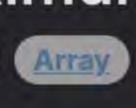
1589. Maximum Sum Obtained of Any Permutation

Leetcode Link

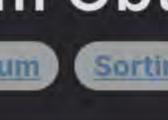
Medium



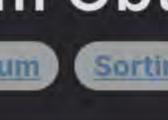
Problem Description

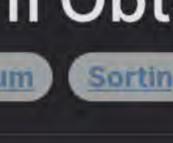


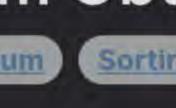












Sorting

In this problem, we have an array nums of integers and an array of requests. Each request is represented by a pair [start, end], which corresponds to a sum operation of the elements from nums [start] up to nums [end], inclusive. The goal is to calculate the maximum possible sum of all these individual request sums when we are allowed to permute nums. Since the result could be very large, it is asked to return the sum modulo 10^9 + 7, which is a common way to handle large numbers in programming problems to avoid integer overflow.

To find the maximum total sum of all requests, we need to understand that the frequency of each element being requested affects

Intuition

the total sum. If an element is included in many requests, we want to assign a larger value from nums to that position to maximize the sum. Conversely, if an element is rarely requested or not at all, it should be assigned a smaller value. The crux of the solution lies in the following insights:

1. Count the frequency of each index being requested: We can achieve this by using a "difference array" technique. In this

approach, for each request [start, end], we increment d[start] and decrement d[end + 1]. After that, a prefix sum pass over

the most frequently requested indices.

- the d array gives us the number of times each index is included in the range of the requests. 2. Sort both the nums and the frequency array d: By sorting the frequency array, we have a non-decreasing order of frequencies. Sorting nums also arranges the values in non-decreasing order. The intuition here is that we want to assign the highest values to
- 3. Calculate the maximum sum: We pair each number from nums with the corresponding frequency from d (after sorting both arrays), multiply them together, and accumulate the result to get the total sum. This works because both arrays are sorted, so the largest numbers are paired with the highest frequencies, ensuring the maximum total sum.
- Finally, we take the calculated sum modulo 10^9 + 7 to prevent overflow and return this value as the result.

1. Initialization of the difference array: We initialize an array d of zeros with the same length as nums. This array will help us keep

2. Populate the difference array: For each request [1, r] in requests, increment d[1] by 1 and decrement d[r + 1] by 1 (being

requests.

Solution Approach

careful not to go out of bounds). This is the difference array technique where d[1] represents the change at index 1, and d[r +

10**9 + 7, which we defined earlier as mod, and return this result.

Suppose our nums array is [3, 5, 7, 9] and our requests array is [[0, 1], [1, 3], [0, 2]].

For the request [1, 3], we increment d[1] and decrement d[4]: d = [1, 1, −1, 0, −1].

For the request [0, 2], we increment d[0] and decrement d[3]: d = [2, 1, -1, -1, -1].

arrays, which is the most time-consuming part of the algorithm.

Here is the step-by-step breakdown of implementing the solution:

track of how many times each index in nums is requested.

- 11 represents the change just after index r. 3. Calculate frequencies through prefix sums: We convert the difference array into a frequency array by calculating the prefix
- how many times each index is involved in a request after summing up the contributions from d[0] up to d[i]. 4. Sort the arrays: We sort both nums and the frequency array d in non-decreasing order using nums.sort() and d.sort(). By sorting these arrays, we ensure that the largest numbers in nums are lined up with the indices that occur most frequently in the

sum. In essence, for i in range(1, n): d[i] += d[i - 1] converts the difference array into a frequency array, which tells us

number in nums with its corresponding frequency in d and accumulate the sum. Since both arrays are sorted, the highest frequency gets paired with the largest number, which is critical for maximizing the total sum. 6. Return the result modulo 10^9 + 7: To avoid overflow issues and comply with the problem constraints, we take the sum modulo

The space complexity of the solution is O(n) due to the additional array d, and the time complexity is O(n log n) because we sort the

5. Calculate the total sum: We calculate the total sum using sum(a * b for a, b in zip(nums, d)). Here, we multiply each

Example Walkthrough Let's walk through the solution approach with a small example.

