368. Largest Divisible Subset

Medium Array Math Dynamic Programming Sorting

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 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. 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

To solve this problem, we need an efficient way to determine relationships between numbers in terms of divisibility, and we want to build the largest group (subset) of numbers where each pair of numbers meets the divisibility condition.

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.

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. 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 by `nums[j]` and not vice versa.

```
nums.sort()
```

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]`.

```
n = len(nums)
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.

```
k = 0
```

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]`.

```
for i in range(n):
    for j in range(i):
        if nums[i] % nums[j] == 0:
            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`.

```
    if f[k] < f[i]:
        k = i
```

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

```
m = f[k]
```

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 `m` each 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 subset size).

```
i = k
ans = []
while m:
    if nums[k] % nums[i] == 0 and f[i] == m:
        ans.append(nums[i])
        k, m = i, m - 1
    i -= 1
```

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

```
return ans
```

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`.

Example Walkthrough

Let's illustrate the solution approach with an example. Consider the following small set of distinct positive integers: `nums = [1, 2, 4, 8]`.

1. Sort the array `nums`. In this example, `nums` is already sorted in ascending order: `[1, 2, 4, 8]`.
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 including each number individually is just the number itself.
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 array.
4. Iterate through each element `nums[i]`. We check 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 `8`) and the maximum subset size `m = f[k] = 4`.
6. To reconstruct the subset, we start with `ans = []` and loop backwards from `k = 3`. As we check `nums[i]` we look for divisibility (`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]`, `[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, effectively avoiding redundant calculations and leading to a more efficient algorithm.

Python Solution

```python
from typing import List

class Solution:
    def largestDivisibleSubset(self, nums: List[int]) -> List[int]:
        # Sort the array to ensure divisibility can be checked in ascending order
        nums.sort()
        n = len(nums)
        # Initialize a list to keep track of the longest subset ending at each index
        dp = [1] * n
        max_index = 0 # To track the index at which the largest divisible subset ends

        # Build up the dp array with the length of each largest divisible subset
        for i in range(n):
            for j in range(i):
                # Check if the current number is divisible by the j-th number
                if nums[i] % nums[j] == 0:
                    # Update the dp[i] to the maximum length achievable
                    dp[i] = max(dp[i], dp[j] + 1)
            # Update the index of the largest divisible subset if a new max is found
            if dp[max_index] < dp[i]:
                max_index = i

        # The size of the largest divisible subset
        subset_size = dp[max_index]
        # Initialize the result list for storing the largest subset
        current_num = nums[max_index]
        # List to store the largest divisible subset
        largest_subset = []

        # Backtrack from the largest index found to build the subset
        while subset_size > 0:
            # If the current number divides the number at max_index and its dp value
            # corresponds to the current subset size, add it to the result
            if max_index == 0 or current_num % nums[max_index] == 0 and dp[max_index] == subset_size:
                largest_subset.append(current_num) # Add the number to the subset
                # Update the max_index and decrement the size for next numbers
                max_index, subset_size = current_index, subset_size - 1
                # Move one index backwards
                current_num = nums[max_index]

        # Return the largest divisible subset in the original order
        return largest_subset
```

Java Solution

```java
import java.util.Arrays;
import java.util.ArrayList;
import java.util.List;

class Solution {
    /**
     * Finds the largest divisible subset in the given array.
     *
     * @param nums The array of numbers
     * @return The largest divisible subset
     */
    public List<Integer> largestDivisibleSubset(int[] nums) {
        // Sort the array of numbers
        Arrays.sort(nums);

        // Length of the nums array
        int length = nums.length;

        // Array to store the size of the largest divisible subset
        // that ends with nums[i]
        int[] dp = new int[length];
        Arrays.fill(dp, 1);

        // Initializing all values in dp to 1
        Arrays.fill(dp, 1);

        // Variable to track the index of the largest element of the subset
        int maxIndex = 0;

        // Dynamic programming to fill the dp array
        for (int i = 0; i < length; ++i) {
            for (int j = 0; j < i; ++j) {
                // If nums[i] is divisible by nums[j], update dp[i]
                if (nums[i] % nums[j] == 0) {
                    dp[i] = Math.max(dp[i], dp[j] + 1);
                }
            }

            // Update maxIndex if a larger subset is found
            if (dp[maxIndex] < dp[i]) {
                maxIndex = i;
            }
        }

        // Size of the largest divisible subset
        int subsetSize = dp[maxIndex];

        // List to store the largest divisible subset
        List<Integer> result = new ArrayList<>();

        // Construct the result list by going backwards from maxIndex
        for (int i = maxIndex; subsetSize > 0; --i) {
            // If the current number can be included in the subset
            if (nums[maxIndex] % nums[i] == 0 && dp[i] == subsetSize) {
                result.add(nums[i]);
                maxIndex = i; // Update the maxIndex to the current number's index
                --subsetSize; // Decrease the size of the subset as we add to the result
            }
        }

        return result;
    }
}
```

