1262. Greatest Sum Divisible by Three

Medium | Greedy | Array | Dynamic Programming | Sorting | Leetcode Link

Problem Description

The problem presents a situation where we are given an array of integers, `nums`. Our task is to find the maximum sum of elements selected from this array, with the condition that the sum must be divisible by 3. Not all elements of the array are required to be included in the sum, meaning we can select a subset of the array to achieve our goal, and this subset can even be empty.

The value we are looking for is the largest total we can assemble from the numbers in the array that, when divided by 3, does not leave a remainder. This means the total sum % 3 should equal 0. The challenge comes from determining which elements should be chosen to maximize the sum while meeting the divisibility criterion, as the array may contain any integer, positive or negative.

Intuition

The intuition here is to recognize that the remainder when dividing by three can only be 0, 1, or 2. So for any set of numbers, the total sum can leave one of these three remainders. We can leverage this insight to iteratively build up possible sums and track the largest sum for each of the three remainder categories.

To arrive at the solution:

1. We initialize an array, `f`, with three values: `0, -inf, -inf`. This array holds the maximum sum that has a remainder of `0, 1, and 2` respectively when divided by 3. We start with `0` since an empty set has a sum divisible by 3, and `-infinity` for the others as a base case, indicating that there are no sums for those remainders yet.
2. We iterate through each number in the input array. For every number `x`, we create a copy of `f`, called `g`, to keep track of the new sums we compute in this iteration.
3. For each possible remainder value (`j = 0, 1, 2`), we consider two options: a) Exclude the current number `x` from the sum. This means the maximum sum for the remainder `j` remains `f[j]`. b) Include the current number `x` in the sum. To find which prior sum `x` should be added to for the correct remainder, we use the expression `(j - x) % 3`. Then we add `x` to that prior sum: `f[(j - x) % 3] + x`.
4. We select the larger of the two options from step 3 for each remainder category and update `g` with this value.
5. After considering the current number `x`, we replace `f` with the new values in `g` to reflect the updated maximum sums. This way, `f` always represents the best sums found so far for each remainder category.
6. Once we've considered all numbers, `f[0]` will hold the maximum sum divisible by 3.

This dynamic programming solution efficiently computes the largest sum by solving smaller subproblems iteratively and using past solutions to construct new ones.

Solution Approach

The given Python solution uses dynamic programming to keep track of the maximum sum we can achieve for each of the three possible remainders when dividing the sum by 3, as we iterate through the input array, `nums`. The algorithm progressively builds upon previously computed results to reach the final answer.

Algorithm and Data Structures:

1. We start by initializing a list, `f`, with three elements: `[0, float('-inf'), float('-inf')]`. This list will hold the maximum sums we can obtain that have a remainder of `0` (`f[0]`), `1` (`f[1]`), and `2` (`f[2]`) when divided by 3.
2. We iterate through each element `x` in the input array `nums`. In each iteration, we will decide whether to include `x` in our previous maximum sums to possibly get new maximum sums.
3. We create a copy of `f`, called `g`, before modifying `f`. This step is crucial because we want to consider every element `x` based on the previous state of `f` and without being affected by changes made in the current iteration.
4. Now for each possible remainder `(j)` in `[0, 1, 2]`, we consider both possibilities – taking the current element `x` or not taking it. We compute this using a loop:

```python
for j in range(3):
    g[j] = max(f[j], f[(j - x) % 3] + x)
```

   We take the maximum between: a) `f[j]`: The previous maximum sum that had a remainder of `j` when divided by three (representing the case where we do not add the current element `x`). b) `f[(j - x) % 3] + x`: The sum of the current element and the maximum sum that, when added to `x`, will result in a remainder of `j` after division by 3 (representing the case where we do add the current element `x`).

5. Finally, `f` is updated to `g` after the inner loop to store the new maximum sums for the current iteration.
6. After the loop over `nums` is complete, `f[0]` holds the maximum sum of elements that is divisible by 3, and since `f` is updated iteratively after considering each element in `nums`, it contains the global maximum.

