1043. Partition Array for Maximum Sum

Medium Array Dynamic Programming Leetcode Link

Problem Description

The problem presents us with an integer array `arr` and asks us to partition this array into continuous subarrays. Each of these subarrays can have a length of up to `k`, which is given as part of the input. After partitioning the array into these subarrays, we need to modify each subarray so that all its elements are equal to the maximum value in that subarray.

Our goal is to find the largest possible sum that can be obtained from the array after completing this partition and modification process. It's important to note that when choosing how to partition the array, we're looking for the method that will lead to this maximum sum. We are assured that the answer will not exceed the capacity of a 32-bit integer.

The intuitive challenge is to balance between creating subarrays with high maximum values and making these subarrays as large as possible, since the final sum will be influenced by both the size of the subarrays and the value to which their elements are set.

Intuition

The solution to this problem employs a dynamic programming approach, which is a method used to solve complex problems by breaking them down into simpler subproblems. The key insight in dynamic programming is that the solution to the overall problem depends on the solutions to its subproblems, and these subproblems often overlap.

In this case, our subproblems involve determining the maximum sum that can be obtained from the first `i` elements of the array for each `i` from 1 to `n`, where `n` is the total number of elements in `arr`.

We start by defining `f[i]` as the maximum sum obtainable from the first `i` elements after partitioning. To compute `f[i]`, we need to consider all possible ends for the last subarray that ends at position `i`. This involves looking at each possible starting position `j` for this subarray that is not more than `k` indices before `i`. As we consider each possible `j`, we keep track of the maximum value `mx` within this range and update `f[i]` by considering the maximum sum that can be achieved by making `arr[j - 1]` the first element of the last subarray.

The transition equation helps us update `f[i]` by considering the value of `f[i]` before adding the new subarray and the additional sum that comes from setting all elements in the current subarray to the maximum value found (`mx` multiplied by the length of the subarray, which is `(i - j + 1)`).

By iteratively building up from smaller subarrays to the full array, we ensure that each `f[i]` contains the optimum solution for the first `i` elements, and hence `f[n]` gives us the answer for the entire array.

Solution Approach

The problem is approached using a dynamic programming technique. Here is a step-by-step explanation of how the solution is implemented, corresponding to the provided Python code:

1. **Dynamic Programming Array Initialization:**
   - A dynamic programming array `f` is initialized with a length of `n + 1`, where `f[i]` will eventually store the maximum sum obtainable for the first `i` elements of `arr`.
   - This array is initially filled with zeroes, as we have not yet computed any subarray sums.
2. **Nested Loop Structure:**
   - We iterate through the array using a variable `i` which goes from `1` to `n`, inclusive. This represents the rightmost element of the current subarray window we are considering.
   - Within this outer loop, we have an inner reverse loop with a variable `j` that starts from `i` and goes backwards to `max(0, i - k)`, considering each possible subarray that ends at position `i` and has a length of at most `k`.
3. **Maximum Subarray Value:**
   - A variable `mx` is used to keep track of the maximum value in the subarray starting from a potential starting point `j`.
   - As we move leftwards in the inner loop (`j` decreases), we update `mx` to reflect the maximum value encountered so far using the expression `mx = max(mx, arr[j - 1])`.
4. **Updating Dynamic Programming Array:**
   - We then calculate the sum of the current subarray by multiplying the largest number found (`mx`) by the size of the subarray (which is `(i - j + 1)`). This reflects the sum of the partitioned subarray if all of its values are changed to the maximum value found in it.
   - The dynamic programming array `f[i]` is updated with the maximum of its current value and the sum of the subarray formed by adding the newly computed subarray sum to the maximum sum achievable before the current subarray (`f[j - 1]`).
   - The state transition equation used here is:

     ```
     f[i] = max(f[i], f[j - 1] + mx * (i - j + 1))
     ```

5. **Return the Final Answer:**
   - After populating the dynamic programming array with the best solutions for all subarray sizes, the final answer to the problem is the last element of the array, `f[n]`, which represents the maximum sum obtainable for the entire array.

