4 Number of expressions with a target result

Given a list of integers and a target result, count the number of ways in which we can add `+` or `-` operators between the integers such that the expression that is formed is equal to the target result.

Example: Given numbers `[1, 2, 2, 3, 1]` and target result 3, we can form three expressions:

- `1 + 2 + 2 - 3 + 1 = 3`
- `1 + 2 - 2 + 3 - 1 = 3`
- `1 - 2 + 2 + 3 - 1 = 3`

Therefore the answer is 3.

Clarification questions

Q: What result should be returned if the list contains a single number?
A: No operators can be added in this case, so the only expression that can be formed is the given number itself. Return 1 if the number is equal to the result, 0 otherwise.

Q: What result should be returned if the list contains no numbers?
A: The result should be zero, since no expressions can be formed.

Q: How long can the list be?
A: It may contain up to 30 numbers.

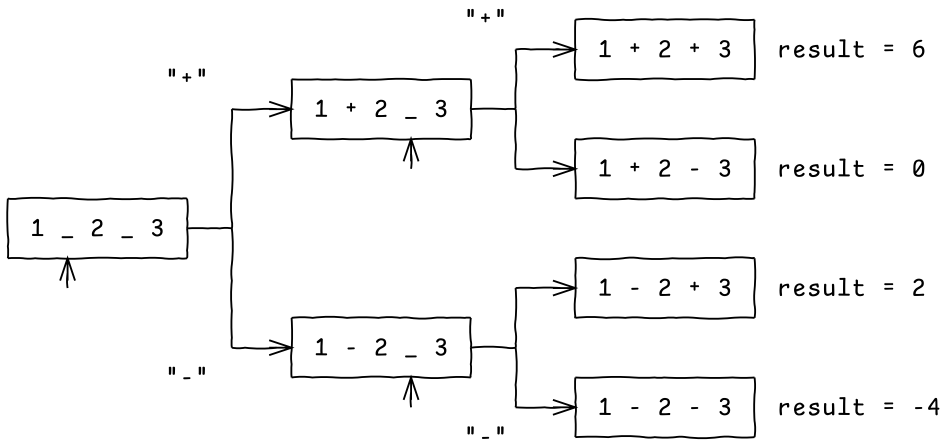
Solution 1: brute-force, $O(2^n)$ time

A straightforward solution is to generate all possible expressions, keep only the ones that are equal to the target result, and count them.

We can generate expressions incrementally, assigning operators one by one. For each operator, we have 2 choices: either use `+` or `-`. To generate all expressions, we first use `+` for the first operator, then continue assigning the other operators recursively; then we revisit the first operator and use `-` this time, then assign the other operators recursively once more. We apply the same rule to the other operators.

Whenever we form a full expression, we check if its result is equal to the target, and if it is, we count it.

Here is an example of using this method on numbers `[1, 2, 3]`:



The solution can be implemented as:

```python
def count_expressions(numbers, target_result):
    def count(index, partial_result):
        """
        Counts the number of expressions equal to `target_result`,
        given that the first `index` operators have been assigned,
        thus the left part of the expression is equal to `partial_result`.
        """
        if index == len(numbers):
            # We formed a full expression. Count it if we hit the target.
            if partial_result == target_result:
                return 1
            return 0
        # For the operator before `numbers[index]`, we have two options:
        # Add the `+` sign:
        count_add = count(index + 1, partial_result + numbers[index])
        # Add the `-` sign:
        count_sub = count(index + 1, partial_result - numbers[index])
        # Each option may yield some valid expressions. Sum up the counters.
        return count_add + count_sub
    first_index = 1
    partial_result = numbers[0]
    return count(first_index, partial_result)
```

