1043. Partition Array for Maximum Sum

Dynamic Programming

Problem Description

Array

Medium

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

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.

implemented, corresponding to the provided Python code: **<u>Dynamic Programming Array Initialization:</u>**

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

The problem is approached using a <u>dynamic programming</u> technique. Here is a step-by-step explanation of how the solution is

Nested Loop Structure:

first i elements of arr.

Solution Approach

subarray, which is (i - j + 1).

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

• We iterate through the array using a variable i which goes from 1 to n, inclusive. This represents the rightmost element of the current

This array is initially filled with zeroes, as we have not yet computed any subarray sums.

each possible subarray that ends at position i and has a length of at most k.

- **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.
- mx = max(mx, arr[j 1]).**Updating <u>Dynamic Programming</u> 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 -

As we move leftwards in the inner loop (j decreases), we update mx to reflect the maximum value encountered so far using the expression

1]).

Return the Final Answer:

The state transition equation used here is:

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

is 0(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

After populating the <u>dynamic programming</u> array with the best solutions for all subarray sizes, the final answer to the problem is the last

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 (0(n)) due to the use of the dynamic programming array, and the time complexity

element of the array, f[n], which represents the maximum sum obtainable for the entire array.

We start with f[0] = 0 because no elements give a sum of 0. 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. 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

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 = 1

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

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.

31.

results.

class Solution:

Solution Implementation

Length of the given array

 $dp = [0] * (array_length + 1)$

for i in range(1, array_length + 1):

array_length = len(arr)

return dp[array_length]

subarray.

= 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.

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

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.

def maxSumAfterPartitioning(self, arr: List[int], k: int) -> int:

Initialize the dp (dynamic programming) array with 0's,

Check partitions of lengths from 1 to k

for j in range(i, max(0, i - k), -1):

Return the maximum sum for the entire array

public int maxSumAfterPartitioning(int[] arr, int k) {

// dp array to store the maximum sum at each partition

// Try all possible partitions of size up to k

for (int j = i; j > Math.max(0, i - k); --j) {

// Return the maximum sum after partitioning

int maxSumAfterPartitioning(vector<int>& arr, int k) {

int n = arr.size(); // Get the size of the array

// Function to calculate the maximum sum of subarrays after partitioning

// k: An integer that defines the maximum length of each partition

const dp: number[] = new Array(n + 1).fill(0);

function maxSumAfterPartitioning(arr: number[], k: number): number {

// Check all possible partitions up to the length 'k'

Initialize the dp (dynamic programming) array with 0's,

where dp[i] will be the max sum for the subarray arr[0:i]

for (let j = i; j > Math.max(0, i - k); ---j) {

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

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

// Update the maximum element in this partition

maxInPartition = Math.max(maxInPartition, arr[j - 1]);

// Initialize the maximum element in the current partition to zero

dp[i] = Math.max(dp[i], dp[j - 1] + maxInPartition * (i - j + 1));

vector<int> dp(n + 1, 0); // Create a dynamic programming table initialized with zeros

// n is the length of the array

// Loop over the array elements

int maxInPartition = 0;

for (int i = 1; i <= n; ++i) {

int[] dp = new int[n + 1];

where dp[i] will be the max sum for the subarray arr[0:i]

Start from the first element and compute the max sum for each subarray

Update the maximum element in the current partition

max_element = 0 # To keep track of the maximum element in the current partition

15*(1) = 15 is greater than 15*(2) - 1*(1) = 29. So, f[2] = 15.

Python

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. $dp[i] = max(dp[i], dp[j - 1] + max_element * (i - j + 1))$

The updated code follows Python 3 standards, has standardized variable names that clearly describe their purposes, and contains (

// 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

class Solution { // Function to find the maximum sum after partitioning the array

return dp[n];

// arr: An array of integers

// Iterate through the array

for (let i = 1; i <= n; ++i) {

 $dp = [0] * (array_length + 1)$

#include <vector>

#include <algorithm>

#include <cstring>

class Solution {

public:

int n = arr.length;

from typing import List

```python

Java

## C++

```
// 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
```

let maxElement: number = 0; // Variable to keep track of the max element in 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

maxElement = Math.max(maxElement, arr[j - 1]); // Update max element of the current partition

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

```python

from typing import List

return dp[n];

```
# 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]
```

Time Complexity:

Time and Space Complexity

The time complexity of the given code is 0(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.

The updated code follows Python 3 standards, has standardized variable names that clearly describe their purposes, and contains commen

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.