Solution: Partition Equal Subset Sum

Let's solve the Partition Equal Subset Sum problem using the Dynamic Programming pattern.

We'll cover the following

Statement

Given a non-empty array of positive integers, determine if the array can be divided into two subsets so that the sum of both subsets is equal.

Constraints:

- $1 \leq$ `nums.length` ≤ 200
- $1 \leq$ `nums[i]` ≤ 100

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive approach

We can solve this problem with the following two steps:

1. First, we calculate the sum of the array. If the sum of the array is odd, there can't be two subsets with an equal sum, so we return FALSE.
2. If the sum is even, we calculate $sum/2$ and find a subset of the array with a sum equal to $sum/2$.

The naive approach is to solve the second step using recursion. In this approach, we calculate the result repeatedly every time. For example, consider an array, `[1, 6, 20, 7, 8]`, which can be partitioned into `[1, 20]` and `[6, 7, 8]` with the sum `21`. While computing the sum, we encounter `21` twice: once in the first subset (`6 + 7 = 13, 13 + 8 = 21`) and once in the second subset (`1 + 20 = 21`). However, since we're not storing these sums, it is computed twice.

The time complexity of this approach is $O(2^n)$. In the worst case, for each element in an array, this solution tests two possibilities, i.e., whether to include or exclude it. We can avoid the repeated work done in the naive approach by storing the result calculated at each step. We store all the values in a lookup table.

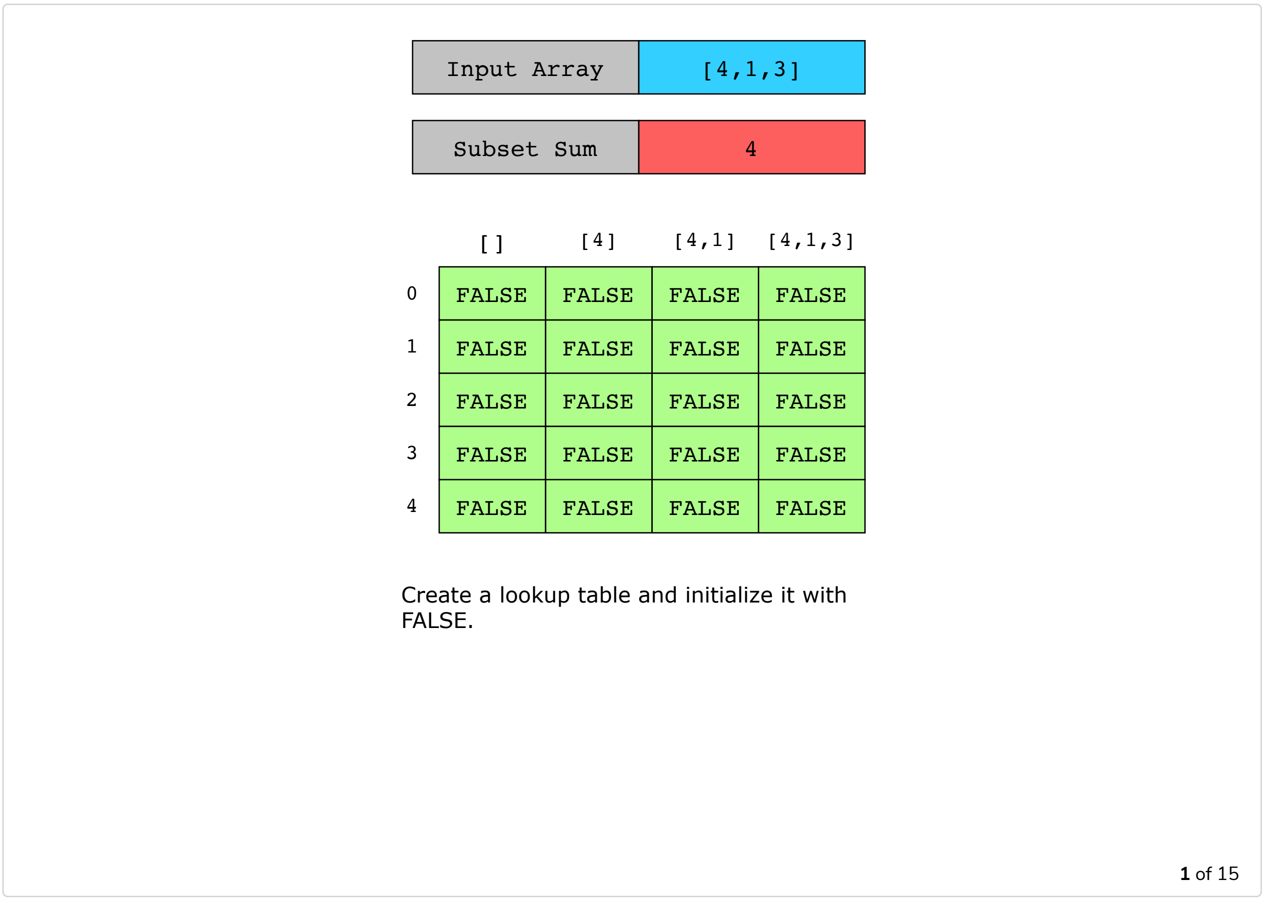
Optimized approach using dynamic programming

We use the bottom-up approach of dynamic programming, also known as the **tabulation technique**. In this approach, the smallest problem is solved, the result is saved, and larger subproblems are computed based on the evaluated results. The problem is divided into subproblems, which are dependent on each other. We start by initializing a lookup table and setting up the values of the base cases. For every subsequent, larger subproblem, we fetch the results of the required preceding smaller subproblems and use them to get the solution to the current subproblem.

