2518. Number of Great Partitions Array Dynamic Programming Hard

Leetcode Link

In this problem, we are given an array called nums which contains positive integers and another integer k. Our task is to find out how

Problem Description

many distinct ways (great partitions) there are to split the array into two groups such that the sum of the numbers in each group is at

partition.

least k. A great partition means that neither of the two groups has a sum that is less than k. Each element from the array nums must be put into one, and only one, of the two groups. The order of elements within the groups is not important, but distinct partitions are those where at least one element appears in a different group compared to another

Since there can be many distinct great partitions, we are required to return this number modulo 10^9 + 7 to keep the number

manageable and within the bounds of typical integer limits. One key point to understand is that if the sum of all elements in the array is less than 2k, we cannot possibly form two groups each having a sum greater or equal to k, hence the answer is straightforwardly 0 in such cases.

To find the number of distinct great partitions, we can start by considering a dynamic programming approach that helps us

understand whether it is possible to reach a certain sum with a subset of the given elements. The primary intuition behind the solution is to build up the count of subsets that can make up sums from 0 to k-1 respectively since sums from k to 2k-1 will automatically form a partition where both groups have at least k.

will represent the number of ways we can achieve a sum j using the first i numbers of the array nums.

At the start, we have only one way to achieve a sum of 0, which is by choosing no elements (f[0][0] = 1). As we iterate over the elements of nums, we keep updating the matrix based on whether we include the current element in the subset or not. This helps us accumulate counts of all possible subset sums up to k.

To arrive at the solution, we initialize a matrix 'f' with dimensions (n+1) x k where n is the number of elements in nums. Each f[i][j]

The total number of subsets of nums is 2^n. When building our dynamic programming matrix, we are actually counting the number of

subsets that reach every possible sum less than k. These counts have to be subtracted from the total since combining them with their complements does not yield a great partition—because at least one of the groups would have a sum of less than k. Once we calculate the subset counts, the final answer is adjusted to account for duplications and to ensure the result falls within the

 Multiply the number of all subsets (2ⁿ) by 2 modulo 10⁹ + 7. Subtract twice the sum of subsets that have a sum less than k from the total count. Add the modulus to ensure non-negativity, then take the result modulo 10^9 + 7 to get the final answer.

The solution uses a dynamic programming approach to solve the partition problem. Here's how the steps of the algorithm unfold: 1. First, we check if the total sum of the array is less than 2k. If it is, we immediately know it's impossible to partition the array into two groups each with a sum greater than or equal to k, so we return 0.

2. We define a two-dimensional array f with n+1 rows and k columns. n is the length of the input array nums. We initialize the first

row, which corresponds to using 0 elements from the array, such that f[0][0] is 1 (one way to make a sum of 0 with 0 elements)

3. We then use nested loops to fill in the dynamic programming matrix f. The outer loop runs through elements of nums from 1 to n,

4. For each element i and each sum j, f[i][j] is updated to be the sum of:

and the rest are 0.

Solution Approach

desired modulus of 10^9 + 7. Specifically, we:

○ f[i - 1][j], which is the count of ways to reach the sum j without using the i-th element.

inclusive. The inner loop runs through all possible sums j from 0 to k-1, inclusive.

specified modulus. This is why we see % mod after every arithmetic operation.

2 % mod) since every element either can be included or excluded from a set.

modulo to get the non-negative remainder within the desired range.

operations stay within bounds of the problem's specifications.

- If j is greater than or equal to nums[i 1] (the current element we're considering), we add f[i 1][j nums[i 1]] to the count, which represents using the i-th element to reach the sum j. 5. All computations are done modulo 10^9 + 7 to ensure we don't encounter integer overflow and that all operations fit within the
- 7. After filling out the dynamic programming matrix, we calculate the final answer. We know ans is the count of all subsets, but we want to exclude the cases where subsets sum to less than k. We sum up all f[n][j] for j from 0 to k-1 (which are the counts of subsets that don't form a great partition) and subtract twice this sum from ans. We multiply by 2 because for each such subset

that forms a sum less than k, both it and its complement would be counted in ans, but neither makes a great partition.

8. Finally, in case the subtraction makes the number negative, we add mod (which has no effect modulo mod) before taking the

6. As we calculate f[i][j], we also keep a running total of all possible subsets (represented by ans) by repeatedly doubling (ans *

sum with different subsets of the array. The dynamic programming matrix (f) is central to this solution, allowing us to build up the solution incrementally.

The code is a direct implementation of this dynamic programming approach, using the patterns of modularity arithmetic to ensure all

Let's walk through an example to illustrate the solution approach using a small array. Suppose we have an array nums = [1, 2, 3]

From a data structure perspective, we utilize a 2D array (list of lists in Python) to store the number of ways to achieve each possible

2. We define a dynamic programming matrix f with dimensions 4 x 3 (n+1 x k), as there are 3 elements in nums and we need to consider sums up to k-1 which is 2.

