Daily Coding Problem #42

Problem

This problem was asked by Google.

Given a list of integers S and a target number k, write a function that returns a subset of S that adds up to k. If such a subset cannot be made, then return null.

Integers can appear more than once in the list. You may assume all numbers in the list are positive.

For example, given S = [12, 1, 61, 5, 9, 2] and k = 24, return [12, 9, 2, 1] since it sums up to 24.

Solution

Let's consider the brute force method: selecting all subsets, summing them, and checking if they equal k. That would take O(2^N * N) time, since generating all subsets takes O(2^N) and we need to sum everything in the subset.

We can do a little better by implicitly computing the sum. That is, for each call, we can basically choose whether to pick some element (let's say the last) in our set and recursively looking for `k - last` in the remaining part of the list, or exclude the last element and keep on looking for k in the remaining part of the list recursively.

```
def subset_sum(nums, k):
    if k == 0:
        return []
    if not nums and k != 0:
        return None

    nums_copy = nums[:]
    last = nums_copy.pop()

    with_last = subset_sum(nums_copy, k - last)
    without_last = subset_sum(nums_copy, k)
    if with_last is not None:
        return with_last + [last]
    if without_last is not None:
        return without_last
```

This will run in O(2^N) theoretically, but practically, since we copy the whole array on each call, it's still O(2^N * N), which is worse than exponential.

Let's try to improve the running time by using dynamic programming. We have the recursive formula nailed down. How can we use bottom-up dynamic programming to improve the runtime?

We can construct a table `A` that's size `len(nums) + 1` by `k + 1`. At each index `A[i][j]`, we'll keep a subset of the list from `0..i` (including lower, excluding upper bound) that can add up to `j`, or null if no list can be made. Then we will fill up the table using pre-computed values and once we're done, we should be able to just return the value at `A[-1][-1]`. Let's first initialize the list:

```
A = [[None for _ in range(k + 1)] for _ in range(len(nums) + 1)]
```

To begin, we can initialize each element of the first row (`A[i][0] for i in range(len(nums) + 1)`) with the empty list, since any subset of the list can make 0: just don't pick anything!

```
for i in range(len(nums) + 1):
    A[i][0] = []
```

Each element of the first column (`A[0][j]` for j in range(1, len(nums))`) starting from the first row should be null, since we can't make anything other than 0 with the empty set. Since we've initialized our whole table to be null, then we don't need to do anything here.

```
[], [], [], [], [], ...
None, None, None, ...
None, None, None, ...
None, None, None, ...
...
```

Now we can start populating the table. Iterating over each row starting at 1, and then each column starting at 1, we can use the following formula to compute `A[i][j]`:

- First, let's consider the last element of the list we're looking at: `nums[i - 1]`. Let's call this `last`.
- If `last` is greater than `j`, then we definitely can't make `j` with `nums[:i]` including `last` (since it would obviously go over). So let's just copy over whatever we had from `A[i - 1][j]`. If we can make `j` without `last`, then we can still make `j`. If we can't, then we still can't.
- If `last` smaller than or equal to `j`, then we still might be able to make `j` using `last`

- If we can make `j` without `last` by looking up the value at `A[i - 1][j]` and if it's not null, then use that.
- Else, if we can't make `j` without `last`, check if we can make it *with* `last` by looking up the value at `A[i - 1][j - last]`. If we can, then copy over the list from there and append the last element to it.
- Else, we can't make it with or without `j`, so set `A[i][j]` to null.

```
for i in range(1, len(nums) + 1):
    for j in range(1, k + 1):
        last = nums[i - 1]
        if last > j:
            A[i][j] = A[i - 1][j]
        else:
            if A[i - 1][j] is not None:
                A[i][j] = A[i - 1][j]
            elif A[i - 1][j - last] is not None:
                A[i][j] = A[i - 1][j - last] + [last]
            else:
                A[i][j] = None
```

Putting it all together:

```
def subset_sum(nums, k):
    A = [[None for _ in range(k + 1)] for _ in range(len(nums) + 1)]

    for i in range(len(nums) + 1):
        A[i][0] = []
```

```
    for i in range(1, len(nums) + 1):
        for j in range(1, k + 1):
            last = nums[i - 1]
            if last > j:
                A[i][j] = A[i - 1][j]
            else:
                if A[i - 1][j] is not None:
                    A[i][j] = A[i - 1][j]
                elif A[i - 1][j - last] is not None:
                    A[i][j] = A[i - 1][j - last] + [last]
                else:
                    A[i][j] = None

    return A[-1][-1]
```

This runs in O(k * N) time and space.
