368. Largest Divisible Subset

Sept. 1, 2019 | 18K views

Previous Next

Average Rating: 4.85 (20 votes)

Given a set of **distinct** positive integers, find the largest subset such that every pair (S_i, S_j) of elements in this subset satisfies:

S_i % S_j = 0 or S_j % S_i = 0.

If there are multiple solutions, return any subset is fine.

Example 1:

```
Input: [1,2,3]
Output: [1,2] (of course, [1,3] will also be ok)
```

Example 2:

```
Input: [1,2,4,8]
Output: [1,2,4,8]
```

Solution

Mathematics

Before elaborating the solutions, we give some corollaries that one can derive from the property of modulo operation, which would prove useful later to solve the problem.

Given a list of values `[E, F, G]` sorted in ascending order (*i.e.* `E < F < G`), and the list itself forms a divisible subset as described in the problem, then we could extend the subset without ***enumerating*** all numbers in the subset, in the following two cases:

- **Corollary I:** For any value that can be divided by the *largest* element in the divisible subset, by adding the new value into the subset, one can form another divisible subset, *i.e.* for all `h`, if `h % G == 0`, then `[E, F, G, h]` forms a new divisible subset.
- **Corollary II:** For all value that can divide the *smallest* element in the subset, by adding the new value into the subset, one can form another divisible subset, *i.e.* for all `d`, if `E % d == 0`, then `[d, E, F, G]` forms a new divisible subset.

The above two corollaries could help us to structure an efficient solution, since it suffices to have just one comparison in order to extend the subset.

Approach 1: Dynamic Programming

Intuition

At the first glance, the problem might seem to be similar with those combination problems such as two sum and 3sum. Indeed, like those combinations problems, it turned out to be rather helpful to ***sort*** the original list first, which would help us to reduce the number of enumerations at the end.

As another benefit of sorting the original list, we would be able to apply the mathematical corollaries explained at the beginning of the article.

So first of all, we sort the original list. And as it turns out, this is another dynamic programming problem. The key of solving a dynamic programming problem is to formulate the problem in a *recursive* and *sound* way.

Here is our attempt, which you would see some theoretical supports later.

> For an ordered list $[X_1, X_2, ...X_n]$, we claim that the *largest* divisible subset from this list is the largest subset among all possible divisible subsets that **end with** each of the number in the list.

Let us define a function named $\text{EDS}(X_i)$, which gives the largest divisible subset that ends with the number X_i. By "*ends with*", we mean that the number X_i should be the largest number in the subset. For example, given the list $[2, 4, 7, 8]$, let us calculate $\text{EDS}(4)$ by enumeration. First, we list all divisible subsets that ends with 4, which should be $\{4\}$ and $\{2, 4\}$. Then by definition, we have $\text{EDS}(4) = \{2, 4\}$. Similarly, one can obtain that $\text{EDS}(2) = \{2\}$ and $\text{EDS}(7) = \{7\}$.

Note: a single number itself forms a divisible subset as well, though it might not be clearly stated in the problem statement.

Finally let us define our target function that gives the largest divisible subset from an order list $[X_1, X_2, ...X_n]$, as $\text{LDS}([X_1, X_2, ...X_n])$. Now, without further due, we claim that the following equation should hold:

$$\text{LDS}([X_1, X_2, ...X_n]) = \max\left(\forall\ \text{EDS}(X_i)\right),\ 1 \leq i \leq n \qquad (1)$$

We could prove the above formula literally *by definition*. First of all, $\forall\ \text{EDS}(X_i)$ cover the divisible subsets in all cases (*i.e.* subsets ends with X_i). And then we pick the largest one among the largest subsets.

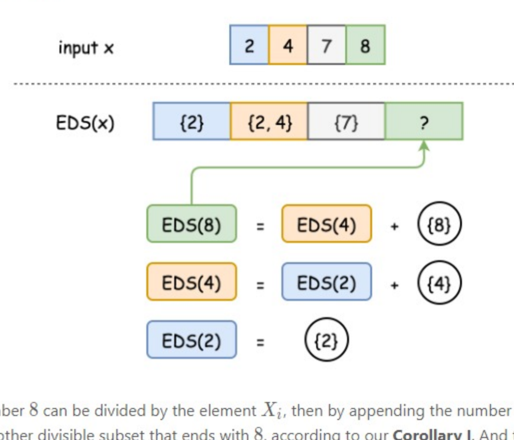
> Furthermore, we could calculate the function $\text{EDS}(X_i)$ recursively, with the corollaries we defined at the beginning of the article.

