## 2464. Minimum Subarrays in a Valid Split

**Dynamic Programming** 

#### **Problem Description**

<u>Array</u>

**Math** 

Medium

In this problem, we must take an integer array nums and divide it into subarrays according to specific rules. A subarray here is defined as a contiguous part of the array. A splitting is considered valid if every subarray meets these conditions:

• The greatest common divisor (GCD) of the first and last elements of the subarray is greater than 1.

Number Theory

Every element in nums must be included in exactly one subarray.

validity criteria. If we can't split the array to meet these conditions, the function should return -1.

Our objective is to find the minimum number of such subarrays into which the original array can be split while maintaining the

To give an example, suppose nums = [2, 3, 5, 2, 4]. In this array, we could split it into two subarrays. One subarray could be [2, 3], as the GCD of 2 and 3 is 1; hence we cannot stop here. We consider the next element, which is 5, and since the GCD of 2

and 5 is also 1, we continue to the next element and check 2 with 2. Now, the GCD of 2 and 2 is 2, which is greater than 1, so we can make a valid subarray [2, 3, 5, 2]. The remaining single element, 4, by itself forms a valid subarray because the first and last elements are the same, and obviously, the GCD of a number with itself is greater than 1. Thus, we have split nums into 2 subarrays and met all the criteria, so the answer is 2.

Intuition

The solution approach requires dynamic programming to avoid recalculating subproblems multiple times. The idea is to use a

## function dfs which, through recursion, iteratively tries to split the array from a starting index, say i, incrementing one element at a time until we find a valid split (where the GCD of the first and last is greater than 1) and then moves to find the next split from the

next index, say j+1.

We utilize the concept of memoization with the @cache decorator to store the results of already computed states. This enhancement prevents unnecessary recomputations and allows us to find the minimum number of subarrays efficiently.

Each recursive call of the dfs function will return the minimum number of subarrays starting from the index i. We loop from i to the end of the array (n), each time checking if we can get a valid subarray with nums[i] as the first element and nums[j] as the

last element. If we find a valid split (GCD > 1), we compute the number of splits from j+1 onwards and add 1 to it (as 1 split is done up to j) and keep track of the least number of splits needed.

approach ensures that we explore all possible splits and always move towards the solution with the minimum splits required.

Solution Approach

The provided solution takes advantage of recursive depth-first search (DFS) with memoization, which is a common technique in

At the end of the recursion, we clear the cache for efficiency and return the answer if it's finite; otherwise, we return -1. This

subproblems to avoid recalculating them. Let's walk through the critical components of the algorithm:

• Recursion: The core of our solution is a recursive function named dfs. This function accepts an index i and returns the

minimum number of valid subarrays the array can be split into starting from that index. By calling dfs(0), we start the process

memoization for us. The dfs.cache\_clear() is called at the end before returning the final answer to clear the cache, ensuring

dynamic programming. The goal is to explore all possible ways to split the array and use memoization to store the results of

#### • Memoization: We use Python's @cache decorator from the functools module on our dfs function, which automatically handles

from the beginning of the array.

that no memory is wasted after the computation.
 Base Condition: The base case for our recursive function is when the index i is greater than or equal to n, the length of the array. In this case, the function returns 0, indicating that no further splits are needed.

Iteration within Recursion: Inside the dfs function, we loop through the array starting from the current index i to the end

index n. In each iteration, we are checking whether we can create a valid subarray starting at index i and ending at index j.

call dfs(j + 1) to determine how many subarrays we can get starting from the next index after the current valid subarray. To

ensure we find the minimum number of needed splits, we keep track of the smallest result returned from these recursive calls.

- We use the built-in [math](/problems/math-basics).gcd function to find the greatest common divisor between the first and last elements of the current subarray being considered.

   Validation: If the GCD of nums[i] and nums[j] is greater than 1, it indicates that a valid subarray is found. We then recursively
- valid subarray is found, we update ans to be the minimum between the existing ans and the result of 1 + dfs(j + 1) the 1 added represents the current valid subarray we have just found.
   Returning the Result: After the recursion terminates, if ans is still infinity, it means no valid subarray was created, and we

return -1. If ans is finite, it means a valid split was found, and we return the ans as the minimum number of splits needed.

In summary, the solution recursively explores all possible subarray splits and intelligently caches results to minimize runtime. This

dynamic programming approach, combining recursion and memoization, enables us to solve the problem optimally.

Let's illustrate the solution approach with a small example. Suppose we have an array nums = [2, 6, 3, 4].

