# 1363. Largest Multiple of Three

Hard Greedy Array Dynamic Programming Leetcode Link

## Problem Description

The problem requires creating the largest possible multiple of three from a given array of single-digit numbers (`digits`). The solution must concatenate elements from the array in any order to form the largest multiple of three. If it's not possible to form such a number, the solution should return an empty string. When forming the number, leading zeros should be discarded unless the number itself is zero. Since the expected output can be very large, the answer is required to be returned as a string.

## Intuition

To solve this problem, we have to understand a few properties of multiples of three:

1. **Divisibility Rule for Three**: For a number to be divisible by three, the sum of its digits must also be divisible by three.
2. **No Leading Zeros**: Without loss of generality, we can ignore zeros at the beginning as they do not contribute to the value of the number. However, we must be careful to leave one zero if all digits are zeros.

Given these two points, we can outline the solution approach:

1. **Sorting**: First, sort the array to make sure we can form the largest possible number at the end by concatenating the digits in descending order.
2. **Dynamic Programming (DP)**: Use DP to find the largest number we can make that is divisible by three. Create an array `f` where `f[i][j]` represents the maximum length of a subsequence formed by the first `i` digits that has a remainder of `j` when divided by 3.
3. **Subsequence Selection**: Once the DP matrix is filled, reconstruct the largest possible number from it. We do this in two steps:
   - Start from `f[n][0]` (since this is where the subsequence with a sum divisible by three and the largest size will be stored) and trace back to find which digits are included in this subsequence.
   - Remove leading zeros, because they don't count unless the number is entirely composed of zeros.

This approach ensures that we'll find the largest subsequence of digits that satisfies the divisibility rule, and by sorting initially, we ensure it's the largest multiple of three possible with the given digits.

## Solution Approach

The solution adopts a dynamic programming approach to construct the largest multiple of three from the array of `digits`. Let's break down the implementation:

1. **Sorting**: `digits.sort()` is used to sort the array in ascending order so that when we construct our number, it is automatically arranged in descending order (as we will construct it backward).

2. **Dynamic Programming Initialization**: Initialize a DP matrix `f` with dimensions `(n + 1) x 3`, where `n` is the number of digits. Each cell is initially filled with `-inf` to signify that there is no valid subsequence yet found for that combination.

### Example Walkthrough

Let's illustrate the solution approach with a small example.

Suppose we are given the array of digits `[8, 1, 9, 3, 4, 5]`. Our goal is to find the largest possible multiple of three.

1. **Sorting**: We sort the array in ascending order, which gives us `[1, 3, 4, 5, 8, 9]`.

2. **Dynamic Programming Initialization**: Let `n` be the number of digits in the sorted array. We initialize a DP matrix `f` with dimensions `(n + 1) x 3`, which in our case is a 7×3 matrix since there are 6 digits. Each cell in `f` is initially filled with `-inf`, which represents that we haven't found any valid subsequences for that remainder. The first row of the DP matrix is initialized to `0`, because a subsequence with `0` digits always has a remainder of `0` and a size of `0`.

Now, we traverse the sorted array and update the DP matrix:

- For the first digit, `1`, we can only form numbers with a remainder of `1` (since 1 is not divisible by 3). We update `f[1][1]` with the length of this subsequence, which is `1` (just the single digit `1`).
- For the second digit, `3`, which is divisible by three, we update `f[2][0]`, `f[2][1]`, and `f[2][2]` based on the previous results in `f[1][0]`, `f[1][1]`, and `f[1][2]` respectively. Since `3` adds a remainder of `0` to whatever we have, all remainders are possible.

This process continues until we've traversed all digits. For ease of illustration, let's assume the final updated DP matrix indicates the largest size subsequences ending in a remainder of 0, 1, or 2 modulo 3 are of sizes 5, `inf`, and `inf`, respectively.

3. **Subsequence Selection**: Since we're interested in a multiple of three, we look at `f[n][0]`. Here, `n` is the total number of digits we started with, so we look at `f[6][0]`. Suppose `f[6][0]` indicates a size of `4`. We then trace back through the DP matrix to determine which digits contribute to this subsequence of size `4` that sums to a multiple of three. Tracing backwards, let's say we identify the subsequence `[9, 3, 5, 1]`.

Finally, since we require the largest possible number and we initially sorted the array in ascending order, we construct our number in reverse order. The largest possible multiple of three that we can obtain from the subsequence `[9, 3, 5, 1]` is `9531`. Since there are no leading zeros to remove, this is our final answer.

The dynamic programming approach ensures that we have considered all possible combinations, and by reconstructing the sequence from the DP matrix, we get the largest value possible. Therefore, the largest multiple of three that we can form from the array `[8, 1, 9, 3, 4, 5]` is `9531`.