We start with an array d of the same length as nums, plus one for the technique to work (length of nums plus one). Here d = [0, 0, 0, 0, 0].

We go through each request and update our difference array d accordingly: For the request [0, 1], we increment d[0] and decrement d[2]: d = [1, 0, −1, 0, 0].

Now we convert the difference array to a frequency array:

Step 5: Calculate the total sum

Finally, we take our sum 37 modulo $10^9 + 7$. Since 37 is much less than $10^9 + 7$, our final result is 37.

We sort nums (it is already sorted) and we sort d: nums = [3, 5, 7, 9] and d = [0, 1, 2, 2, 3].

Therefore, the maximum possible sum of all request sums given the permutation of nums is 37.

Python Solution

from typing import List

class Solution:

We calculate the sum by multiplying each element in nums by its corresponding frequency in d in their respective sorted orders:

difference_array[end + 1] -= 1 18 # Calculate the prefix sum of the difference array to get the actual frequency array. for i in range(1, length_of_nums):

Sort both the nums array and the frequency array in non-decreasing order.

Initialize the modulus to avoid overflow issues with very large integers.

Calculate the total sum by summing the product of corresponding elements

Increment the start index to indicate a new range starts at this index.

Decrement the element after the end index to indicate the range ends at this index.

Step 1: Initialization of the difference array

Step 2: Populate the difference array

• d = [2, 1, -1, -1, -1] becomes d = [2, 3, 2, 1, 0] after calculating prefix sums. Step 4: Sort the arrays

Step 3: Calculate frequencies through prefix sums

Step 6: Return the result modulo 10^9 + 7

length_of_nums = len(nums)

for start, end in requests:

difference_array = [0] * length_of_nums

difference_array[start] += 1

if end + 1 < length_of_nums:</pre>

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

Arrays.sort(nums);

long maxSum = 0;

return (int) maxSum;

Arrays.sort(frequency);

final int mod = (int) 1e9 + 7;

for (int i = 0; i < length; ++i) {

// Return the maximum sum as an integer.

frequency[i] += frequency[i - 1];

// Sort both the nums array and the frequency array.

// Variable to store the result of the maximum sum.

maxSum = (maxSum + (long) nums[i] * frequency[i]) % mod;

• sum = 3*0 + 5*1 + 7*2 + 9*2 = 0 + 5 + 14 + 18 = 37

def maxSumRangeQuery(self, nums: List[int], requests: List[List[int]]) -> int: # Get the length of the nums array.

Initialize a difference array with the same length as nums.

Iterate over each request to populate the difference array.

19 20 difference_array[i] += difference_array[i - 1]

nums.sort()

difference_array.sort()

modulus = 10**9 + 7

10

11

13

14

15

16

17

23

24

25

29

30

31

17

18

20

21

22

23

26

27

28

29

30

33

34

35

36

37

39

40

41

43

42 }

```
32
           # from the sorted nums array and the sorted frequency array.
33
            total_sum = sum(num * frequency for num, frequency in zip(nums, difference_array))
34
35
           # Return the total sum modulo to ensure the result fits within the required range.
36
           return total_sum % modulus
37
Java Solution
   class Solution {
       public int maxSumRangeQuery(int[] nums, int[][] requests) {
           // Define the length of the nums array.
           int length = nums.length;
           // Create an array to keep track of the number of times each index is included in the ranges.
           int[] frequency = new int[length];
           // Iterate over all the requests to calculate the frequency of each index.
 9
           for (int[] request : requests) {
10
               int start = request[0], end = request[1];
               frequency[start]++;
               // Decrease the count for the index right after the end of this range
               if (end + 1 < length) {
14
                   frequency[end + 1]--;
15
16
```

// Modulo value to be used for not to exceed integer limits during the calculation.