C++ Solution

```cpp
class Solution {
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();
        // Array to store the size of the largest divisible subset that ends with nums[i].
        vector<int> subsetSizes(n, 1);

        // Variable to keep track of the index at which the largest subset ends.
        int maxSubsetIndex = 0;

        // Calculate the size of the largest subset where each element is divisible by its previous ones.
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < i; ++j) {
                // If nums[i] is divisible by nums[j], consider this as a potential maximum size.
                if (nums[i] % nums[j] == 0) {
                    subsetSizes[i] = max(subsetSizes[i], subsetSizes[j] + 1);
                }
            }
            // Update the index of the largest subset if the current one is larger.
            if (subsetSizes[maxSubsetIndex] < subsetSizes[i]) {
                maxSubsetIndex = i;
            }
        }

        // Construct the largest subset by iterating from the end to the beginning of the subset.
        int currentSize = subsetSizes[maxSubsetIndex];
        vector<int> largestSubset;
        for (int i = maxSubsetIndex; currentSize > 0; --i) {
            // If nums[i] is part of the subset, add it to the result.
            if (nums[maxSubsetIndex] % nums[i] == 0 && subsetSizes[i] == currentSize) {
                largestSubset.push_back(nums[i]);
                // Update the index to the last added number and decrement the currentSize.
                maxSubsetIndex = i;
                --currentSize;
            }
        }

        // Return the constructed largest divisible subset.
        return largestSubset;
    }
};
```

Typescript Solution

```typescript
// Function to sort an array of numbers in ascending order.
const sort = (array: number[]): number[] => array.sort((a, b) => a - b);

// Function to retrieve the maximum of two numbers.
const max = (a: number, b: number): number => (a > b ? a : b);

// Function to get the largest divisible subset from an array of numbers.
function largestDivisibleSubset(nums: number[]): number[] {
    // Sort the input array to make the divisibility condition easier to check.
    nums = sort(nums);

    // Get the size of the array.
    const n = nums.length;
    // Array to store the size of the largest divisible subset that ends with nums[i].
    const subsetSizes = new Array(n).fill(1);

    // Variable to keep track of the index at which the largest subset ends.
    let maxSubsetIndex = 0;

    // Calculate the size of the largest subset where each element is divisible by its previous ones.
    for (let i = 0; i < n; ++i) {
        for (let j = 0; j < i; ++j) {
            // If nums[i] is divisible by nums[j], consider this as a potential maximum size.
            if (nums[i] % nums[j] === 0) {
                subsetSizes[i] = max(subsetSizes[i], subsetSizes[j] + 1);
            }
        }
        // Update the index of the largest subset if the current one is larger.
        if (subsetSizes[maxSubsetIndex] < subsetSizes[i]) {
            maxSubsetIndex = i;
        }
    }

    // Construct the largest subset by iterating from the end to the beginning of the subset.
    let currentSize = subsetSizes[maxSubsetIndex];
    const largestSubset = [];
    for (let i = maxSubsetIndex; currentSize > 0; --i) {
        // If nums[i] is part of the subset, add it to the result.
        if (nums[maxSubsetIndex] % nums[i] === 0 && subsetSizes[i] === currentSize) {
            largestSubset.push(nums[i]);
            // Update the index to the last added number and decrement the currentSize.
            maxSubsetIndex = i;
            --currentSize;
        }
    }

    // Return the constructed largest divisible subset in reverse to maintain original order.
    return largestSubset.reverse();
}
```

Time and Space Complexity

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 space complexity of the code is determined by the additional storage used:

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

The variables `k`, `i`, and `m` use constant space.

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`.