Patterns and Techniques:

- The algorithm uses a bottom-up dynamic programming approach to break the problem down into smaller subproblems that depend on each other, to arrive at the solution to the larger problem.
- Initialization of the data list `f` with `-infinity` for non-zero remainders is used to indicate that initially, there are no sums that could produce these remainders.
- Modulo operation is used to keep track of the remainders and map them within the range [0, 1, 2] as per the constraints of the problem.
- The algorithm avoids recalculating sums for each subset by storing the maximum sum for each possible remainder, thus optimizing the runtime.

By maintaining the state of all subproblems and smartly building up to the answer, the given algorithm reaches the correct solution in an efficient manner.

Example Walkthrough

Let's go through an example to illustrate the solution approach using a small array. Suppose the input array is `nums = [3, 6, 5, 1, 8]`. We want to find the maximal sum of its elements that is divisible by 3.

1. We initialize `f` as `[0, float('-inf'), float('-inf')]`. This represents the maximal sums with remainders `0`, `1`, and `2` respectively when divided by 3. Initially, the only sum we have that is divisible by 3 is `0`, since we haven't included any elements yet.
2. We start iterating through `nums`. For `x = 3` (the first element), we create a copy of `f`, calling it `g`.
3. Performing the inner loop, we compare the existing sums in `f` with the sums we'd get by adding `x` to each.
4. For `j = 0`, `g[0] = max(f[0], f[(0 - 3) % 3] + 3) = max(0, 0 + 3) = 3`. For `j = 1`, `g[1] = max(f[1], f[(1 - 3) % 3] + 3) = max(float('-inf'), 0 + 3) = 3`. For `j = 2`, `g[2] = max(f[2], f[(2 - 3) % 3] + 3) = max(float('-inf'), 0 + 3) = 3`.

   After iterating with `x = 3`, `g` is `[3, 3, 3]`, so we update `f` to this.

5. We then move to `x = 6` and again make a copy of `f` to `g`. We proceed in a similar manner, updating `g[j]` with comparisons between `f[j]` and `f[(j - 6) % 3] + 6`. This step will end up leaving `f` unchanged as `6` is a multiple of `3` and adding it to any subsum in `f` will not increase their value because `f` is already optimal from the previous iteration.

   After iterating with `x = 6`, `f` is still `[3, 3, 3]`.

6. Moving to `x = 5`, making a copy of `f` to `g`, and performing the inner loop, we get:

   For `j = 0`, `g[0] = max(f[0], f[(0 - 5) % 3] + 5) = max(3, 3 + 5) = 8`. For `j = 1`, `g[1] = max(f[1], f[(1 - 5) % 3] + 5) = max(3, 3 + 5) = 8`. For `j = 2`, `g[2] = max(f[2], f[(2 - 5) % 3] + 5) = max(3, 8 + 5) = 13`.

   After iterating with `x = 5`, `f` becomes `[8, 8, 13]`.

7. For `x = 1`, we repeat the process and find that `f` can be updated to `[8, 14, 13]` after considering this number.
8. Lastly, for `x = 8`, we use the similar update rule and end up with `f` being `[18, 14, 13]` since we can add `8` to `f[1]`, previously `14`, and get a remainder of `0` when divided by `3`, which updates `f[0]`.
9. After considering all elements, the first element of `f` (`f[0]`) contains the maximum sum of elements that is divisible by `3`, which is `18`. That is the final answer.

By following these steps with our input array of `nums = [3, 6, 5, 1, 8]`, we have determined that the maximum sum we can obtain from its elements that is divisible by `3` is `18`. The sequence of the elements contributing to this sum can be `[3, 6, 1, 8]` or `[6, 5, 1, 8]`, both of which give us the sum of `18`.

Python Solution

```python
class Solution:
    def max_sum_div_three(self, nums: List[int]) -> int:
        # Initialize a list to store the maximum sums divisible by 0, 1, and 2 respectively.
        # We start with 0 for the sum divisible by 3 and negative infinity for others as placeholders.
        max_sums = [0, float('-inf'), float('-inf')]

        # Iterate over each number in the input list.
        for num in nums:
            # Create a copy of the current state of max_sums to calculate the new state.
            new_max_sums = max_sums.copy()

            # Update the new_max_sums for each of the three states (0, 1, and 2).
            for remainder in range(3):
                # Calculate the maximum between the current state and the new possible state obtained
                # by adding the current number `num` and adjusting for the new remainder.
                new_max_sums[remainder] = max(max_sums[remainder], max_sums[(remainder - num) % 3] + num)

            # Update max_sums to the state we've calculated for this iteration.
            max_sums = new_max_sums

        # Return the maximum sum that is divisible by 3.
        return max_sums[0]