By using dynamic programming, the solution avoids recomputing the sums for overlapping subproblems, which makes it efficient. The space complexity of the algorithm is linear (`O(n)`) due to the use of the dynamic programming array, and the time complexity is `O(n * k)` because of the nested loops, where for each `i`, a maximum of `k` positions is considered.

Example Walkthrough

Let's consider `arr = [1, 15, 7, 9, 2, 5, 10]` with `k = 3`. Our task is to partition `arr` into continuous subarrays of length at most `k`, and then maximize the sum of the array after each element in a subarray has been made equal to the maximum element of that subarray.

1. We start with `f[0] = 0` because no elements give a sum of 0.
2. For `i = 1 (arr[0] = 1)`, the only subarray we can have is `[1]`, and the maximum sum we can obtain is 1*(1) = 1. So, `f[1] = 1`.
3. At `i = 2 (arr[1] = 15)`, we could have a subarray `[1, 15]` or just `[15]`. The best option is to make subarray `[15]` because 15*(1) = 15 is greater than 15*(2) - 1*(1) = 29. So, `f[2] = 15`.
4. Moving on to `i = 3 (arr[2] = 7)`, we could have `[1, 15, 7]`, `[15, 7]`, or just `[7]`. The choice `[15, 7]` gives us the highest sum because 15*(2) = 30, while the others give lower sums. Adding the previous f[1], we get `f[3] = f[1] + 15*2 = 1 + 30 = 31`.
5. For `i = 4 (arr[3] = 9)`, the potential subarrays are `[1, 15, 7, 9]`, `[15, 7, 9]`, or `[7, 9]`. Out of these, `[15, 7, 9]` gives the highest contribution with 15*(3) = 45. So `f[4] = f[1] + 45 = 1 + 45 = 46`.
6. With `i = 5 (arr[4] = 2)`, we consider `[15, 7, 9, 2]`, `[7, 9, 2]`, and `[9, 2]`. However, `[9, 7, 2]` is not considered since its length is greater than `k`. Choosing `[9, 2]` allows us to add 9*(2) to f[3], giving the largest sum. Therefore, `f[5] = f[3] + 9*2 = 31 + 18 = 49`.
7. At `i = 6 (arr[5] = 5)`, we consider `[9, 2, 5]`, `[2, 5]`, and `[5]`. The subarray `[9, 2, 5]` gives the maximum contribution with 9 * 3 = 27. So, `f[6] = f[3] + 27 = 31 + 27 = 58`.
8. Finally, for `i = 7 (arr[6] = 10)`, we consider `[2, 5, 10]`, `[5, 10]`, and `[10]`. `[5, 10]` yields the best result with 10 * 2 contribution, giving us `f[7] = f[5] + 20 = 49 + 20 = 69`.

So, the maximum sum that can be obtained from the array after completing this partition and modification process is `f[7] = 69`.

To recap, we solved this by iterating over each element, considering every possible partition that could end at that element within the constraint of `k` length, and choosing the one which maximizes the sum at each step while leveraging previously computed results.

Python Solution

```python
class Solution:
    def maxSumAfterPartitioning(self, arr: List[int], k: int) -> int:
        # Length of the given array
        array_length = len(arr)

        # Initialize the dp (dynamic programming) array with 0's,
        # where dp[i] will be the max sum for the subarray arr[0:i]
        dp = [0] * (array_length + 1)

        # Start from the first element and compute the max sum for each subarray
        for i in range(1, array_length + 1):
            max_element = 0  # To keep track of the maximum element in the current partition

            # Check partitions of lengths from 1 to k
            for j in range(i, max(0, i - k), -1):

                # Update the maximum element in the current partition
                max_element = max(max_element, arr[j - 1])

                # Update the dp table:
                # dp[i] is the maximum of its previous value and the sum of the new partition
                # The new partition sum is calculated by multiplying the size of the partition
                # (i - j + 1) with the maximum element in that partition.
                dp[i] = max(dp[i], dp[j - 1] + max_element * (i - j + 1))

        # Return the maximum sum for the entire array
        return dp[array_length]
```