Here is how we implement this algorithm:

1. First, we calculate the sum of the array, `nums`. If the sum of the array is odd, there can't be two subsets with an equal sum, so we return FALSE.
2. Create a lookup table, `dp`, of size $(((s/2)+1) \times (n+1))$, where s is the sum, and n is the size of the array. The `dp[0][0]` represents that the sum is 0, and none of the elements is included in the sum. Therefore, $(s/2+1)$ rows and $(n+1)$ columns are needed. Initialize all cells of `dp` with FALSE.
3. Since each element in the array is a positive number, therefore the sum of elements can't be 0. Hence, each element of the first row in `dp` is set to TRUE to represent the solution of the smallest sub-problem.
4. The FALSE in the first column except $[0][0]$ location indicates that an empty array has no subset whose sum is greater than 0.
5. Fill the table in a bottom-up approach where `[i][j]` represents the current row and column entry.
   - If the `j`th element of the array is greater than `i`, it will make the sum greater than `i`, which means we cannot include this element in our subset. Therefore, we copy the previous column's value, which is `dp[i][j-1]`, into `dp[i][j]`.
   - If the `j`th element of the array is less than or equal to `i`, we have two choices: either include it in our subset or exclude it. Here, we want to find out if it is possible to form a subset with a sum of `i` using the first `j` elements of the array.
     - In the first choice, we need to find a subset that adds up to `i - nums[j-1]` using the first `j-1` elements of the `nums` array. That means we are looking at the value of `dp[i - nums[j - 1]][j - 1]`.
     - In the second choice, we exclude the `j`th element from our subset and find a subset that adds up to `i` using the first `j-1` elements of the nums array. This means we are looking at the value of `dp[i][j - 1]`.
     - Finally, we set `dp[i][j]` to the logical OR of these two choices: `dp[i][j] = dp[i - nums[j - 1]][j - 1] OR dp[i][j - 1]`.
6. Return the value present at the last row and last column of the `dp`, which denotes whether the array can be partitioned or not.
   - If we get TRUE, then the array can be partitioned.
   - If we get FALSE, then the array can not be partitioned.

Here's the demonstration of the steps above:



Input Array	[4,1,3]

Subset Sum	4

	[]	[4]	[4,1]	[4,1,3]
0	FALSE	FALSE	FALSE	FALSE
1	FALSE	FALSE	FALSE	FALSE
2	FALSE	FALSE	FALSE	FALSE
3	FALSE	FALSE	FALSE	FALSE
4	FALSE	FALSE	FALSE	FALSE

Create a lookup table and initialize it with FALSE.

1 of 15

Java

```java
import java.util.*;

class PartitionEqualSum {

    public static boolean canPartitionArray(int[] nums) {
        int arraySum = 0;

        // Calculate sum of array.
        for (int num : nums) {
            arraySum += num;
        }

        // If total sum is odd, it cannot be partitioned into equal sum subsets.
        if (arraySum % 2 != 0) {
            return false;
        }

        // Calculate the subset sum.
        int subsetSum = arraySum / 2;

        // Create a lookup table and fill all entries with FALSE.
        boolean[][] dp = new boolean[subsetSum + 1][nums.length + 1];

        // Initialize the first row as TRUE as each array has a subset whose sum is zero
        for (int i = 0; i <= nums.length; i++) {
            dp[0][i] = true;
        }
```

Partition Equal Subset Sum

Solution summary

To recap, the solution to this problem can be divided into the following parts:

1. Create a lookup table and initialize the first row with TRUE.
2. Fill the lookup table in the bottom-up approach by checking if the current number in the input array can be included in the subset sum.
3. In case it can be included in the subset sum, then the table entry is marked TRUE, otherwise, it is marked
4. After filling up, the value present at the last row and column of the lookup table denotes whether the array can be partitioned or not.

Time complexity

The time complexity of the solution above is $O(n \times s)$, where n is the size of the input array and s is the sum of the array. This is the time required to fill the lookup table.

Space complexity

The space complexity of the above solution is $O(n \times s)$. This is space taken by the lookup table.

Can we do better?

A space-optimized solution can be devised by using an array instead of a table, which reduces the space complexity from $O(n \times s)$ to $O(s)$. We initialize an array `dp` of size $((s/2) + 1)$ with FALSE in each slot in the array except the first slot, which is initialized with TRUE. This means that a subset with a sum of 0 can always be formed by selecting no elements from the input array.

We can then iterate through the elements in the input array `nums`. For each element `val` in `nums`, we can update `dp` as follows:

- Iterate through `dp` in reverse order, starting from $s/2$ down to `val`.
- For each index `j` in the `dp`, the algorithm sets `dp[j]` to TRUE if `dp[j]` is already TRUE or if `dp[j - val]` is TRUE. The reason for this is that if `dp[j - val]` is TRUE, it means that a subset with the sum `j - val` can be constructed using the previous elements of the input array. Therefore, by including the current element `val`, a subset with sum `j` can be constructed.

In the end, the last element of `dp` represents the output.

← Back
Partition Equal Subset

Next →