Algorithm

Let us explain the algorithm on how to calculate $\text{EDS}(X_i)$, following the above example with the list of $[2, 4, 7, 8]$. As a reminder, previously we have obtained the $\text{EDS}(X_i)$ value for all elements less than 8, *i.e.*:

$$\text{EDS}(2) = \{2\} \quad \text{EDS}(4) = \{2, 4\} \quad \text{EDS}(7) = \{7\}$$

To obtain $\text{EDS}(8)$, we simply enumerate all elements before 8 and their $\text{EDS}(X_i)$ values respectively, with the following procedure:



- If the number 8 can be divided by the element X_i, then by appending the number 8 to $\text{EDS}(X_i)$ we obtain another divisible subset that ends with 8, according to our **Corollary I**. And this new subset stands as a potential value for $\text{EDS}(8)$. For example, since $8 \bmod 2 == 0$, therefore $\{2, 8\}$ could be the final value for $\text{EDS}(8)$, and similarly with the subset $\{2, 4, 8\}$ obtained from $\text{EDS}(4)$
- If the number 8 can NOT be divided by the element X_i, then we could be sure that the value of $\text{EDS}(X_i)$ would not contribute to $\text{EDS}(8)$, according to the definition of divisible subset. For example, the subset $\text{EDS}(7) = \{7\}$ has no impact for $\text{EDS}(8)$.
- We then pick the largest new subsets that we form with the help of $\text{EDS}(X_i)$. Particularly, the subset $\{8\}$ stands for a valid candidate for $\text{EDS}(8)$. And in a hypothetical case where 8 cannot be divided by any of its previous elements, we would have $\text{EDS}(8) = \{8\}$.

Here we give some sample implementation based on the above idea. *Note:* for the Python implementation, we showcase the one from StefanPochmann for its conciseness and efficiency.

Java Python Copy

```python
class Solution(object):
    def largestDivisibleSubset(self, nums):
        """
        :type nums: List[int]
        :rtype: List[int]
        """
        # The container that holds all intermediate solutions.
        # key: the largest element in a valid subset.
        subsets = {-1: set()}

        for num in sorted(nums):
            subsets[num] = max([subsets[k] for k in subsets if num % k == 0], key=len) | {num}

        return list(max(subsets.values(), key=len))
```

Complexity Analysis

- Time complexity : $\mathcal{O}(N^2)$. In the major loop of the algorithm, we need to calculate the $\text{EDS}(X_i)$ for each element in the input list. And for each $\text{EDS}(X_i)$ calculation, we need to enumerate all elements before X_i. As a result, we end up with the $\mathcal{O}(N^2)$ time complexity.
- Space complexity : $\mathcal{O}(N^2)$. We maintain a container to keep track of $\text{EDS}(X_i)$ value for each element in the list. And in the worst case where the entire list is a divisible set, the value of $\text{EDS}(X_i)$ would be the sublist of $[X_1, X_2...X_i]$. As a result, we end up with the $\mathcal{O}(N^2)$ space complexity.

Approach 2: Dynamic Programming with Less Space

Intuition

Following the same intuition of dynamic programming, however we could do a bit better on the *space complexity*.

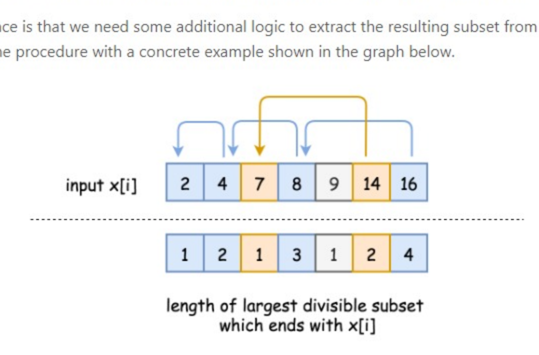
Instead of keeping the largest divisible subset for each of the input elements, *i.e.* $\text{EDS}(X_i)$, we could simply record its size, namely $\text{size}(\text{EDS}(X_i))$. As a result, we reduce the space complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$.

In exchange, we need to reconstruct the largest divisible subset at the end, which is a tradeoff between time and space.

Algorithm

The main algorithm remains almost the same as the Approach #1, which includes calculating **the size** of the largest divisible subset that ends with each element X_i. We denote this resulting vector as `dp[i]`.

The difference is that we need some additional logic to extract the resulting subset from `dp[i]`. Here we elaborate the procedure with a concrete example shown in the graph below.



In the upper part of the graph, we have a list of elements (X_i) sorted in ascending order. And in the lower part of the graph, we have the size value of the largest divisible subset that ends with each element X_i.

To facilitate the reading, we draw a link between each element X_i with its neighbor element in its largest divisible subset. For example, for the element $X_i = 8$, the resulting largest divisible subset that ends with 8 would be $\{2, 4, 8\}$, and we see a link between 8 and its neighbor 4.

- The reconstruction of the resulting subset begins with finding the largest size (*i.e.* 4) and its index in `dp[i]`. The resulting largest divisible subset would end with the element who has the max value of `dp[i]`.
- Then starting from the index of the largest size, we run a loop to go backwards (from X_i to X_1) to find the next element that should be included in the resulting subset.
- We have two criteria to determine this next element: (1). the element should be able to divide the previous tail element in the resulting subset, *e.g.* $16 \bmod 8 == 0$. (2). The value of `dp[i]` should correspond to the current size of the divisible subset, *e.g.* the element 7 would NOT be the next neighbor element after the element 8, since their `dp[i]` values do not match up.

