

# 2022. Convert 1D Array Into 2D Array

Easy   Array   Matrix   Simulation

## Problem Description

The given problem presents us with a 1-dimensional array `original` and asks us to create a 2-dimensional array with `m` rows and `n` columns. The new 2D array should be filled with all the elements from `original` in such a way that the first `n` elements of `original` become the first row of the 2D array, the next `n` elements become the second row, and so on, continuing this process until we have `m` rows.

The key condition here is that each row of the newly formed 2D array must be filled with exactly `n` elements. Therefore, a 2D array can only be formed if the total number of elements in `original` is equal to `m * n`. If this condition is not met, it is not possible to form a 2D array that satisfies the criteria, and we should return an empty 2D array.

## Intuition

The solution to this problem hinges on a simple mathematical verification followed by a grouping operation. The verification is to check whether the total number of elements in the `original` array is equal to the total number of elements that would be in the resulting 2D array (`m * n`). If they are not equal, it is impossible to construct the requested 2D array, so we return an empty list.

## Solution Approach

The implementation of the solution uses basic list comprehension and slicing, which are common Python techniques for manipulating lists.

- Verification Step:** The first step in the solution is to verify whether the transformation from a 1D array to a 2D array is possible. This is done by checking if the product of `m` (number of rows) and `n` (number of columns) equals the length of the `original` array. If the condition `m * n != len(original)` is `True`, then it means that the 1D array cannot be perfectly partitioned into a 2D array with the specified dimensions, and the function returns an empty list `[]`.
- Transformation Step:** If the verification step is successful, the code proceeds to transform the `original` array into the desired 2D array. This is where list comprehension and slicing come into play. The list comprehension iterates over the `original` list, starting from index 0, all the way to the last element in steps of `n`.
  - The slice `original[i : i + n]` extracts `n` elements from the `original` array starting at index `i`. The slice's end index `i + n` is non-inclusive, meaning that it will extract elements up to but not including index `i + n`.
  - The `range` function in the list comprehension `range(0, m * n, n)` is used to generate the starting indices for each row. The third argument of `range` is the step, which we set to `n` to ensure we skip ahead by one full row each time.
- Construction Step:** The slices extracted during the transformation step are each a row of the 2D array. The list comprehension collects all these rows and constructs a list of lists, which is the desired 2D array.

The result is a 2D array that uses all elements of the `original` array to form an array with `m` rows and `n` columns as required by the problem. This algorithm is efficient, running in  $O(m*n)$ , which is the size of the `original` array since each element is visited once.

## Example Walkthrough

Let's walk through a small example to illustrate the solution approach described above:

Suppose we have the following 1-dimensional array `original` and values for `m` and `n`:

```
1 original = [1, 2, 3, 4, 5, 6]
2 m = 2
3 n = 3
```

First, let's verify whether the transformation from a 1D array to a 2D array is possible:

- The length of `original` is 6.
- We want to convert it into a 2D array with `m = 2` rows and `n = 3` columns.
- The total number of elements required to fill this 2D array is `m * n = 2 * 3 = 6`.

Since `len(original)` (which is 6) equals `m * n` (also 6), the transformation is possible.

Now let's transform `original` into the desired 2D array:

- We start at index 0 of `original`. The first slice we take is `original[0:0 + 3]`, which gives us the first row `[1, 2, 3]`.
- We then move to the next set of elements by stepping forward `n` places. The next index is 3, so we take the slice `original[3:3 + 3]` which gives us the second row `[4, 5, 6]`.

By putting these rows together, we construct our 2D array as follows:

```
1 [[1, 2, 3],
2  [4, 5, 6]]
```

In this case, the resulting 2D array has exactly `m` rows and `n` columns, using all elements of `original`. The approach works perfectly for this example.

