368. Largest Divisible Subset Medium Array Sorting Math **Dynamic Programming Leetcode Link**

Problem Description Given a non-empty set of nums containing distinct positive integers, the task is to find the largest subset of these integers such that

This means for any two numbers in the subset, either one will be divisible by the other without leaving a remainder. The subset containing the highest number of such numbers needs to be identified. If more than one subset fits this requirement, any of them may be returned as the solution. Intuition

each pair of numbers in the subset, (answer[i], answer[j]), is in a relationship where one of the numbers is a multiple of the other.

The first insight is that if we sort the list of numbers nums in ascending order, any larger number can only be divisible by a smaller number and not vice versa. This imposes an order on potential divisibility relationships.

build the largest group (subset) of numbers where each pair of numbers meets the divisibility condition.

To solve this problem, we need an efficient way to determine relationships between numbers in terms of divisibility, and we want to

The second insight is that we can use Dynamic Programming to build up knowledge of the largest divisible subset up to any given index as we iterate through our sorted list. We define an array f[i] to represent the size of the largest divisible subset that includes nums [i] as the largest number in the subset. So for each number nums [i], we look back at all previous numbers nums [j] where j < i

and update f[i] if nums[i] % nums[j] == 0. Lastly, knowing the size of the largest subset isn't enough; we want the subset itself. We keep track of the index k at which we attain the maximum size of the subset. Once we finish populating our f array, we can backtrack from nums [k] and construct the actual subset by looking at elements that could be used to reach nums [k]. To construct the result, we traverse in reverse, starting from

nums [k] going backward, and add numbers to our subset ans that have the correct divisibility property and help us reach the

previously calculated optimum size m at each step until we've constructed the full largest subset. **Solution Approach** The solution implements Dynamic Programming, which is a method to solve problems by combining the solutions of subproblems. Let's walk through the code and explain the solution's approach:

1 nums.sort()

by nums[j] and not vice versa.

1. First, the list nums is sorted so we can guarantee that for any pair (nums[i], nums[j]) where i > j, nums[i] can only be divisible

2. The variable n is initialized to the length of nums, and an array f of the same length is created with all elements set to 1. This array will hold the size of the largest divisible subset ending with nums [i].

1 n = len(nums)2 f = [1] * n

3. Two variables, k and m, are used. k is the index at which the largest subset ends, and m will hold the size of the largest subset.

- - 4. We iterate through each element nums [i] from start to end, and for each i, we iterate backwards from i 1 to 0 to check all possible nums[j] that nums[i] could be divisible by. If nums[i] % nums[j] is 0, meaning nums[j] divides nums[i] without a remainder, we update f[i] if it would lead to a larger subset size than currently recorded at f[i].
- f[i] = max(f[i], f[j] + 1)5. While updating f[i], we also keep track of the maximum subset size we've seen so far and the corresponding index k.

1 m = f[k]

if f[k] < f[i]:

subset size).

i -= 1

Example Walkthrough

ans.append(nums[i])

k, m = i, m - 1

k = i

1 for i in range(n):

for j in range(i):

if nums[i] % nums[j] == 0:

6. Now the variable m is assigned to the length of the largest subset.

1 k = 0

7. To re-construct the actual subset, we initialize an empty list ans and loop backwards from the index k of the largest number belonging to the largest divisible subset. We decrement meach time we add a new element to ans. For each element we consider, it must satisfy two conditions: nums[k] % nums[i] == 0 (divisibility) and f[i] == m (it contributes to the maximum

8. Finally, ans is returned, which contains the elements of nums forming the largest divisible subset.

1. Sort the array nums. In this example, nums is already sorted in ascending order: [1, 2, 4, 8].

including each number individually is just the number itself.

effectively avoiding redundant calculations and leading to a more efficient algorithm.

Sort the array to ensure divisibility can be checked in ascending order

Build up the dp array with the length of each longest divisible subset

Initialize a list to keep track of the longest subset ending at each index

max_index = 0 # To track the index at which the largest divisible subset ends

Check if the current number is divisible by the j-th number

If the current number divides the number at max_index and its dp value

Update the dp[i] to the maximum length achievable

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

8) and the maximum subset size m = f[k] = 4.

2 ans = []while m: if nums[k] % nums[i] == 0 and f[i] == m:

1 i = k

1 return ans

8].

array.

The use of sorting, coupled with dynamic programming and some clever backtracking, enables this solution to find the largest subset efficiently. The Dynamic Programming pattern here helps avoid redundant re-computation by storing optimal sub-solutions in the array f.