// Compute the maximum sum by pairing the largest numbers with the highest frequencies.

// Convert the frequency array to a prefix sum array to get how many times each index is requested.

```
C++ Solution
  #include <vector> // Required for using the std::vector
 2 #include <algorithm> // Required for using the std::sort algorithm
   #include <cstring> // Required for using the memset function
   class Solution {
   public:
        int maxSumRangeQuery(std::vector<int>& nums, std::vector<std::vector<int>>& requests) {
           int n = nums.size();
           // Array to keep track of frequency of indices being requested
           std::vector<int> frequency(n, 0);
12
13
           // Increment start index and decrement just past the end index for each request
            for(const auto& req : requests) {
               int start = req[0], end = req[1];
                frequency[start]++;
               if (end + 1 < n) {
                   frequency[end + 1]--;
19
20
21
           // Convert frequency values into prefix sum to get the total count for each index
           for (int i = 1; i < n; ++i) {
                frequency[i] += frequency[i - 1];
24
25
26
27
           // Sort the input array and frequency array
           std::sort(nums.begin(), nums.end());
28
           std::sort(frequency.begin(), frequency.end());
30
            long long totalSum = 0;
31
            const int MOD = 1e9 + 7; // Modulo value for result
33
           // Compute the maximum sum of all ranges
34
           for (int i = 0; i < n; ++i) {
35
                totalSum = (totalSum + 1LL * nums[i] * frequency[i]) % MOD;
37
38
39
           // Casting to int as the Problem statement expects an int return type
           return static_cast<int>(totalSum);
40
42 };
43
```

for (let i = 0; i < lengthOfNums; ++i) {</pre> 26 answer = (answer + nums[i] * frequency[i]) % modulo; 28 29

Typescript Solution

11

12

13

14

16

17

18

19

20

22

23

24

25

30

32

31 }

function maxSumRangeQuery(nums: number[], requests: number[][]): number {

// Loop through each request to build the frequency array

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

if (end + 1 < lengthOfNums) {</pre>

frequency[end + 1]--;

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

// Sort the arrays to maximize the sum

frequency[i] += frequency[i - 1];

// Convert frequency array to prefix sum array

let answer = 0; // variable to store the final answer

// Calculate the maximum sum of all range queries

return answer; // return the final answer

const modulo = 10 ** 9 + 7; // use modulo to avoid overflow

frequency[start]++;

nums.sort((a, b) => a - b);

Time and Space Complexity

frequency.sort((a, b) => a - b);

const lengthOfNums = nums.length; // total number of elements in nums

const frequency = new Array(lengthOfNums).fill(0); // array to keep track of frequency of each index

Time Complexity The time complexity of the code can be broken down as follows:

3. Prefix sum of d array: This for loop iterates n-1 times, updating the d array with the cumulative frequency. This step takes 0(n) time.

1. Additional array d: This array is of size n, taking up O(n) space.

4. Sorting nums and d: Sorting an array of n elements takes 0(n log n) time. Since both nums and d are sorted, this step takes 2 * O(n log n) time, which simplifies to O(n log n).

1. Initializing the array d: It is created with n zeros, where n is the length of nums. This operation takes O(n) time.

The most time-consuming operation here is the sorting step. Therefore, the overall time complexity of the algorithm is 0(n log n)

2. Sorting nums and d: In Python, the sort method sorts the array in place, so no additional space is needed for this step beyond

2. Populating the d array with the frequency of each index being requested: The for loop runs for each request, and each request

updates two elements in d. The number of requests is the length of requests, let's say m. Hence, this step would take O(m) time.

- The space complexity of the code can be analyzed as follows:
- 3. Temporary variables used for calculations and the sum operation (mod, loop variables, etc.) take 0(1) space.

5. Calculating the sum product of nums and d: The zip operation iterates through both arrays once, so this takes O(n) time.

the input arrays.

due to the sorting of nums and d.

Space Complexity

Therefore, the additional space used by the algorithm is for the d array, which gives us a space complexity of O(n).