3. Initialize f to have the first row as [1, 0, 0] representing there is one way to reach a sum of 0 with 0 elements.

4. We fill in the matrix f. Iterating over nums and possible sums, the updated matrix would look like this at each step:

1. The total sum of the array 1 + 2 + 3 is 6, which is greater than 2*k (6), so it's possible to have two groups each with a sum of at

 Include the first element (1): 1 f = [2 [1, 0, 0], [1, 1, 0], // Include or exclude the 1

2 [1, 0, 0], [1, 1, 0], [1, 1, 1], // Include or exclude the 2 (we can now reach sums 1 and 2 using 1 and/or 2)

1 f = [

1 f = [

• {2, 3} & {1}

{1} & {2, 3}

{2} & {1, 3}

• {3} & {1, 2}

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

45

44 }

}, 1);

return result;

Time and Space Complexity

};

// Populate the DP array

for (int i = 1; i <= numCount; ++i) {</pre>

for (int j = 0; j < k; ++j) {

dp[i][j] = dp[i - 1][j];

// Update the result by multiplying it by 2 modulo MODULO

// If the current sum can accommodate the value, include it

dp[i][j] = (dp[i][j] + dp[i - 1][j - value]) % MODULO;

// Subtract the counts of partitions of all sums except k (sum we're interested in)

// Must ensure we're working with positive modulo for the subtracted value

result = (result - dp[numCount][j] * 2 % MODULO + MODULO) % MODULO;

let dp: number[][] = _.times(numCount + 1, () => _.times(k, _.constant(0)));

// The base case: there's one way to partition zero elements into subsets of sum zero

// If the current sum can fit the value, include it in the partition

// Adjust the result by taking into account the number of subsets that have the sum of k

dp[i][j] = (dp[i][j] + dp[i - 1][j - value]) % MODULO;

// All possible subsets divided by including or excluding the current element

return (acc - dp[numCount][j] * 2 % MODULO + MODULO) % MODULO;

// Carry over the count from the previous row (excluding the current number)

// Carry over the count from the previous row (excluding the current number)

result = static_cast<int>((result * 2LL) % MODULO);

int value = nums[i - 1];

if (j >= value) {

for (int j = 0; j < k; ++j) {

// Return the final result

// Import array utility functions from lodash

if (totalSum < k * 2) return 0;</pre>

// Get the number of elements in nums

// Initialize a 2D DP array with zeros

for (let i = 1; i <= numCount; i++) {</pre>

for (let j = 0; j < k; j++) {

if (j >= value) {

let value: number = nums[i - 1];

dp[i][j] = dp[i - 1][j];

let numCount: number = nums.length;

// Constant to represent the modulo operation for large numbers

// Populate the DP array with the number of partitions

let result: number = _.reduce(_.range(k), (acc, j) => {

// Return the final count of the number of valid partitions

result = (result + dp[numCount][k]) % MODULO;

return result;

Typescript Solution

import _ from "lodash";

dp[0][0] = 1;

const MODULO: number = 1e9 + 7;

Python Solution

class Solution:

2 [1, 0, 0],

3 [1, 1, 0],

4 [1, 1, 1],

later. Thus ans = $16 \% (10^9 + 7)$.

Summing these up gives us 1 + 2 + 2 = 5.

def countPartitions(self, nums: List[int], k: int) -> int:

Initialize a 2D array to use for dynamic programming

Iterate over the range from 1 to n inclusive

Double the answer on each iteration

dp[i][j] = dp[i - 1][j]

// Calculate the total sum of the array

long[][] dpCounts = new long[n + 1][k];

int currentValue = nums[i - 1];

// This will hold the final answer

answer = answer * 2 % MOD;

for (int j = 0; j < k; ++j) {

if (j >= currentValue) {

for (int i = 1; $i \le n$; ++i) {

long totalSum = 0;

for (int num : nums) {

totalSum += num;

if (totalSum < k * 2) {

// Length of the input array

return 0;

int n = nums.length;

dpCounts[0][0] = 1;

long answer = 1;

if j >= nums[i - 1]:

Base case: There's one way to have a sum of 0 (with no elements)

Carry over the previous count for this sum

If the current number can be used to build up the sum

// If the sum is less than 2k, we can't partition the array as requested

// Base case: for zero numbers, there's one way to achieve a sum of 0 (empty set)

// Dynamic programming array to hold counts for subproblems

// Update the answer as we consider the current element

dpCounts[i][j] = dpCounts[i - 1][j];

dp[i][j] = (dp[i][j] + dp[i - 1][j - nums[i - 1]]) % mod

Get the length of the nums list

 $dp = [[0] * k for _ in range(n + 1)]$

Variable to store the answer

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

ans = ans *2 % mod

for j in range(k):

n = len(nums)

dp[0][0] = 1

ans = 1

Check if the total sum of nums is less than the double of k

4 [1, 0, 0],

[1, 0, 0]

[1, 0, 0]

Include the second element (2):

Example Walkthrough

and an integer k = 3.

least k.