Let's illustrate the solution approach with an example. Consider the following small set of distinct positive integers: nums = [1, 2, 4,

2. Initialize n = len(nums) which is 4 in our case, and f = [1, 1, 1, 1], corresponding to the fact that the largest divisible subset

4. Iterate through each element nums[i]. We check all previous elements nums[j] to find if nums[i] can be made larger by including nums[j]. As 1 divides every other integer, f will become [1, 2, 3, 4] by the end. 5. During iteration, we keep track of the maximum subset size and the index k. After completion, k = 3 (corresponding to number

3. Initialize k = 0 to keep track of the index of the largest subset, although its value will likely change as we iterate through the

(nums[k] % nums[i] == 0) and if f[i] equals m. We find that all elements in nums satisfy this, so sequentially, ans becomes [8],

class Solution:

8

10

11

12

13

14

15

16

17

23

24

25

26

27

28

29

30

31

32

38

39

40

41

42

43

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

nums.sort()

n = len(nums)

dp = [1] * n

for i in range(n):

for j in range(i):

subset_size = dp[max_index]

current_index = max_index

Move one index backwards

current_index -= 1

return largest_subset

largest_subset = []

while subset_size:

if nums[i] % nums[j] == 0:

The size of the largest divisible subset

List to store the largest divisible subset

Start from the end element of the largest subset

Backtrack from the largest index found to build the subset

Return the largest divisible subset in the original order

// If nums[i] is divisible by nums[j], update dp[i]

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

// Update maxIndex if a larger subset is found

if (nums[i] % nums[j] == 0) {

if (dp[maxIndex] < dp[i]) {</pre>

// Size of the largest divisible subset

// List to store the largest divisible subset

List<Integer> result = new ArrayList<>();

maxIndex = i;

int subsetSize = dp[maxIndex];

[8, 4], [8, 4, 2], and finally [8, 4, 2, 1]. 7. The resulting subset, which is the largest one where every pair of elements are divisible by one another, is [8, 4, 2, 1].

By following these steps, the solution capitalizes on the sorted property of the array and the previously computed subset sizes,

6. To reconstruct the subset, we start with ans = [] and loop backwards from k = 3. As we check nums [i] we look for divisibility

- Python Solution from typing import List
- 18 dp[i] = max(dp[i], dp[j] + 1)# Update the index of the largest divisible subset if a new max is found 19 20 if dp[max_index] < dp[i]:</pre> 21 $max_index = i$ 22

```
33
               # corresponds to the current subset size, add it to the result
34
               if nums[max_index] % nums[current_index] == 0 and dp[current_index] == subset_size:
35
                   largest_subset.append(nums[current_index])
36
                   # Update the max_index and decrement the size for next numbers
37
                   max_index, subset_size = current_index, subset_size - 1
```

```
Java Solution
  import java.util.Arrays;
   import java.util.ArrayList;
   import java.util.List;
   class Solution {
 6
       /**
        * Finds the largest divisible subset in the given array.
9
10
        * @param nums The array of numbers
        * @return The largest divisible subset
11
12
        */
       public List<Integer> largestDivisibleSubset(int[] nums) {
13
           // Sort the array of numbers
14
15
           Arrays.sort(nums);
16
17
           // Length of the nums array
           int length = nums.length;
18
19
20
           // Array to store the size of the largest divisible subset
           // that ends with nums[i]
22
           int[] dp = new int[length];
23
24
           // Initialize all values in dp to 1
25
           Arrays.fill(dp, 1);
26
27
           // Variable to track the index of the largest element of the subset
28
           int maxIndex = 0;
29
30
           // Dynamic programming to fill the dp array
31
            for (int i = 0; i < length; ++i) {</pre>
                for (int j = 0; j < i; ++j) {
32
```

