1425. Constrained Subsequence Sum

Hard | Queue | Array | Dynamic Programming | Sliding Window | Monotonic Queue | Heap (Priority Queue) | Leetcode Link

Problem Description

Given an integer array `nums` and an integer `k`, you are tasked with finding the maximum sum of a non-empty subsequence within the array. A subsequence here is defined as a sequence that can be derived from the array by deleting some elements without changing the order of the remaining elements. However, there's an additional constraint with regard to the subsequence. For any two consecutive integers in the subsequence, say `nums[i]` and `nums[j]` where `i < j`, the difference between the indices `j - i` must be less than or equal to `k`.

In simplified terms, you need to choose a subsequence such that any two adjacent numbers in this subsequence are not more than `k` positions apart in the original array, and the sum of this subsequence is as large as possible.

Intuition

To arrive at the solution, consider two important aspects: dynamic programming (DP) for keeping track of the subsequences and a 'sliding window' to enforce the constraint of the elements being within a distance `k` of each other.

With dynamic programming, create an array `dp` where each `dp[i]` stores the maximum sum of the subsequence up to the i-th element by following the constraint. To maintain the `k` distance constraint, use a deque (double-ended queue) that acts like a sliding window, carrying indices of elements that are potential candidates for the maximum sum. At each step:

- Remove indices from the front of the deque which are out of the current window of size `k`.
- Calculate `dp[i]` by adding the current element `nums[i]` to the max sum of the window (which is `dp[q[0]]`, q being the deque). If there are no elements in `q`, just take the current element `nums[i]`.
- Since we want to maintain a decreasing order of `dp` values in the deque to quickly access the maximum sum, remove elements from the end of the deque which have a `dp` value less than or equal to the current `dp[i]`.
- Add the current index `i` to the deque.
- Update the answer with the maximum `dp[i]` seen so far.

This approach ensures that the deque always contains indices whose `dp` values form a decreasing sequence, and thus the front of the deque gives us the maximum sum for the current window, while satisfying the distance constraint.

Solution Approach

The implementation uses dynamic programming in combination with a deque to efficiently solve the problem by remembering previous results and ensuring that the constraint is maintained.

The initial setup involves creating a DP table as a list `dp` of size `n` (the length of `nums`) and initializing an answer variable `ans` to negative infinity to keep track of the maximum subsequence sum found so far.

Here is a step-by-step walkthrough of the implementation:

1. Iterate over the array `nums` using the index `i` and the value `v`. This loop will determine the `dp[i]` value for each element in `nums`.

2. If the deque `q` has elements and the oldest element's index in the deque (`q[0]`) is out of the window (i.e., `i - q[0] > k`), pop it from the front of the deque.

3. Set `dp[i]` to the maximum of 0 and the sum of the value `v` added to `dp[q[0]]` (the maximum sum within the window). If `q` is empty, just add `v` to 0. This step ensures that we do not consider subsequences with negative sums since they would decrease the overall maximum sum we are trying to compute.

4. While the deque has elements and the last element in the deque has a `dp` value less than or equal to `dp[i]` (since `dp[i]` now holds the maximum sum up to `i`), pop elements from the end of the deque. This maintains the invariant that the deque holds indices in decreasing order of their `dp` values.

5. Append the current index `i` to the deque.

6. Update `ans` with the maximum value between `ans` and `dp[i]` to update the global maximum sum each time `dp[i]` is calculated.

7. Once the iteration over `nums` is complete, `ans` will contain the maximum sum of a subsequence satisfying the given constraint, so return `ans`.

This algorithm makes use of the following patterns and data structures:

- **Dynamic Programming:** By storing and reusing solutions to subproblems (`dp[i]`), we solve larger problems efficiently.
- **Monotonic Queue (Deque):** A deque allows us to efficiently track the "window" of elements that are within `k` distance from the current element while simultaneously maintaining a monotonically decreasing order of `dp` values.
- **Sliding Window Technique:** The concept of having a window moving through the data while maintaining certain conditions (here, a maximum sum within a distance `k`) is a classic example of this technique.

Example Walkthrough

Let's assume we have an integer array `nums = [10, 2, -10, 5, 20]` and `k = 2`, and we want to find the maximum sum of a non-empty subsequence such that consecutive elements in the subsequence are not more than `k` positions apart in the original array.