## Python Solution

```python
from math import inf
from typing import List

class Solution:
    def largestMultipleOfThree(self, digits: List[int]) -> str:
        # Sort the digits array in ascending order
        digits.sort()
        # Number of digits
        n = len(digits)
        # A 2D array to keep track of maximum length of subsequence that gives remainder j when divided by 3
        dp = [[-inf for _ in range(3)] for _ in range(n + 1)]
        # Initialize the subsequence length to 0 when there are no digits
        dp[0][0] = 0

        # Dynamic programming to fill the 2D array
        for i, digit in enumerate(digits, 1):
            for remainder in range(3):
                # Compare and select the maximum length between not picking and picking the digit
                dp[i][remainder] = max(dp[i - 1][remainder],
                                       dp[i - 1][(remainder - digit % 3 + 3) % 3] + 1)

        # If there is no subsequence which forms a multiple of three, return an empty string
        if dp[n][0] <= 0:
            return ""

        # Build the multiple of three by going backwards in the filled 2D array
        subsequence = []
        remainder = 0
        for i in range(n, 0, -1):
            next_remainder = (remainder - digits[i - 1] % 3 + 3) % 3
            if dp[i - 1][next_remainder] + 1 == dp[i][remainder]:
                # If the digit should be included, add to subsequence
                subsequence.append(digits[i - 1])
                remainder = next_remainder

        # Remove leading zeros
        leading_zeros_removed = 0
        while leading_zeros_removed < len(subsequence) - 1 and subsequence[leading_zeros_removed] == 0:
            leading_zeros_removed += 1
        # Concatenate the digits to form the required number
        return "".join(map(str, subsequence[leading_zeros_removed:]))


# Example usage:
# solution = Solution()
# print(solution.largestMultipleOfThree([8, 1, 9]))  # Output: "981"

```

## Java Solution

```java
import java.util.Arrays;

class Solution {
    public String largestMultipleOfThree(int[] digits) {
        // Sort the array in ascending order
        Arrays.sort(digits);

        // Get the length of the digits array
        int length = digits.length;

        // Initialize a 2D dynamic programming array to store the maximum count of digits that can form a multiple of three
        int[][] dp = new int[length + 1][3];

        // Define an 'inf' (infinity) value as a very large number to be used in our calculations
        final int INF = Integer.MAX_VALUE / 2; // Using half of MAX_VALUE to avoid overflow

        // Fill the dynamic programming array with negative infinity to differentiate between used and unused states
        for (int[] row : dp) {
            Arrays.fill(row, -INF);
        }

        // The count for zero elements to form a sum that is a multiple of three is zero
        dp[0][0] = 0;

        // Build the dynamic programming matrix
        for (int i = 1; i <= length; ++i) {
            for (int j = 0; j < 3; ++j) {
                // Get the mod value of the current digit
                int modValue = (j - digits[i - 1] % 3 + 3) % 3;

                // Maximize the count for dp[i][j] by either taking the previous count or by including the current digit if it can
                dp[i][j] = Math.max(dp[i - 1][j], dp[i - 1][modValue] + 1);
            }
        }

        // If there is no combination that forms a multiple of three, return an empty string
        if (dp[length][0] <= 0) {
            return "";
        }

        // Use StringBuilder to construct the largest number that is a multiple of three
        StringBuilder resultBuilder = new StringBuilder();

        // Trace back the dynamic programming table in reverse to build the number
        for (int i = length, j = 0; i > 0; --i) {
            // Calculate the previous mod value to decide if the current digit should be included
            int previousMod = (j - digits[i - 1] % 3 + 3) % 3;

            // If including the current digit leads to the maximum count, append it to the resultBuilder
            // and update the mod value
            if (dp[i - 1][previousMod] + 1 == dp[i][j]) {
                resultBuilder.append(digits[i - 1]);
                j = previousMod;
            }
        }

        // Remove leading zeros if any
        int startIndex = 0;
        while (startIndex < resultBuilder.length() - 1 && resultBuilder.charAt(startIndex) == '0') {
            ++startIndex;
        }

        // Return the final result string starting from the first non-zero digit
        return resultBuilder.substring(startIndex);
    }
}