Identifying the Minimum Splits: We initiate an ans variable for each recursive call with a value of infinity (inf). Whenever a

We start by calling dfs(0) because we need to examine the array starting from the first element.
 The recursive call dfs(i) loop is initiated, and i = 0. We start looking for valid subarrays beginning with the first element nums[0] = 2.
 The iteration within the recursion checks pairs of elements starting at i = 0 and ending at j = 0, 1, 2, ... until the end looking for valid

4. When j = 0, our subarray is [2]. The GCD of 2 and 2 is 2, which is greater than 1, so we found a valid subarray. We then call dfs(j + 1), which

5. The dfs(1) will perform its loop. For j = 1, we get the subarray [6], and the GCD of 6 and 6 is 6, so another valid subarray is found. We call

6. The dfs(2) will similarly check for valid subarrays starting from nums[2] = 3. However, we soon realize when we reach j = 3 that [3, 4] is a

valid subarray because the GCD of 3 and 4 is 1 (not valid), so we must include both in the subarray where the GCD of 3 and 4 is still 1. Finally,

7. As the base condition is met (no further elements left to process), the dfs function would return 0.
8. Adding up the splits, we see that dfs(2) returns 1 (it found one valid subarray), dfs(1) returns 2 (it found a valid subarray plus the one found by dfs(2)), and dfs(0) returns 3 since it identifies the valid subarray [2] and then relies on dfs(1).

#### At the end of the recursion, we find that the minimum number of subarrays is 3. This is the optimal solution as no fewer subarrays

Solution Implementation

from functools import lru\_cache

return 0

return min\_splits

length = len(nums)

dfs.cache\_clear()

# Get the length of the input list

# Clear the cache of the DFS function

min\_splits = float('inf')

def valid\_subarray\_split(self, nums: List[int]) -> int:

for end\_index in range(start\_index, length):

# Return the minimum number of splits found

if gcd(nums[start\_index], nums[end\_index]) > 1:

return min\_splits\_result if min\_splits\_result < float('inf') else -1

from math import gcd

class Solution:

from typing import List

**Python** 

**Example Walkthrough** 

splits.

**is** dfs(1).

dfs(2).

can satisfy the conditions.

found, which is 3 in this case. The cache is cleared after returning the final answer to free up memory.

# Start with a large number assuming no valid split is possible initially

min\_splits = min(min\_splits, 1 + dfs(end\_index + 1))

we must continue with the single element subarray [4], with a GCD of 4 with itself which is valid.

explored, the saved result would be used instead of re-calculating it.

After exploring all possibilities, since ans is not infinity, we do not return -1. Instead, we return the minimum number of splits

any index greater than 0, thus optimizing the solution significantly. If we were to reach an index 1 that had been previously

Throughout the process, memoization saves the result of each recursive call, preventing the re-computation of dfs(j) where j is

# Define a Depth-First Search (DFS) function with memoization to find the minimum number of valid splits
@lru\_cache(maxsize=None)
def dfs(start\_index):
 # Base case: if we have gone past the end of the array, no more splits are needed
 if start\_index >= length:

# Try splitting the array at different positions, updating the minimum splits if a valid split is found

# Check if the gcd of the numbers at the current segment (start\_index to end\_index) is greater than 1

# Return the final result: if min\_splits\_result is still inf, return -1 to indicate no valid splits; otherwise, return mi

# If so, include this segment and add 1 for the split to the result of the recursive call for the next segment

# # Call the DFS function starting with index 0 min\_splits\_result = dfs(0)

```
Java
class Solution {
   private int arrayLength;  // Represents the length of the input array
   private final int INFINITY = Integer.MAX_VALUE; // A value to represent infinity
   // Method to find the minimum number of valid subarrays with GCD greater than 1
   public int validSubarraySplit(int[] nums) {
       arrayLength = nums.length; // Initialize the length of the array
       memo = new int[arrayLength]; // Initialize the memo array
       numbers = nums;
                                 // Assign the input array to the instance variable numbers
       int answer = depthFirstSearch(0); // Begin the DFS from the first index
       return answer < INFINITY ? answer : −1; // If no valid split found, return −1, else return answer
   // Performs a depth-first search to find the minimum number of valid subarrays
   private int depthFirstSearch(int index) {
       // If the index is beyond the last element, return 0 as no further split needed
       if (index >= arrayLength) {
           return 0;
       // If we already computed the result for this index, return the stored value
       if (memo[index] > 0) {
           return memo[index];
       int answer = INFINITY; // Start with an infinite answer
       // Iterate over the array elements starting from current index
       for (int j = index; j < arrayLength; ++j) {</pre>
           // If the GCD of the starting element and the current element is greater than 1
           if (greatestCommonDivisor(numbers[index], numbers[j]) > 1) {
               // We have a valid subarray. Recursively solve for the remaining subarray,
               // counting 1 for the current valid subarray and attempting to minimize the answer
               answer = Math.min(answer, 1 + depthFirstSearch(j + 1));
       // Store the result for the current index in the memoization array
       memo[index] = answer;
       return answer;
   // Helper method to compute the Greatest Common Divisor of two numbers.
   private int greatestCommonDivisor(int a, int b) {
       return b == 0 ? a : greatestCommonDivisor(b, a % b);
```

C++

public:

#include <vector>

class Solution {

#include <functional>

#include <algorithm>

#include <numeric> // For std::gcd

static constexpr int INF = 0x3f3f3f3f;

int validSubarraySplit(std::vector<int>& nums) {

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

// Kick-start the DFS from the beginning of the array

// Return the total number of splits if answer is less than infinity, otherwise return -1

let answer: number = dfs(0);