Include the third element (3):

```
5. All operations are done modulo 10^9 + 7.
```

to appear within bounds, which in this case just confirms 6 as the final count of great partitions.

So, for the given nums = [1, 2, 3] and k = 3, the number of distinct great partitions is 6. These partitions are:

[1, 2, 2] // Include or exclude the 3 (we can reach sums 1 and 2 using combinations of 1, 2, and 3)

6. The total number of subsets is $2^n = 2^3 = 8$. We double this to account for all possible subsets and exclude invalid subsets

7. From the matrix f, the subset counts for sums less than k are f[3] [0], f[3] [1], and f[3] [2], which are 1, 2, and 2, respectively.

8. Subtracting twice this sum from ans we get: 16 - 2*5 = 6. We add $10^9 + 7$ to ensure non-negativity and take modulo $10^9 + 7$

```
• {1, 2} & {3}
{1, 3} & {2}
```

if sum(nums) < k * 2: return 0 6 # Define a modulus number for the result to prevent integer overflow 8 9 mod = 10**9 + 7

In that case, there would be no way to partition the nums into k equal-sum parts

Dimensions are (n+1) x k, and it will hold the number of ways to reach a certain sum

Add the number of ways to reach the sum without the current number

```
32
33
34
35
36
```

```
37
 38
             # Use the final answers in dp to adjust the overall answer
 39
             # Subtract twice the sum of all counts for partitions of size (k-1)
 40
             # Adding mod before taking mod again for handling negative values
 41
             return (ans - sum(dp[-1]) * 2 + mod) % mod
 42
Java Solution
   class Solution {
         // Defining the MOD constant as per the problem constraints
         private static final int MOD = (int) 1e9 + 7;
  4
         // Method to count the number of ways to partition the array into two non-empty parts
  5
  6
         // such that the difference between their sums is equal to k
         public int countPartitions(int[] nums, int k) {
```

// f[i][j] will store the count of partitions for the first 'i' numbers that have sum 'j' in one part

// Count the subsets excluding the current value (carry over from the previous step)

dpCounts[i][j] = (dpCounts[i][j] + dpCounts[i - 1][j - currentValue]) % MOD;

// Count the subsets including the current value, if it can be part of the subset

45 46 // Subtracting the invalid partitions from the total answer (double counted subsets) 47 48

```
// Loop only up to k to consider only one side of the partition
             for (int j = 0; j < k; ++j) {
 49
                 answer = (answer - dpCounts[n][j] * 2 % MOD + MOD) % MOD;
 50
 51
 52
             // Return the final answer converted to an integer
             return (int) answer;
 53
 54
 55 }
 56
C++ Solution
   #include <vector>
    #include <numeric>
     #include <cstring> // Include header for memset
    class Solution {
    public:
         // Declare the MOD as a constant with a more descriptive name and use upper case for constants
         const int MODULO = 1e9 + 7;
  9
 10
         // Count the number of valid partitions that can divide the given set into k subsets
         int countPartitions(std::vector<int>& nums, int k) {
 11
             // Calculate the sum of all elements in the array
 12
             long long totalSum = std::accumulate(nums.begin(), nums.end(), 0LL);
 13
 14
             // If the total sum is less than 2k, no valid partitions exist
 15
             if (totalSum < k * 2) return 0;</pre>
 16
             // Get the number of elements in nums
 17
             int numCount = nums.size();
 18
 19
             // Create a 2D DP array to store the intermediate results
 20
             long long dp[numCount + 1][k];
 21
             // Initialize the answer to 1
 22
             int result = 1;
 23
 24
             // Clear the DP array with zeroes using memset
 25
             memset(dp, 0, sizeof dp);
 26
             // The base case: there's one way to partition zero elements into subsets of sum zero
 27
             dp[0][0] = 1;
```

// Function to count the number of valid partitions that can divide the given set into k subsets function countPartitions(nums: number[], k: number): number { // Calculate the sum of all elements in the array let totalSum: number = _.sum(nums); 10 // If the total sum is not enough to form k partitions, return 0 11

```
Time Complexity
The time complexity of the provided code can be analyzed based on the nested loops it uses:

    There is an outer loop iterating over the array nums with n elements.

    Inside this outer loop, there is an inner loop that iterates k times.

Each operation inside the inner loop executes in constant time. Therefore, the total number of operations performed will be 0(n * k).
Hence, the time complexity of the code is 0(n * k).
```

To analyze space complexity, we look at the memory allocation: • The 2D list f of size (n+1) * k is the primary memory consumer, where n is the length of nums and k is the provided partition value.

Space Complexity

No other significant memory usage is present that scales with the input size. Therefore, the space complexity is determined by the size of this 2D list.

Hence, the space complexity of the code is 0(n * k).