2580. Count Ways to Group Overlapping Ranges

Medium <u>Array</u> <u>Sorting</u>

Problem Description The given problem asks us to find different ways to partition a set of ranges into two distinct groups. Each range is a pair of

range must belong to exactly one group, and overlapping ranges—that is, ranges which share at least one common integer—must be in the same group. Our task is to find the total number of distinct ways to achieve such a partition and return this number modulo 10^9 + 7.

The problem is a combinatorial one which means that the solution typically involves counting certain arrangements or groupings. Since partitions are affected by whether ranges overlap, our approach must consider overlapping intervals as a single groupable

integers [start, end], which includes all integers from start to end inclusively. The two key rules for partitioning are: each

entity.

Intuition

The first step towards the solution of the problem revolves around identifying overlapping ranges. To facilitate this identification,

we sort the intervals based on their start and end points. Once sorted, we can iterate through the list of ranges and merge any

overlapping ranges. This merging process is akin to finding connected components in a graph, where overlapping ranges are

vertices connected by an edge.

As we merge overlapping ranges, we keep count of the non-overlapping intervals, or "chunks," since each of these can be assigned independently to one of the two groups. Here lies the crux of the problem: for each non-overlapping interval, we have two choices—the first group or the second group. Thus, if we end up with cnt non-overlapping intervals, there are 2^cnt ways to arrange them between the two groups.

The solution utilizes modular exponentiation to compute 2°cnt modulo 10°9 + 7, which is a standard way to handle potentially large numbers in combinatorial problems, ensuring the result fits within typical integer size bounds in programming languages.

Ultimately, the given Python code keeps track of non-overlapping intervals by comparing the start of the current range to the maximum end point (mx) seen thus far. If the current start is greater than mx, we have encountered a new non-overlapping interval. The max(mx) and undeteresting all non-overlapping interval. The max(mx) and undeteresting all non-overlapping interval.

maximum end point (mx) seen thus far. If the current start is greater than mx, we have encountered a new non-overlapping interval. The max(mx, end) update ensures we consider the entire merged interval for future comparisons. After counting all non-overlapping intervals, the pow function in Python computes the result of 2^cnt modulo $10^9 + 7$, thus providing the total number of ways to split the ranges into two groups that satisfy the constraints.

The solution to this problem involves understanding the characteristics of interval partitioning and efficiently calculating the number of ways to split these intervals into two non-overlapping groups.

Here's an overview of the steps involved, further clarified with algorithms and data structures used:

1. **Sort the intervals**: Before we can count the non-overlapping intervals, we need to be able to identify them easily. <u>Sorting</u> the

linear fashion. In Python, this is done by simply calling the sort() method on the ranges list which sorts in-place.

given ranges list by their starting points (and also by the end points in case of identical starts) helps us to handle them in a

Sorting brings a pattern to how we explore the intervals—ranges that could possibly overlap are now next to each other, which simplifies the process of merging.

previous and does not contribute to increasing the count.

the total number of ways to partition the set of ranges is 2^{cnt}.

Solution Approach

2. **Merging Overlapping Intervals**: As we iterate through the sorted intervals, we use a variable mx to keep track of the farthest end point we've seen so far. For each interval, if the start point is greater than mx, it signifies the beginning of a new non-

overlapping interval, and we increment our count cnt. If the start is not greater than mx, the interval overlaps with the

We then update mx to be the maximum of itself and the current interval's end point to reflect the most extended range of

Calculating the Number of Ways: Given cnt non-overlapping intervals, and knowing each interval has two choices of groups,

To perform this calculation, we could multiply by 2 for each non-overlapping interval we find, but that might become

current overlapping intervals.

Counting Non-Overlapping Intervals: The variable cnt represents the count of these non-overlapped, merged intervals.

Each of them can either go to the first group or the second, giving us two choices per interval.

computationally expensive and prone to overflow errors for very large numbers. Instead, we use modular exponentiation.

5. **Modular Exponentiation**: To find 2^cnt modulo 10^9 + 7, we use the pow function in Python which takes three arguments—base 2, exponent cnt, and modulus 10^9 + 7. This power function uses a fast exponentiation algorithm to efficiently compute large powers under a modulus, a process referred to as modular exponentiation.

