5 Partitioning a set into equal-sum parts

Given a set of positive integers, determine if it can be partitioned into two parts having equal sum.

Example 1: `numbers = [2, 3, 5, 6]` can be partitioned into `[2, 6]` and `[3, 5]`, which both have sum 8.

Clarification questions

Q: What result should be returned for the empty set?
A: The set is guaranteed not to be empty.

Q: How large may the set be?
A: Up to 50 elements.

Solution 1: dynamic programming, top-down, $O(nS)$ time

A straightforward brute-force solution is to enumerate all possible subsets `s1`; for each one, compute the complement `s2 = numbers - s1`; then check if `sum(s1) == sum(s2)`.

The problem with this approach is its performance: if there are `n` numbers, this solution has time complexity $O(n2^n)$, since there are 2^n possible subsets, and computing the sum of each set requires $O(n)$ operations. This is too slow: the algorithm does not scale to the given upper limit of 50 numbers.

Nevertheless, the brute-force idea is useful, since we notice an interesting property: if two subsets `s1` and `s2` form an equal-sum partition, then the following are true:

- `sum(s1) == sum(s2)`;
- `sum(s1) + sum(s2) == sum(numbers)`.

This means that `sum(s1) == sum(s2) == sum(numbers) / 2`. Therefore we can reduce the problem to finding subset `s1` having sum `S = sum(numbers) / 2`.

The search can be formulated using recursion. For the first element of numbers `numbers[0]`, we have two choices:

- Do not include `numbers[0]` into `s1`. Try to form `s1` from `numbers[1:]`, with the target sum `S`.
- Include `numbers[0]` into `s1`. Form the rest of `s1` from `numbers[1:]`, with the target sum `S - numbers[0]`.

Here is an example of the steps we make to solve `numbers = [2, 3, 5, 6]`, with recursion depth shown up to 2:

- Form `s1` from `numbers[0:]` with `target_sum` 8:
    - Try without including `numbers[0] == 2` into `s1`. Form `s1` from `numbers[1:]` with `target_sum` 8:
        - Try without including `numbers[1] == 3` into `s1`. Form `s1` from `numbers[2:]` with `target_sum` 8: …

    - Try including `numbers[1] == 3` into `s1`. Form `s1` from `numbers[2:]` with `target_sum` 8 - 3 = 5: …
- Try including `numbers[0] == 2` into `s1`. Form `s1` from `numbers[1:]` with `target_sum` 8 - 2 = 6:
    - Try without including `numbers[1] == 3` into `s1`. Form `s1` from `numbers[2:]` with `target_sum` 6: …
    - Try including `numbers[1] == 3` into `s1`. Form `s1` from `numbers[2:]` with `target_sum` 6 - 3 = 3: …

Let's analyze the time complexity of this solution. Suppose we define a function `find_subset(index, target_sum)` that implements the recursion.

Naively, it would seem that we have to make 2^n calls of `find_subset`, since the recursion depth is `n` and each call makes 2 deeper calls.

However, the number of possible parameter combinations that can be passed to `find_subset` is only `n * S` (`n` values for `index` and up to `S` values for `target_sum`). This means that out of the 2^n calls, many are redundant, so we can cache the results to achieve time complexity of only $O(nS)$.

> **Computing an upper bound for the cache size**
>
> Notice how easy it was to reason about the time complexity once we found an upper bound for the cache size. This is often the most straightforward way to compute not only the time complexity of dynamic programming solutions, but also their memory requirements.

We can implement the solution as:

```
from functools import lru_cache

def can_partition(numbers):
    @lru_cache(maxsize=None)
    def find_subset(index, target_sum):
        """
        Searches for a subset of numbers[index:]
        having sum target_sum.
        """
        if target_sum == 0:
            # If we hit the target_sum, we found a valid subset.
            return True
        if target_sum < 0 or index >= len(numbers):
            # We either overshot the target sum,
            # or ran out of numbers.
            return False
        number = numbers[index]
        # Search numbers[index+1:], either using the current number
```

```
        # or without using it.
        return (find_subset(index + 1, target_sum - number) or
                find_subset(index + 1, target_sum))

    total = sum(numbers)
    if total % 2:
        # Impossible to split it into subsets of equal sum.
        return False
    return find_subset(0, total // 2)
```

Notice the similarity to the solution of the change-making problem. The main difference is that for the change-making problem, we were allowing the use of each number (a coin) multiple times; for this problem, it can only be used once. To enforce this constraint, the recursive function requires the argument `index`, in addition to the remaining sum.

> **Identify similarities and differences between problems**
>
> A good way to improve your understanding of dynamic programming problems and their solutions is to think about how they relate to each other, as there are often many similarities between them.