The updated code follows Python 3 standards, has standardized variable names that clearly describe their purposes, and contains comments that explain each step of the algorithm, hopefully making it clearer how the function works. Note that `List` should also be imported from `typing` module in the final code if not already in the script:

```python
from typing import List
```

Java Solution

```java
class Solution {

    // Function to find the maximum sum after partitioning the array
    public int maxSumAfterPartitioning(int[] arr, int k) {
        // n is the length of the array
        int n = arr.length;
        // dp array to store the maximum sum at each partition
        int[] dp = new int[n + 1];

        // Loop over the array elements
        for (int i = 1; i <= n; ++i) {
            // Initialize the maximum element in the current partition to zero
            int maxInPartition = 0;
            // Try all possible partitions of size up to k
            for (int j = i; j > Math.max(0, i - k); --j) {
                // Update the maximum element in this partition
                maxInPartition = Math.max(maxInPartition, arr[j - 1]);
                // Update dp[i] with the maximum sum using the maximum element times the size of the partition
                // and compare it with the existing value in dp[i] to keep the max sum at each partition
                dp[i] = Math.max(dp[i], dp[j - 1] + maxInPartition * (i - j + 1));
            }
        }
        // Return the maximum sum after partitioning
        return dp[n];
    }
}
```

C++ Solution

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

class Solution {
public:
    int maxSumAfterPartitioning(vector<int>& arr, int k) {
        int n = arr.size(); // Get the size of the array
        vector<int> dp(n + 1, 0); // Create a dynamic programming table initialized with zeros

        // Iterate over the array to fill the dp table
        for (int i = 1; i <= n; ++i) {
            int maxElement = 0; // Store the maximum element in the current partition

            // Try different partition lengths up to 'k'
            for (int j = i; j > max(0, i - k); --j) {
                // Update maxElement with the largest value in the current partition
                maxElement = max(maxElement, arr[j - 1]);

                // Update the dp table by considering the best partition ending at position 'i'
                dp[i] = max(dp[i], dp[j - 1] + maxElement * (i - j + 1));
            }
        }

        // Return the maximum sum after partitioning the last element
        return dp[n];
    }
};
```

Typescript Solution

```typescript
// Function to calculate the maximum sum of subarrays after partitioning
// arr: An array of integers
// k: An integer that defines the maximum length of each partition
function maxSumAfterPartitioning(arr: number[], k: number): number {
    const n: number = arr.length; // The length of the input array
    // Initialize an array to store the maximum sum of subarrays up to each index
    const dp: number[] = new Array(n + 1).fill(0);

    // Iterate through the array
    for (let i = 1; i <= n; ++i) {
        let maxElement: number = 0; // Variable to keep track of the max element in the current partition
        // Check all possible partitions up to the length 'k'
        for (let j = i; j > Math.max(0, i - k); --j) {
            maxElement = Math.max(maxElement, arr[j - 1]); // Update max element of the current partition
            // Update the dp array with the maximum sum by comparing the existing sum and
            // the new sum formed by adding the max element multiplied by the partition size
            dp[i] = Math.max(dp[i], dp[j - 1] + maxElement * (i - j + 1));
        }
    }
    // Return the maximum sum after partitioning, which is stored at the end of dp array
    return dp[n];
}
```

Time and Space Complexity

Time Complexity:

The time complexity of the given code is `O(n * k)`. This is because the outer loop runs for `n` iterations (from `1` to `n`), where `n` is the length of the input array `arr`. The inner loop runs up to `k` times for each outer loop iteration, but no more than `i` times (down to `max(0, i - k)`). For each iteration of the inner loop, a constant amount of work is done: updating the maximum value in the current window (`mx`) and possibly updating the maximum sum `f[i]`. So, the total time taken is proportional to `n * k`.

Space Complexity:

The space complexity of the code is `O(n)` since it employs a one-dimensional array `f` with a size of `n + 1` to store the maximum sum that can be obtained up to each index `0` to `n`.