In summary, the solution exploits the linearity of a sorted list to merge overlapping intervals, a count to track independent

choices, and modular arithmetic to handle large numbers. The sorting incurs an O(n log n) time complexity, where n is the

number of intervals. After sorting, the merge process and the counting that follows is done in a single pass, which contributes an

O(n) time complexity. The overall space complexity is O(log n) due to the recursive stack calls that could happen in sorting

Using these strategies, the code provided efficiently solves the problem while avoiding common issues associated with large number calculations in combinatorial problems.

Sort the intervals: We start by sorting these ranges. Sorted ranges: [[1, 3], [2, 5], [6, 8], [9, 10]] (Note: Since the ranges are already sorted, we don't need to do anything in this case.)
 Merging Overlapping Intervals: Now, we need to merge overlapping ranges.
 Start with mx = 0 (maximum end point seen so far).

• For the first range [1, 3], since 1 > mx, it is a new non-overlapping interval, so increment cnt (count of non-overlapping intervals).

Move to the next range [2, 5], since 2 <= mx, it overlaps with the previous range, so we keep cnt the same.

• The next range [6, 8] has a start greater than mx, so it's a new non-overlapping interval, increment cnt.

Update mx to max(mx, end) which is 8.
 Finally, the last range [9, 10] also starts after mx, so increment cnt once more.
 Update mx to 10.

Calculating the Number of Ways: Since each non-overlapping interval can be assigned to either of two groups, the total

Counting Non-Overlapping Intervals: We determined that there are 3 non-overlapping intervals after merging.

number of distinct ways is 2^{cnt}, which is 2³ = 8 ways.

Python

Java

class Solution {

class Solution:

Solution Implementation

intervals.sort()

mod = 10**9 + 7

from typing import List

algorithm and the pow function.

Update mx to max(mx, end) which is 3.

Update mx to max(mx, end) which is 5.

Example Walkthrough

In conclusion, for the given example with ranges [[1, 3], [2, 5], [6, 8], [9, 10]], there are 8 distinct ways to partition

Modular Exponentiation: Finally, we calculate 2^{cnt} modulo 10⁹ + 7.

Using the pow function in Python: pow(2, 3, 10**9 + 7) which equals 8.

def countWays(self, intervals: List[List[int]]) -> int:

Sort intervals by their starting points

overlap_count, $\max_{end} = 0$, -1

if start > max end:

max_end = max(max_end, end)

and take the result modulo mod

return pow(2, overlap_count, mod)

public int countWavs(int[][] ranges) {

if (range[0] > maxEnd) {

// Update the maximum endpoint.

maxEnd = Math.max(maxEnd, range[1]);

return quickPow(2, count, (int) 1e9 + 7);

count++;

// Sort the ranges by their starting points.

Arrays.sort(ranges, $(a, b) \rightarrow a[0] - b[0]$);

After merging, we have cnt = 3 non-overlapping intervals.

Let's say we are given the following ranges: [[1, 3], [2, 5], [6, 8], [9, 10]].

- these ranges into two groups that satisfy the given constraints.
- # Iterate through each interval for start, end in intervals: # If the current start is greater than the max end found so far, # it means there is no overlap with the previous intervals

Initialize the overlap counter and max_end variable to keep track of the furthest end point

Update the max end to the maximum of current max_end and the current interval's end

overlap_count += 1 # Increment the count for non-overlapping interval

Define the modulo for the result as per the problem statement

int count = 0; // Initialize the count of distinct segments.

// Calculate 2^count modulo 10^9+7 to get the number of ways.

// Define long long type for large number calculations

// Iterate over each bit of the exponent

for (; exponent > 0; exponent >>= 1) {

base = (base * base) % modulus;

return quickPowerMod(2, disjointRangeCount, 1e9 + 7);

// Sort the ranges in increasing order based on their start values

let maxEnd = -1; // Initialize the maximum end value seen so far

let totalWays = 1; // Initialize total ways to count combinations

totalWays = (totalWays * 2) % MODULO;

const MODULO = 10 ** 9 + 7; // Define the modulo value for avoiding large numbers

// If the start of the current range is greater than the maxEnd seen so far,

// this indicates a new independent interval, so we double the ways and take modulo.

if (exponent & 1) {

function countWays(ranges: number[][]): number {

ranges.sort((a, b) \Rightarrow a[0] - b[0]);

for (const [start, end] of ranges) {

// Iterate over each range

if (start > maxEnd) {

// Define a quick power function to calculate (a ^ n) mod

result = (result * base) % modulus;