```

Java Solution

```java
class Solution {
    public int maxSumDivThree(int[] nums) {
        // Initialize a variable representing an impossible negative value
        final int INFINITY = 1 << 30;

        // Initialize max sums for each remainder (0, 1, 2)
        // when dividing by 3. Start with 0 for remainder 0 and
        // negative infinity for remainders 1 and 2, meaning
        // they're initially not reachable.
        int[] maxSums = new int[] {0, -INFINITY, -INFINITY};

        // Loop through each number in the given nums array
        for (int num : nums) {
            // Clone current maxSums array to temporary array to store updates
            int[] newMaxSums = maxSums.clone();

            // For each possible remainder (0, 1, 2)
            for (int remainder = 0; remainder < 3; ++remainder) {
                // Update the maximum sum for the current remainder considering
                // the new number. The max sum is either the current max sum
                // without the new number or the max sum with remainder equal to
                // the difference between the current remainder and num % 3,
                // adjusted to be within the range [0, 2] using modulo, plus the new number.
                newMaxSums[remainder] = Math.max(maxSums[remainder],
                                                 maxSums[(remainder - num % 3 + 3) % 3] + num);
            }

            // Update maxSums array with the computed values for this iteration
            maxSums = newMaxSums;
        }
        // After processing all numbers, return the max sum that is divisible by 3,
        // which would be stored at index 0
        return maxSums[0];
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int maxSumDivThree(vector<int>& nums) {
        const int INF = 1 << 30;                          // Define a large number to represent infinity.
        vector<int> dp = {0, -INF, -INF};                 // Initialize dp array to store max sums for modulo 3 values (0, 1, 2).

        for (int x : nums) {                              // Iterate through each number in nums.
            vector<int> new_dp = dp;                      // Make a copy of the current state of dp.
            for (int j = 0; j < 3; ++j) {
                // Calculate the index for the updated sum in the original dp array.
                int idx = (j - x % 3 + 3) % 3;
                // Choose the maximum between not taking the current number (f[j])
                // and taking the current number (f[idx] + x)
                new_dp[j] = max(dp[j], dp[idx] + x);
            }
            dp = move(new_dp);                            // Update the dp array with the new values.
        }
        return dp[0];                                     // Return the maximum sum that is divisible by 3.
    }
};

```

Typescript Solution

```typescript
function maxSumDivThree(nums: number[]): number {
    // Define 'inf' as a large number to act as a placeholder for negative infinity.
    const inf = 1 << 30;
    // dp array will hold the maximum sum for modulo 0, 1, and 2.
    const dp: number[] = [0, -inf, -inf];

    // Loop through each number in the input array.
    for (const num of nums) {
        // Clone the current state of 'dp'.
        const nextState = [...dp];

        // Iterate over the 0, 1, and 2 possible sums.
        for (let i = 0; i < 3; ++i) {
            // Calculate the index for the updated modulo class after adding 'num'.
            // Make sure to stay within the bounds of the array by using the modulo operator.
            // Update the 'dp' entry if a better (larger) sum is found.
            dp[i] = Math.max(nextState[i], nextState[(i - (num % 3) + 3) % 3] + num);
        }
    }
    // The answer is the maximum sum divisible by 3, which is stored at index 0 after the loop.
    return dp[0];
}

```

Time and Space Complexity

The given Python code defines a function `maxSumDivThree` that calculates the maximum sum of a subsequence of the input list `nums` such that the sum is divisible by 3.

Time Complexity

The time complexity of the code can be determined by looking at the number of iterations and the operations performed within each iteration. We iterate over each element in `nums` exactly once, where `nums` length is `n`. Inside this loop, we perform constant time operations for each of the three possible remainders when an element is divided by 3. Therefore, the time complexity is `O(n)`, where `n` is the length of the input list `nums`.

Space Complexity

The space complexity is determined by the additional space used by the algorithm beyond the input size. We have an array `f` of fixed size 3, and a temporary array `g` also of fixed size 3. These arrays don't scale with the size of the input; they are used for storing remainders with respect to division by 3. Hence, the space complexity is `O(1)`, which means it uses constant additional space regardless of the size of the input.