which is greater than the given number x
    const searchPrimeIndex = (x: number): number => {
        let left = 0;
        let right = primes.length;
        while (left < right) {
            const mid = (left + right) >> 1;
            if (primes[mid] > x) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }
        return left;
    };

    // Iterate through the numbers array, starting from the second to last element, moving backwards
    const count = nums.length;
    for (let i = count - 2; i >= 0; --i) {
        // Continue to the next if the current number is less than the next number (strictly increasing)
        if (nums[i] < nums[i + 1]) {
            continue;
        }

        // Find the index of the smallest prime number that is greater than the difference needed
        const primeIndex = searchPrimeIndex(nums[i] - nums[i + 1]);

        // If we cannot find a suitable prime number to make the sequence increasing, return false
        if (primeIndex === primes.length || primes[primeIndex] >= nums[i]) {
            return false;
        }

        // Perform the subtraction to make the sequence strictly increasing
        nums[i] -= primes[primeIndex];
    }

    // If all values are checked and the array can be made strictly increasing, return true
    return true;
}

```

Time and Space Complexity

The time complexity of the given code can be analyzed in several parts:

1. Generating the prime numbers up to `max(nums)` involves checking each number against all previously found primes. In the worst case, if `max(nums)` is a prime number itself, this would generate `O(max(nums) / \log(max(nums)))` primes, since the density of primes is roughly `1 / \log(n)` where `n` is the number being considered for primality. However, this check is done `O(\sqrt{max(num)})` times for each number since we only need to check divisibility up to its square root. This gives us a complexity of `O(max(nums) \sqrt{max(nums)} / \log(max(nums)))` for the loop generating primes.
2. The main processing of `nums` array iterates over the elements once, which contributes `O(n)`, where `n` is the length of `nums`.
3. The `bisect_right` function is used within the loop over `nums`. Assuming that the list of prime numbers is roughly `m` in size, where `m` is `O(max(nums) / \log(max(nums)))`, the `bisect_right` operation contributes `O(\log m)` complexity each time it's called. It's called once for each of `n` iterations, so that part contributes `O(n \log m)`.
4. The operation `nums[i] -= p[j]` is constant time, `O(1)`.

Considering the time complexities together, the dominating term is from generating the primes and the iterative process with binary search. Since the prime generation complexity is less common in analysis, we'll assume that `max(nums)` is bounded by a polynomial size of `n`, which means that `m` is `O(n / \log(n))`. Hence, we focus on the `O(n \log m)` term for binary search within the loop. This suggests a time complexity of `O(n(\log(n) - \log(\log(n))))`, which simplifies to `O(n \log n)` under the assumption that `max(nums)` is not exponentially larger than `n`.

The space complexity of the algorithm is dominated by the list `p` that stores the primes. As calculated before, this will be about `O(max(nums) / \log(max(nums)))`. Using the same bounding assumption as for the time complexity, we see that this simplifies to `O(n)`, under the assumption that `max(nums)` is not exponentially larger than `n`.

So, in agreement with the reference answer, we can say the time complexity is `O(n \log n)` and the space complexity is `O(n)`.