// Square the base for the next bit of exponent

auto quickPowerMod = [&](ll base, int exponent, int modulus) {

// If the current bit of exponent is set, multiply the result with base

// Since there are 2 ways to cover each disjoint range, return 2^count of such ranges mod 10^9+7

using ll = long long;

ll result = 1:

return result;

Calculate the number of ways to arrange the intervals

There are 2 options for each non-overlapping interval (include or exclude)

using power of 2 (as each non-overlapping interval doubles the number of ways)

int maxEnd = -1; // Variable to keep track of the maximum endpoint seen so far. // Iterate through the ranges. for (int[] range : ranges) { // If the current range starts after the maximum endpoint so far, // it is a distinct segment which increases the count.

```
* Calculate the power of a number modulo 'mod' efficiently using binary exponentiation.
     * @param base The base number.
     * @param exponent The exponent.
     * @param mod The modulus for the operation.
     * @return The result of (base^exponent) % mod.
    private int quickPow(long base, int exponent, int mod) {
        long result = 1: // Initialize result to 1.
        // Loop until the exponent becomes zero.
        for (; exponent > 0; exponent >>= 1) {
            // If the least significant bit of exponent is 1, multiply the result with base.
            if ((exponent & 1) == 1) {
                result = result * base % mod;
            // Square the base for the next iteration.
            base = base * base % mod;
        return (int) result; // Return the result as an integer.
C++
#include <vector>
#include <algorithm>
class Solution {
public:
    int countWays(vector<vector<int>>& ranges) {
        // Sort the intervals based on their starting points
        sort(ranges.begin(), ranges.end());
        // Initialize the counter for disjoint ranges and max end of intervals seen so far
        int disjointRangeCount = 0, maxEnd = -1;
        // Traverse through each range in the sorted ranges
        for (const auto& range : ranges) {
            // Increment the counter if the current range starts after the max end of previous ranges
            if (range[0] > maxEnd) {
                disjointRangeCount++;
            // Update max end. if current range's end is greater
            maxEnd = max(maxEnd, range[1]);
```

};

TypeScript

```
// Update the maxEnd with the maximum of current range's end and maxEnd
        maxEnd = Math.max(maxEnd, end);
    // Return the total number of ways to cover all intervals modulated by MODULO
    return totalWays;
from typing import List
class Solution:
    def countWays(self, intervals: List[List[int]]) -> int:
        # Sort intervals by their starting points
        intervals.sort()
        # Initialize the overlap counter and max_end variable to keep track of the furthest end point
        overlap_count, \max_{end} = 0, -1
        # Iterate through each interval
        for start, end in intervals:
            # If the current start is greater than the max end found so far,
            # it means there is no overlap with the previous intervals
            if start > max end:
                overlap_count += 1 # Increment the count for non-overlapping interval
            # Update the max end to the maximum of current max_end and the current interval's end
            max_end = max(max_end, end)
        # Define the modulo for the result as per the problem statement
        mod = 10**9 + 7
        # There are 2 options for each non-overlapping interval (include or exclude)
        # Calculate the number of ways to arrange the intervals
        # using power of 2 (as each non-overlapping interval doubles the number of ways)
        # and take the result modulo mod
        return pow(2, overlap_count, mod)
Time and Space Complexity
Time Complexity
```

The provided code has two primary operations that contribute to its time complexity: 1. Sorting the ranges list. 2. Traversing the sorted ranges list to calculate the number of ways.

The sort() method on the list has a time complexity of $0(n \log n)$, where n is the number of elements in the ranges list. This is because the sorting operation is based on the TimSort algorithm, which is a hybrid sorting algorithm derived from merge sort and insertion sort.

After sorting, the for-loop iterates over the ranges exactly once. The iteration has a constant time complexity of O(1) per element for basic operations like comparison and assignment. Thus, for O(n) elements, this part of the code has a time complexity of O(n).

Combining these two parts, the overall time complexity of the code is $O(n \log n) + O(n)$, which simplifies to $O(n \log n)$ because the $n \log n$ term dominates for large n.

Space Complexity

Since the input list ranges is sorted in place, no additional space proportional to the size of the input is required except for the internal space used by the sorting algorithm which is O(n) in the worst case for TimSort.

However, the cnt, mx variables use a constant amount of space, and the mod is also constant, contributing a total of O(1) space.

The space complexity of the code is related to the space used by the input and the internal mechanism of the sort operation.

Therefore, the total space complexity of the code is O(n) (space used by the sorting algorithm) plus O(1) (space used by the variables), which is O(n) overall.