10 MAY 2019 / #PROGRAMMING

A variation on the Knapsack Problem: how to solve the Partition Equal Subset Sum problem in Java

by Fabian Terh

dynamic programming. You can read about it here.

Today I want to discuss a variation of KP: the partition equal subset sum problem. I first saw this problem on Leetcode — this was what prompted me to learn about, and write about, KP.

This is the problem statement (reproduced partially without examples):

> Given a non-empty array containing only positive integers, find if the array can be partitioned into two subsets such that the sum of elements in both subsets is equal.

For the full problem statement, with constraints and examples, check out the Leetcode problem.

Dynamic programming

Like with KP, we'll be solving this using dynamic programming. Since this is a variation of KP, the logic and methodology is largely similar.

Solution

We will house our solution in a method that returns a boolean — true if the array can be partitioned into equal subsets, and false otherwise.

Step 1: Guarding against odd array sum

Trivially, if all the numbers in the array add up to an odd sum, we can return false. We only proceed if the array adds up to an even sum.

Step 2: Creating the table

Table rows represent the set of array elements to be considered, while table columns indicate the sum we want to arrive at. Table values are simply boolean values, indicating whether a sum (column) can be arrived at with a set of array elements (row).

Concretely, row *i* represents a set of array elements from indices 0 to (*i*-1). The reason for this offset of 1 is because row 0 represents an empty set of elements. Therefore, row 1 represents the first array element (index 0), row 2 represents the first two array elements (indices 0–1), and so on. In total, we create `n + 1` rows, inclusive of 0.

We only want to know if we can sum up exactly to half the total sum of the array. So we only need to create `totalSum / 2 + 1` columns, inclusive of 0.

Step 3: Pre-filling the table

We can immediately begin filling the entries for the base cases in our table — row 0 and column 0.

In the first row, every entry — except the first — must be `false`. Recall that the first row represents an empty set . Naturally, we are unable to arrive at any target sum — column number — except 0.

In the first column, every entry must be `true`. We can always,trivially, arrive at a target sum of 0, regardless of the set of elements we have to work with.

Step 4: Building the table (the crux of the problem)

one of the following three conditions are satisfied:

1. the entry at row *i*-1 and column *j* is `true`. Recall what the row number represents. Therefore, if we are able to attain a particular sum with a subset of the elements that we have presently, we can also attain that sum with our current set of elements — by simply not using the extra elements. This is trivial. Let's call this `prevRowIsTrue`.
2. The current element is exactly the sum we want to attain. This is also trivially true. Let's call this `isExactMatch`.
3. If the above two conditions are not satisfied, we have one remaining way of attaining our target sum. Here, we use the current element — the additional element in the set of elements in our current row compared to the set of elements in the previous row — and check that we are able to attain the remainder of the target sum. Let's call this `canArriveAtSum`.

Let's unpack condition 3. We can only use the current element **if** it is less than our target sum. If they're equal, condition 2 would be satisfied. If it's larger, we can't use it. Therefore, the first step is to ensure that current element < target sum.

After using our current element, we are left with the remainder of our target sum. We then check if that is attainable by checking the corresponding entry in the row above.

As with regular KP, we want to progressively build our table from the bottom-up. We start with the base cases, until we arrive at our final solution.

We simply return `return mat[nums.length][totalSum / 2];`.

Working code

Thanks for reading!