## Python Solution

```
1 from typing import List
2
3 class Solution:
4     def construct2DArray(self, original: List[int], num_rows: int, num_cols: int) -> List[List[int]]:
5         # If the total number of elements in the 2D array doesn't match the length of the original array,
6         # it is not possible to construct the 2D array; return an empty list in such a case.
7         if num_rows * num_cols != len(original):
8             return []
9
10        # Construct the 2D array by slicing the original array.
11        # Walk through the original array in steps of `num_cols` and slice it into rows of length `num_cols`.
12        return [
13            original[i : i + num_cols] # Slice from the current index to the index plus the number of columns.
14            for i in range(0, num_rows * num_cols, num_cols) # Iterate in steps of the number of columns.
15        ]
16
```

## Java Solution

```
1 class Solution {
2     // Method to convert a 1D array into a 2D array with given dimensions m x n
3     public int[][] construct2DArray(int[] original, int m, int n) {
4         // Check if the total elements of the 2D array (m * n) match the length of the original 1D array
5         if (m * n != original.length) {
6             // If they don't match, return an empty 2D array
7             return new int[0][0];
8         }
9
10        // Initialize the 2D array with the given dimensions
11        int[][] twoDArray = new int[m][n];
12
13        // Iterate over each row of the 2D array
14        for (int row = 0; row < m; ++row) {
15            // Iterate over each column of the 2D array
16            for (int column = 0; column < n; ++column) {
17                // Calculate the corresponding index in the original 1D array
18                // and assign the value to the 2D array at [row][column]
19                twoDArray[row][column] = original[row * n + column];
20            }
21        }
22
23        // Return the constructed 2D array
24        return twoDArray;
25    }
26 }
27
```

## C++ Solution

```
1 class Solution {
2 public:
3     // Function to construct a 2D array from a 1D array
4     vector<vector<int>> construct2DArray(vector<int>& original, int rows, int cols) {
5         // Return an empty 2D array if the given dimensions do not match the size of the original 1D array
6         if (rows * cols != original.size()) {
7             return {};
8         }
9
10        // Prepare an output 2D array with the given dimensions
11        vector<vector<int>> result(rows, vector<int>(cols));
12
13        // Loop through each element in the output 2D array
14        for (int i = 0; i < rows; ++i) {
15            for (int j = 0; j < cols; ++j) {
16                // Map 1D array index to corresponding 2D array indices
17                result[i][j] = original[i * cols + j];
18            }
19        }
20
21        // Return the constructed 2D array
22        return result;
23    }
24 };
25
```

## Typescript Solution

```
1 function construct2DArray(original: number[], m: number, n: number): number[][] {
2     // If the length of the original array is not equal to the product of dimensions m and n
3     // Then it's not possible to construct a 2D array of m x n, return empty array
4     if (m * n !== original.length) {
5         return [];
6     }
7
8     // Initialize an empty array for the 2D array
9     const twoDArray: number[][] = [];
10
11    // Loop through the original array with step of size n
12    for (let i = 0; i < original.length; i += n) {
13        // Push a slice of the original array of length n into twoDArray
14        twoDArray.push(original.slice(i, i + n));
15    }
16
17    // Return the constructed 2D array
18    return twoDArray;
19 }
20
```

## Time and Space Complexity

The given code snippet defines a function `construct2DArray` which converts a 1D array to a 2D array with `m` rows and `n` columns.

### Time Complexity

To determine the time complexity, we need to consider the operations performed by the code:

- The function first checks if the total number of elements required for the 2D array (`m * n`) matches the length of the original list. This comparison operation is  $O(1)$ .
- Then, a list comprehension is used to generate the 2D array. This will iterate over the original list in steps of size `n`, creating a total of `m` sublists.

Assuming `k` is the length of the original list, the total number of iterations in the list comprehension is `k / n` which simplifies to `m` (since `m * n = k`).

The slicing operation within each iteration can be considered  $O(n)$  because it involves creating a new sublist of size `n`.

Thus, the time complexity of the list comprehension is  $O(m * n)$ .

Therefore, the total time complexity of the function is  $O(1) + O(m * n)$  which simplifies to  $O(m * n)$ .

### Space Complexity

For space complexity, we consider the additional space required by the program:

- The space needed for the output 2D array which will hold `m * n` elements. Hence, the space complexity for the 2D array is  $O(m * n)$ .
- There is no additional space being used that grows with the input size. Hence, other than the space taken by the output, the space complexity remains constant  $O(1)$  for the code execution.

Therefore, the overall space complexity of the function `construct2DArray` is  $O(m * n)$  because of the storage space for the final 2D array.

In summary:

- Time Complexity:  $O(m * n)$
- Space Complexity:  $O(m * n)$