// Define a constant for infinity to compare against during computation

std::vector<int> memo(n, 0); // Create a memoization array initialized to 0

if (i >= n) return 0; // Base case: beyond array boundaries

// Define the recursive Depth-First Search (DFS) function using std::function

if (memo[i] > 0) return memo[i]; // Return memoized result, if available

// Function to calculate the number of valid subarray splits

std::function<int(int)> dfs = [&](int i) -> int {

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

int result = INF; // Initialize result to infinity

// Check each subarray starting from index i

```
// Only consider the subarray if gcd of start and current elements is greater than 1
                if (std::gcd(nums[i], nums[j]) > 1) {
                    // Choose the smallest count of splits for subarrays
                    result = std::min(result, 1 + dfs(j + 1));
           // Memoize the answer for the current index i
           memo[i] = result;
            return result; // Return the result for the current subarray
       };
       // Kick-start the DFS from the beginning of the array
       int answer = dfs(0);
       // Return the total number of splits if answer is less than infinity, otherwise -1
       return answer < INF ? answer : -1;
};
TypeScript
const INF: number = Infinity; // Define a constant for infinity
// Array to store the memoized results
let memo: number[];
// Function to calculate the number of valid subarray splits
function validSubarraySplit(nums: number[]): number {
  let n: number = nums.length; // Get the size of the input array
  memo = new Array(n).fill(0); // Initialize the memoization array with 0
  // Recursive Depth-First Search (DFS) function
  const dfs = (i: number): number => {
   if (i >= n) return 0; // Base case: beyond array boundaries
   if (memo[i] > 0) return memo[i]; // Return memoized result if available
    let result: number = INF; // Initialize result to infinity
    // Check each subarray starting from index i
    for (let j: number = i; j < n; ++j) {</pre>
     // Only consider the subarray if gcd of start and current elements is greater than 1
      if (gcd(nums[i], nums[j]) > 1) {
       // Choose the smallest count of splits for subarrays
        result = Math.min(result, 1 + dfs(j + 1));
    // Memoize the answer for the current index i
    memo[i] = result;
   return result; // Return the result for the current subarray
```

# return answer < INF ? answer : -1; } // Function to compute the greatest common divisor (gcd) of two numbers function gcd(a: number, b: number): number {</pre>

**}**;

```
function gcd(a: number, b: number): number {
    while (b !== 0) {
      let t: number = b;
      b = a % b;
      a = t;
    return a;
from functools import lru_cache
from math import gcd
from typing import List
class Solution:
   def valid_subarray_split(self, nums: List[int]) -> int:
       # Define a Depth-First Search (DFS) function with memoization to find the minimum number of valid splits
       @lru_cache(maxsize=None)
       def dfs(start_index):
           # Base case: if we have gone past the end of the array, no more splits are needed
            if start_index >= length:
               return 0
           # Start with a large number assuming no valid split is possible initially
            min_splits = float('inf')
            # Try splitting the array at different positions, updating the minimum splits if a valid split is found
            for end_index in range(start_index, length):
               # Check if the gcd of the numbers at the current segment (start_index to end_index) is greater than 1
               if gcd(nums[start_index], nums[end_index]) > 1:
                   # If so, include this segment and add 1 for the split to the result of the recursive call for the next segment
                   min_splits = min(min_splits, 1 + dfs(end_index + 1))
            # Return the minimum number of splits found
            return min_splits
       # Get the length of the input list
        length = len(nums)
       # Call the DFS function starting with index 0
       min_splits_result = dfs(0)
       # Clear the cache of the DFS function
       dfs.cache_clear()
       # Return the final result: if min_splits_result is still inf, return -1 to indicate no valid splits; otherwise, return min_sp
```

# recursively with the starting index i, and for each call to dfs, there's a loop running from i to n. Within this loop, we are making a recursive call to dfs(j + 1), and we also calculate the greatest common divisor (gcd) for each pair of elements between i and j.

Time and Space Complexity

return min\_splits\_result if min\_splits\_result < float('inf') else -1</pre>

recursive call to dfs(j + 1), and we also calculate the greatest common divisor (gcd) for each pair of elements between i and j. The gcd operation itself can be considered as  $0(\log(\min(nums[i], nums[j])))$  on average, but in the worst case, the gcd can be considered 0(n) for very large numbers or certain sequences. Simplifying our analysis by considering gcd as 0(1) for an average case, the total operations are in the order of the sum of sequence from 1 to n, which is n(n + 1)/2, thus  $0(n^2)$ . This  $0(n^2)$  combined with the 0(n) for the iteration from i to n gives a complexity of  $0(n^3)$ .

The space complexity is primarily determined by the size of the cache used in the memoization and the depth of recursion. Memoization could potentially store a result for every starting index i, hence it can take 0(n) space. The maximum depth of the recursion determines the stack space, which is also 0(n) because the function could be recursively called starting from each index in nums. Therefore, the space complexity can be considered as 0(n) for the caching and 0(n) for the recursive call stack, which simplifies to 0(n).

The time complexity of the given code is  $0(n^3)$ , where n is the length of the nums list. This is because the dfs function is called