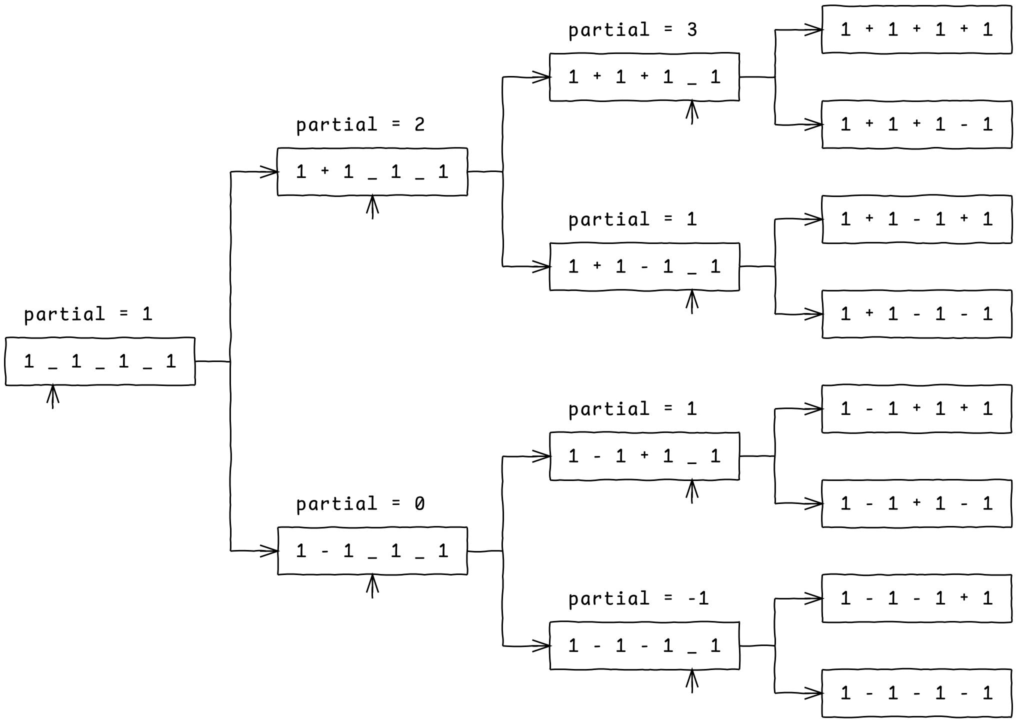
The complexity is $O(2^n)$, where `n = len(numbers)`. This can be proved by induction: if we increase the input number list length by 1 (by appending a number), the number of steps we make doubles.

The complexity can be also be determined graphically from the tree we have drawn for the example. It is a complete binary tree, where each node represents a step in the execution of the algorithm. The height of the tree is `n`, thus the number of nodes is $2^n - 1$, which gives $O(2^n)$ steps.

Since we have to handle up to 30 numbers, the brute-force algorithm has to make 1 billion steps, which is too slow.

Solution 2: dynamic programming, top-down, $O(nS)$ time

We can speed up our computation if we notice that we are doing redundant computations. The previous example is too small to see any redundancies, since there are no two expressions with the same sum. Let's consider another example where the list of numbers is `[1, 1, 1, 1]`:



We find two subtrees that are redundant: the prefixes `1 + 1 - 1` and `1 - 1 + 1` have the same partial result 1. Therefore it does not make sense to compute the expression count for subtrees twice, since it is the same.

To avoid reduntant computations, we can add a cache for the helper function:

```python
from functools import lru_cache

def count_expressions(numbers, target_result):
    @lru_cache(maxsize=None)
    def count(index, partial_result):
        """
        Counts the number of expressions equal to `target_result`,
        given that the first `index` operators have been assigned,
        thus the left part of the expression is equal to `partial_result`.
        """
        if index == len(numbers):
            # We formed a full expression. Count it if we hit the target.
            if partial_result == target_result:
                return 1
            return 0
        # For the operator before `numbers[index]`, we have two options:
        # Add the `+` sign:
        count_add = count(index + 1, partial_result + numbers[index])
        # Add the `-` sign:
        count_sub = count(index + 1, partial_result - numbers[index])
        # Each option may yield some valid expressions. Sum up the counters.
        return count_add + count_sub
    first_index = 1
    partial_result = numbers[0]
    return count(first_index, partial_result)
```

To analyze how caching affects the time complexity, let's think about the worst-case scenario: how many times the `count` function may execute without having the result cached. The answer is $n \times s$, where s is the number of possible distinct results of the partial expressions (we don't know its value yet, but we will look at that in a moment). Note that we multiply s by n since we may reach the same partial result for different operators (partial expressions of different lengths).

Looking at this from another perspective, $n \times s$ is the upper bound for the number of distinct input argument values passed to the function `count(index, partial_result)`.

Computing s is tricky, but we can compute its bounds instead. s is a number in the interval `[-S, S]` with `S = sum([abs(x) for x in numbers])`. Therefore the time complexity of the solution using caching is $O(nS)$.

This may be an overestimate of the actual number of steps when the input numbers are all large (for example: `[1000, 2000, 3000]`). For small numbers, it is a tight bound.

Thinking about the worst-case scenario and upper bounds

When it is hard to reason about the complexity of a solution, it is often easier to focus on the worst-case scenario. But sometimes, even analyzing that is difficult. In these cases it may help to think about an upper bound. Upper bounds are useful in practice, especially for capacity planning, since they provide guarantees about the maximum resource usage of a system.