We initialize our `dp` and `ans`:

- `dp = [0, 0, 0, 0, 0]` with the same length as `nums`
- `ans = -∞` (minimum possible value)

We start by iterating over `nums`:

1. For `i = 0`, `v = 10`. The deque `q` is empty, so `dp[0] = 10` and `q = [0]`. Now `ans = 10`.
2. For `i = 1`, `v = 2`. Since `q[0]` is `0` and `i - q[0] = 1` which is within the range `k`, we calculate `dp[1] = max(dp[q[0]] + v, v) = max(10 + 2, 2) = 12`. Now `q = [0, 1]` after cleaning outnumbers by `dp` value (no changes in this step), and `ans = 12`.
3. For `i = 2`, `v = -10`. Since `i - q[0] = 2` which is within `k`, we calculate `dp[2] = max(dp[q[0]] + v, v) = max(12 - 10, -10)` but here both options are negative, so `dp[2] = 0`. We then remove from `q` since `dp[1] > dp[2]`. Now `q = [1]` and `ans = 12`.
4. For `i = 3`, `v = 5`. Since `i - q[0] = 2` which is equal to `k`, we can use `q[0]`. Thus, `dp[3] = max(dp[q[0]] + v, v) = max(12 + 5, 5) = 17` and `q = [1, 3]`. We update `ans = 17` now.
5. For `i = 4`, `v = 20`. Here, we first remove `q[0]` because `i - q[0] = 3` which is greater than `k`, leaving `q = [3]`. We then calculate `dp[4] = dp[q[0]] + v = 17 + 20 = 37`, so `q = [4]` after removing `q[0]` because `dp[3] < dp[4]`. `ans` is updated to `37`.

After iterating through the loop, we've looked at all possible subsequences that obey the `k` limit rule. The maximum subsequence sum is stored in `ans` which is `37`.

To summarize this walk-through:

- We used DP to remember the maximum subsequence sums for subarrays.
- We maintained a deque `q` to ensure elements are within `k` distance in the subsequence.
- We updated `ans` at every step with the maximum value of `dp[i]`.
- At the end of the loop, `ans` gave us the maximum sum possible under the given constraints.

Python Solution

```python
from collections import deque
from typing import List

class Solution:
    def constrainedSubsetSum(self, nums: List[int], k: int) -> int:
        # The number of elements in nums
        n = len(nums)

        # Initialize a list(dp) to store the maximum subset sum ending at each index
        dp = [0] * n

        # Initializing the answer to negative infinity to handle negative numbers
        max_sum = float('-inf')

        # A deque to keep the indexes of useful elements in our window of size k
        q = deque()

        # Loop through each number in nums
        for i, value in enumerate(nums):
            # If the first element of deque is out of the window of size k, remove it
            if q and i - q[0] > k:
                q.popleft()

            # Calculate the max subset sum at this index as the greater of 0 or the sum at the top of the deque plus the current valu
            dp[i] = max(0, dp[q[0]] if q else 0) + value

            # Pop elements from deque if they have a smaller subset sum than dp[i],
            # because they are not useful for future calculations
            while q and dp[q[-1]] <= dp[i]:
                q.pop()

            # Append the current index to the deque
            q.append(i)

            # Update the maximum answer so far with the current dp value
            max_sum = max(max_sum, dp[i])

        # Return the maximum subset sum found
        return max_sum
```

