Daily Coding Problem #60

Problem

This problem was asked by Facebook.

Given a multiset of integers, return whether it can be partitioned into two subsets whose sums are the same.

For example, given the multiset `{15, 5, 20, 10, 35, 15, 10}`, it would return true, since we can split it up into `{15, 5, 10, 15, 10}` and `{20, 35}`, which both add up to 55.

Given the multiset `{15, 5, 20, 10, 35}`, it would return false, since we can't split it up into two subsets that add up to the same sum.

Solution

The naive, brute force solution would be to try every combination of two subsets and check their sums. We could do this by trying to generate each subset of our input set, and then checking the sum of that subset with the sum of everything not in the subset.

To speed this up, notice that we really only need to find a subset that adds up to half of the total sum of all the integers. This is because of the pigeonhole principle: if one subset adds up to half of the sum, then the rest of the sum must be made up of the rest of the set.

So, we can generate the powerset of our set and check if any of them sum to `k / 2`, where `k` is the sum of the set. We know immediately that if `k` is odd, then we can't partition the sets, so we can immediately return False.

We did powerset in Daily Coding Problem #37, so let's reuse that:

```
def power_set(s):
    if not s:
        return [[]]
    result = power_set(s[1:])
    return result + [subset + [s[0]] for subset in result]
```

Then partition will just be:

```
def partition(s):
    k = sum(s)
    if k % 2 != 0:
        return False
    powerset = power_set(s)
    for subset in powerset:
        if sum(subset) == k / 2:
            return True
    return False
```

This will run in O(N * 2^N) time though, since we must generate every subset and sum them up. Can we make this any faster?

Notice that we've reduced the problem into finding a subset of integers that add up to `k / 2`, which is exactly the same Daily Coding Problem #42: finding a subset of integers that sum up to `k` (a different `k`).

Recall that we solved that problem by created a matrix of size `len(nums) + 1` by `k + 1`, and then using dynamic programming to fill up the matrix. We can something similar here, except we'll use our `k / 2` as our target.

Each entry `A[i][j]` in our matrix will represent whether or not we can make the integer `i` with the elements of our set from 0 to j. So we'll do the following:

- Create a matrix of size `k + 1` by `len(s) + 1` of booleans (all initialized to False).
- Initialize the top row to True, since we can make 0 with anything (by not picking anything)

- Initialize the left column to False (except for the one in the first row), since we

can't make anything other than 0 with nothing

- Iterate over the matrix from top-to-bottom, then left-to-right:
    - At each index `A[i][j]`, look at `A[i][j - 1]` or `A[i - last][j - 1]` and set to True if any are true.
- Return the value at the bottom-right of the matrix.

```
def partition(s):
    k = sum(s)
    if k % 2 != 0:
        return False
    k_over_two = k // 2

    A = [[False for _ in range(len(s) + 1)] for _ in range(k_over_two + 1)]

    for j in range(len(s) + 1):
        A[0][j] = True

    for i in range(1, k_over_two + 1):
        A[i][0] = False

    for i in range(1, k_over_two + 1):
        for j in range(1, len(s) + 1):
            using_last = i - s[j - 1]
            if using_last >= 0:
                A[i][j] = A[i][j - 1] or A[using_last][j - 1]
            else:
                A[i][j] = A[i][j - 1]
    return A[-1][-1]
```

This will take O(K * N) time and space, just like in the knapsack problem.