```

## C++ Solution

```cpp
#include <vector>
#include <string>
#include <algorithm>
#include <cstring>

class Solution {
public:
    // Function that finds the largest multiple of three using the given digits.
    std::string largestMultipleOfThree(std::vector<int>& digits) {
        // Sort the digits in non-decreasing order
        std::sort(digits.begin(), digits.end());
        int numDigits = digits.size();

        // Initializing the dynamic programming table with a very negative number
        int dp[numDigits + 1][3];
        memset(dp, -0x3f, sizeof(dp));

        // Base case: for 0 digits, the sum = 0 mod 3 is achievable with count 0
        dp[0][0] = 0;

        // Build the table in bottom-up manner
        for (int i = 1; i <= numDigits; ++i) {
            for (int j = 0; j < 3; ++j) {
                int prevMod = (j - digits[i - 1] % 3 + 3) % 3;
                // Pick the larger of not taking the current digit or taking it
                dp[i][j] = std::max(dp[i - 1][j], dp[i - 1][prevMod] + 1);
            }
        }

        // If there's no solution (sum = 0 mod 3 not reachable), return an empty string
        if (dp[numDigits][0] <= 0) {
            return "";
        }

        std::string result;
        int mod = 0; // The current remainder

        // Construct the result string by choosing digits in reverse order
        for (int i = numDigits; i > 0; --i) {
            int modToCheck = (mod - digits[i - 1] % 3 + 3) % 3;
            if (dp[i - 1][modToCheck] + 1 == dp[i][mod]) {
                result += (char)('0' + digits[i - 1]);
                mod = modToCheck;
            }
        }

        // Remove leading zeros except the last zero if the result is zero
        int startNonZero = 0;
        while (startNonZero < result.size() - 1 && result[startNonZero] == '0') {
            ++startNonZero;
        }

        // Get the substring from the first non-zero character
        return result.substr(startNonZero);
    }
};

```

## Typescript Solution

```typescript
function largestMultipleOfThree(digits: number[]): string {
    // Sort the digits array in non-decreasing order.
    digits.sort((a, b) => a - b);

    // Get the number of digits.
    const numDigits = digits.length;

    // f array for dynamic programming, where f[i][j] will store the maximum length
    // of a subsequence of the first i digits with a sum modulo 3 equal to j.
    const dpArray: number[][] = new Array(numDigits + 1).fill(0).map(() => new Array(3).fill(-Infinity));
    dpArray[0][0] = 0;

    // Dynamic programming to fill the dpArray.
    for (let i = 1; i <= numDigits; ++i) {
        for (let j = 0; j < 3; ++j) {
            const mod = (j - digits[i - 1] % 3 + 3) % 3;
            dpArray[i][j] = Math.max(dpArray[i - 1][j], dpArray[i - 1][mod] + 1);
        }
    }

    // If there are no subsequences whose sum is divisible by 3, return an empty string.
    if (dpArray[numDigits][0] <= 0) {
        return '';
    }

    // Iterate backwards to find the digits that make up the largest multiple of three.
    const resultDigits: number[] = [];
    for (let i = numDigits, sumModulo = 0; i > 0; --i) {
        const currentDigitModulo = (sumModulo - digits[i - 1] % 3 + 3) % 3;

        // If including the current digit gives us a longer subsequence.
        if (dpArray[i - 1][currentDigitModulo] + 1 === dpArray[i][sumModulo]) {
            resultDigits.push(digits[i - 1]); // Add digit to the result.
            sumModulo = currentDigitModulo;   // Update the sum modulo for previous digits.
        }
    }

    // Remove any leading zeros except for the last digit if the number is zero.
    let leadingZeroIndex = 0;
    while (leadingZeroIndex < resultDigits.length - 1 && resultDigits[leadingZeroIndex] === 0) {
        ++leadingZeroIndex;
    }

    // Join the result digits to form the largest multiple of three and return it as a string.
    return resultDigits.slice(leadingZeroIndex).reverse().join('');
}

```

## Time and Space Complexity

The given code block is designed to find the largest number that can be formed from a list of digits such that the final number is a multiple of three.

### Time Complexity:

The time complexity of the given solution can be analyzed as follows:

- Initializing the list `f` with size `(n + 1) * 3` takes `O(n)` time.
- The outer loop runs `n` times, where `n` is the number of digits.
- The inner loop runs a constant number of times (exactly `3` times) for each iteration of the outer loop.
- Inside the inner loop, it computes the maximum of two values, which takes constant time `O(1)`.
- After the loops, it iterates from `n` down to `1` to construct the final result, which takes `O(n)` in the worst case.
- Inside this iteration, there's a while loop that potentially iterates over all digits to skip leading zeros. In the worst case, it could iterate over all the elements, but this doesn't affect the overall time complexity.

Given the two main loops nested, with the outer loop running `n` times and the inner loop a constant `3` times, the overall time complexity is:

`O(n) + O(n) * O(3) + O(n) = O(n)`

### Space Complexity:

The space complexity of the code can be analyzed as follows:

- The `f` list is a 2D list with dimensions `(n + 1) * 3`, resulting in a space complexity of `O(n)`.
- The `arr` list can potentially contain every digit from the input in the worst case, thus has a space complexity of `O(n)`.
- Minimal extra space is utilized for variables such as `i`, `j`, `k`, and `x`, but this doesn't affect the overall space complexity.

Therefore, the total space complexity of the algorithm is:

`O(n) + O(n) = O(n)`

So the final space complexity is `O(n)`.