Java Solution

```java
class Solution {
    public int constrainedSubsetSum(int[] nums, int k) {
        // Length of the input array
        int n = nums.length;

        // Dynamic programming array to store the maximum subset sum
        // ending with nums[i]
        int[] dp = new int[n];

        // Initialize the answer with the smallest possible integer
        int answer = Integer.MIN_VALUE;

        // Declaring a deque to store indices of useful elements in dp array
        Deque<Integer> queue = new ArrayDeque<>();

        // Loop through each number in the input array
        for (int i = 0; i < n; ++i) {
            // Remove indices of elements which are out of the current sliding window
            if (!queue.isEmpty() && i - queue.peek() > k) {
                queue.poll();
            }

            // Calculate dp[i] by adding current number to the maximum of 0 or
            // the element at the front of the queue, which represents the maximum sum
            // within the sliding window
            dp[i] = Math.max(0, queue.isEmpty() ? 0 : dp[queue.peek()]) + nums[i];

            // Remove indices from the back of the queue where the dp value is less than dp[i]
            // to maintain the decreasing order of dp values in the queue
            while (!queue.isEmpty() && dp[queue.peekLast()] <= dp[i]) {
                queue.pollLast();
            }

            // Offer the current index to the queue
            queue.offer(i);

            // Update the answer with the maximum value of dp[i] so far
            answer = Math.max(answer, dp[i]);
        }

        // Return the maximum sum as answer
        return answer;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int constrainedSubsetSum(vector<int>& nums, int k) {
        // Get the size of the input vector nums.
        int n = nums.size();
        // Create a DP array to store the maximum subset sums at each position.
        vector<int> dp(n);
        // Initialize the answer variable with the minimum possible integer value.
        int maxSubsetSum = INT_MIN;
        // Initialize a double-ended queue to maintain the window of elements.
        deque<int> window;

        // Iterate over the elements in nums.
        for (int i = 0; i < n; ++i) {
            // If the window front is out of the allowed range [i-k, i], remove it.
            if (!window.empty() && i - window.front() > k) {
                window.pop_front();
            }
            // Compute the maximum subset sum at the current index i.
            // It is the maximum sum of the previous subset sum (if not empty) and the current number.
            dp[i] = max(0, window.empty() ? 0 : dp[window.front()]) + nums[i];
            // Update the overall maximum subset sum.
            maxSubsetSum = max(maxSubsetSum, dp[i]);
            // Maintain the window such that its elements are in decreasing order of their dp values.
            while (!window.empty() && dp[window.back()] <= dp[i]) {
                window.pop_back();
            }
            // Add the current index to the window.
            window.push_back(i);
        }
        // Return the maximum subset sum.
        return maxSubsetSum;
    }
};
```

Typescript Solution

```typescript
// Define type alias for easy reference
type Deque = number[];

// Initialize a deque to maintain the window of elements
let window: Deque = [];

// Function to calculate the constrained subset sum
function constrainedSubsetSum(nums: number[], k: number): number {
    // Get the size of the input array `nums`
    const n = nums.length;
    // Create an array to store the maximum subset sums at each position
    const dp: number[] = new Array(n);
    // Initialize the answer variable with the minimum possible number value
    let maxSubsetSum = Number.MIN_SAFE_INTEGER;

    // Iterate over the elements in `nums`
    for (let i = 0; i < n; ++i) {
        // If the window front is out of the allowed range [i-k, i], remove it
        if (window.length !== 0 && i - window[0] > k) {
            window.shift();
        }
        // Compute the maximum subset sum at the current index `i`
        // It is the max of 0 (to avoid negative sums) and the subset sum of the front of the window
        dp[i] = Math.max((window.length === 0 ? 0 : dp[window[0]]), 0) + nums[i];
        // Update the overall maximum subset sum
        maxSubsetSum = Math.max(maxSubsetSum, dp[i]);
        // Maintain the window such that its elements are in decreasing order of their dp values
        while (window.length !== 0 && dp[window[window.length - 1]] <= dp[i]) {
            window.pop();
        }
        // Add the current index to the window
        window.push(i);
    }
    // Return the maximum subset sum
    return maxSubsetSum;
}

// Example usage:
// let result = constrainedSubsetSum([10, 2, -10, 5, 20], 2);
// console.log(result); // Output will be the maximum constrained subset sum.
```

Time and Space Complexity

The given Python code defines a function `constrainedSubsetSum` which calculates the maximum sum of a non-empty subsequence of the array `nums`, where the subsequence satisfies the constraint that no two elements are farther apart than `k` in the original array.

Time Complexity

The time complexity of the code is `O(n)`, where `n` is the number of elements in the array `nums`. This is because the algorithm iterates over each element exactly once. Inside the loop, it performs operations that are constant time on average, such as adding or removing elements from the deque `q`. The deque maintains the maximum sum at each position within the window of `k`, and the operations are done in constant time due to the nature of the double-ended queue.

Even though we have a while-loop inside the for-loop that pops elements from the deque, this does not increase the overall time complexity. Each element is added once and removed at most once from the deque, leading to an average constant time for these operations per element.

Space Complexity

The space complexity of the code is `O(n)`, where `n` is the size of `nums`. This is because the algorithm allocates an array `dp` of the same size as `nums` to store the maximum sum up to each index. Furthermore, the deque `q` can at most contain `k` elements, where `k` is the constraint on the distance between the elements in the subsequence. However, since `k` is a constant with respect to `n`, the primary factor that affects the space complexity is the `dp` array.

In summary:

- Time complexity is `O(n)`.
- Space complexity is `O(n)`.