Java Python Copy

```python
class Solution(object):
    def largestDivisibleSubset(self, nums):
        """
        :type nums: List[int]
        :rtype: List[int]
        """
        if len(nums) == 0:
            return []

        # important step !
        nums.sort()

        # The container that keep the size of the largest divisible subset that ends with X_i
        # dp[i] corresponds to len(EDS(X_i))
        dp = [0] * (len(nums))

        """ Build the dynamic programming matrix/vector """
        for i, num in enumerate(nums):
            maxSubsetSize = 0
            for k in range(0, i):
                if nums[i] % nums[k] == 0:
                    maxSubsetSize = max(maxSubsetSize, dp[k])

            maxSubsetSize += 1
            dp[i] = maxSubsetSize

        """ Find both the size of largest divisible set and its index """
```

Complexity Analysis

- Time complexity : $\mathcal{O}(N^2)$. The additional logic to reconstruct the resulting subset would take us an extra $\mathcal{O}(N)$ time complexity which is inferior to the main loop that takes $\mathcal{O}(N^2)$.
- Space complexity : $\mathcal{O}(N)$. The vector `dp[i]` that keeps tracks of the size of the largest divisible subset which ends with each of the elements, would take $\mathcal{O}(N)$ space in all cases.

Approach 3: Recursion with Memoization

Intuition

A typical code pattern for dynamic programming problems is to maintain a matrix or vector of intermediate solutions, and having one or two loops that traverse the matrix or vector. During the loop, we reuse the intermediate solutions instead of recalculating them at each occasion.

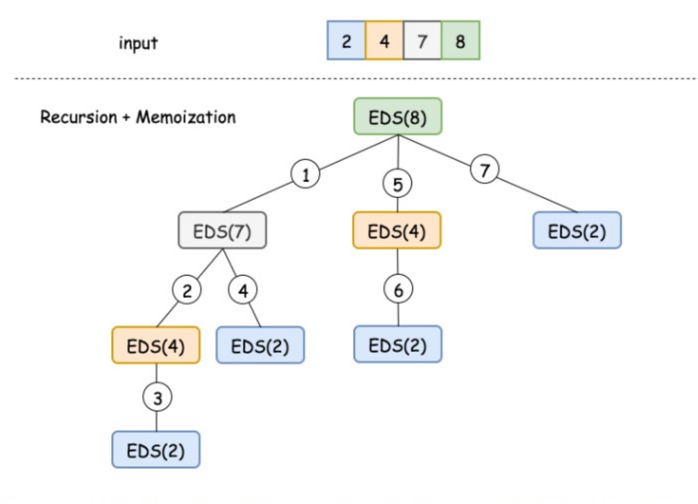
As one might notice, the above code pattern reminds us the techniques of recursion with memoization. Indeed, there is more than one way to implement the solution with the methodology of dynamic programming.

Algorithm

Once we figure out the essence of the problem, as summarized in the **Formula (1)**, it might be more intuitive to implement it via recursion with memoization.

Here we highlight the importance of applying **memoization** together with the recursion, in this case.

Following the same example in Approach #1, we draw the call graph below for the calculation of $\text{EDS}(8)$, where each node represents the invocation of the function $\text{EDS}(X_i)$ and on the edge we indicate the sequence of the invocations.



As one can see, if we do not keep the intermediate results, the number of calculation would grow exponentially with the length of the list.

Java Python Copy

```python
class Solution:
    def largestDivisibleSubset(self, nums: List[int]) -> List[int]:

        def EDS(i):
            """ recursion with memoization """
            if i in memo:
                return memo[i]

            tail = nums[i]
            maxSubset = []
            # The value of EDS(i) depends on it previous elements
            for p in range(0, i):
                if tail % nums[p] == 0:
                    subset = EDS(p)
                    if len(maxSubset) < len(subset):
                        maxSubset = subset

            # extend the found max subset with the current tail.
            maxSubset = maxSubset.copy()
            maxSubset.append(tail)

            # memorize the intermediate solutions for reuse.
            memo[i] = maxSubset
            return maxSubset

        # test case with empty set
        if len(nums) == 0: return []
```

Complexity Analysis

- Time complexity : $\mathcal{O}(N^2)$.
  - In the above implementation, we adopt the bottom-up strategy where we first calculate the $\text{EDS}(X_i)$ for elements with lower index. Through the memoization technique, the latter $\text{EDS}(X_i)$ calculation could reuse the intermediate ones. As a result, we reach the same time complexity as in the Approach #1.
- Space complexity : $\mathcal{O}(N^2)$.
  - In this implementation, we decide to keep the subset instead of its size, as the intermediate solutions. As we discussed in previous approaches, this would lead to the $\mathcal{O}(N^2)$ space complexity, with the worst scenario where the entire input list being a divisible set.
  - In addition, due to the use of recursion, the program would incur a maximum $\mathcal{O}(N)$ space for the call stacks, during the invocation of $\text{EDS}(X_n)$ for the last element in the ordered list. Though, overall the space complexity remains as $\mathcal{O}(N^2)$.

Rate this article: ★★★★★