50 // Construct the result list by going backwards from maxIndex 51 for (int i = maxIndex; subsetSize > 0; --i) { 52 53 // If the current number can be included in the subset 54 if (nums[maxIndex] % nums[i] == 0 && dp[i] == subsetSize) {

```
55
                    result.add(nums[i]);
                   maxIndex = i; // Update the maxIndex to the current number's index
56
57
                   --subsetSize; // Decrease the size of the subset as we add to the result
58
59
60
61
           return result;
62
63 }
64
C++ Solution
1 class Solution {
2 public:
       vector<int> largestDivisibleSubset(vector<int>& nums) {
           // Sort the input array to make the divisibility condition easier to check.
           sort(nums.begin(), nums.end());
           // Get the size of the array.
           int n = nums.size();
9
           // Array to store the size of the largest divisible subset that ends with nums[i].
10
           vector<int> subsetSizes(n, 1);
11
12
           // Variable to keep track of the index at which the largest subset ends.
13
           int maxSubsetIndex = 0;
14
15
           // Calculate the size of the largest subset where each element is divisible by its previous ones.
           for (int i = 0; i < n; ++i) {
16
                for (int j = 0; j < i; ++j) {
17
18
                    if (nums[i] % nums[j] == 0) {
                       // If nums[i] is divisible by nums[j], consider this as a potential maximum size.
19
20
                        subsetSizes[i] = max(subsetSizes[i], subsetSizes[j] + 1);
21
22
23
               // Update the index of the largest subset if the current one is larger.
24
               if (subsetSizes[maxSubsetIndex] < subsetSizes[i]) {</pre>
25
                   maxSubsetIndex = i;
26
27
28
29
           // Construct the largest subset by iterating from the end to the beginning of the subset.
           int currentSize = subsetSizes[maxSubsetIndex];
30
31
           vector<int> largestSubset;
            for (int i = maxSubsetIndex; currentSize > 0; --i) {
32
33
               // If nums[i] is part of the subset, add it to the result.
               if (nums[maxSubsetIndex] % nums[i] == 0 && subsetSizes[i] == currentSize) {
34
                    largestSubset.push_back(nums[i]);
35
36
                   // Update the index to the last added number and decrement the currentSize.
37
                   maxSubsetIndex = i;
38
                   --currentSize;
39
40
           // Return the constructed largest divisible subset.
41
           return largestSubset;
42
43
44 };
45
```

27 // Update the index of the largest subset if the current one is larger. 28 if (subsetSizes[maxSubsetIndex] < subsetSizes[i]) {</pre> 29 30 maxSubsetIndex = i; 31

Typescript Solution

nums = sort(nums);

// Get the size of the array.

const n: number = nums.length;

let maxSubsetIndex: number = 0;

for (let i = 0; i < n; ++i) {</pre>

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

const largestSubset: number[] = [];

maxSubsetIndex = i;

--currentSize;

return largestSubset.reverse();

Time and Space Complexity

if (nums[i] % nums[j] === 0) {

let currentSize: number = subsetSizes[maxSubsetIndex];

for (let i = maxSubsetIndex; currentSize > 0; --i) {

largestSubset.push(nums[i]);

// If nums[i] is part of the subset, add it to the result.

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

1 // Function to sort an array of numbers in ascending order.

const max = (a: number, b: number): number => (a > b ? a : b);

function largestDivisibleSubset(nums: number[]): number[] {

const subsetSizes: number[] = new Array(n).fill(1);

4 // Function to retrieve the maximum of two numbers.

2 const sort = (array: number[]): number[] => array.sort((a, b) => a - b);

7 // Function to get the largest divisible subset from an array of numbers.

// Sort the input array to make the divisibility condition easier to check.

// Variable to keep track of the index at which the largest subset ends.

// Array to store the size of the largest divisible subset that ends with nums[i].

subsetSizes[i] = max(subsetSizes[i], subsetSizes[j] + 1);

// Construct the largest subset by iterating from the end to the beginning of the subset.

if (nums[maxSubsetIndex] % nums[i] === 0 && subsetSizes[i] === currentSize) {

// Update the index to the last added number and decrement the currentSize.

// Return the constructed largest divisible subset in reverse to maintain original order.

// Calculate the size of the largest subset where each element is divisible by its previous ones.

// If nums[i] is divisible by nums[j], consider this as a potential maximum size.

```
Time Complexity
The time complexity of the code is determined by the nested loops and the operations within them.
  • The outer loop runs n times where n is the number of elements in nums.
```

• The inner loop, for each element of the outer loop, runs a maximum of i times which is less than n.

Therefore, in total, it has a complexity of approximately O(n^2) due to the double-loop structure.

- The while loop at the end of the code runs a maximum of n times corresponding to the size of the nums list. The operations inside are of constant time. However, this does not affect the overall time complexity since it's linear and the dominant factor is the O(n^2) from the nested loops. Hence the overall time complexity remains O(n^2).
- **Space Complexity**

• The f list which is of size n contributes O(n) to the space complexity.

contributes O(n) to the space complexity.

- The space complexity of the code is determined by the additional storage used:
- The variables k, i, and m use constant space.

• The ans list could potentially reach the same size as nums in the case that all elements are multiples of each other. Thus, it also

Therefore, the overall space complexity is O(n) as it relies on the storage required for the f and ans lists, which scale linearly with the size of the input nums.