Solution 3: dynamic programming + BFS, bottom-up, $O(nS)$ time

We can rewrite the solution in non-recursive form, using similar expression generation rules.

We start with a simple expression containing only the first number. We then extend this expression with an operator and the next number, in all possible ways, obtaining expressions having two numbers. We store these new expressions in a list. At this point, we can discard the expression containing just one number, and start further expansions from the expressions containing two numbers. We repeat the procedure until we obtain a list of complete expressions.

The problem with this approach is that the memory requirements are very large, since we have to store up to 2^n expressions.

To improve memory efficiency, we do not store the partial expressions, only their results. We can make the storage even more efficient by eliminating duplicate values: we maintain a counter for each partial result; by default it is 1, and we increment it each time we find a new expression having the same result. These counters can be stored in a dictionary, or the Python `collections.Counter` type.

To understand this approach better, consider an example where `numbers = [1, 1, 1, 1]`. At each step, we compute the dictionary `partial_result_count`, where `partial_result_count[value]` stores the number of expressions formed from `numbers[:index]` having the result equal to `value`.

Here are all the steps performed by the algorithm:

- Initially the only expression is `"1"`. Then `partial_result_count = {1: 1}` (i.e. value 1 can be formed in one way).
- We add the second number to the known expressions, once as `"value + 1"` and once as `"value - 1"`. We can obtain two possible partial results:
  - 2 from the previous value 1 by adding 1;
  - 0 from the previous value 1 by subtracting 1.

  We update `partial_result_count = {2: 1, 0: 1}`. Note that we keep counters only for values obtained for expressions with 2 numbers.
- We add the third number to the known expressions, once as `"value + 1"` and once as `"value - 1"`. We can obtain the following partial results:
  - 3 from the previous value 2 by adding 1;
  - 1 from the previous value 0 by adding 1;
  - 1 from the previous value 2 by subtracting 1;
  - -1 from the previous value 0 by subtracting 1.

  We update `partial_result_count = {3: 1, 1: 2, -1: 1}`. We keep counters only for values obtained for expressions with 3 numbers. Note that the counter for value 1 is 2, since there are two possible expressions that generate it. This is the first redundancy encountered so far.
- We add the fourth number to the known expressions, once as `"value + 1"` and once as `"value - 1"`. We can obtain the following partial results:
  - 4 from the previous value 3 by adding 1;
  - 2 from the previous value 1 by adding 1;
  - 0 from the previous value -1 by adding 1;
  - 2 from the previous value 3 by subtracting 1;
  - 0 from the previous value 1 by subtracting 1;

    - -2 from the previous value -1 by subtracting 1.

  We update `partial_result_count = {4: 1, 2: 3, 0: 3, -2: 1}`.
- Since we exhausted all numbers, `partial_result_count` stores the final result count. We simply return `partial_result_count[target_result]`.

This solution can be implemented as:

```python
import collections

def count_expressions(numbers, target_result):
    # The offset reached in the list of numbers.
    index = 1
    # partial_result_count[value] = count of expressions formed
    #     from numbers[:index] having the result equal to `value`.
    partial_result_count = collections.Counter({ numbers[0]: 1 })
    while index < len(numbers):
        next_result_count = collections.Counter()
        for prefix_result, count in partial_result_count.items():
            # For each prefix, we extend the expression with
            # + numbers[index] and - numbers[index].
            new_result_add = prefix_result + numbers[index]
            new_result_sub = prefix_result - numbers[index]
            # Propagate the counters to the new expressions.
            next_result_count[new_result_add] += count
            next_result_count[new_result_sub] += count
        # Advance to the next number.
        partial_result_count = next_result_count
        index += 1
    return partial_result_count[target_result]
```

The time complexity is $O(nS)$, since we iterate over n numbers, and at each iteration we process s partial results. As previously discussed, s a number in the interval `[-S, S]` where `S = sum([abs(x) for x in numbers])`.

The space complexity is $O(S)$. This is better than the recursive solution, which had space complexity $O(nS)$, due to the cache. The solution is also likely faster due to the lack of function call overheads.

Unit tests

We propose three unit tests:

The first test handles a few short lists of numbers, for which the expressions can be computed by hand, thus it is easy to verify that the result is correct.

The second test handles the edge case where there is a single number in the list. We check two subcases: one where the number matches the target result, and another one where it does not.

The third test handles large inputs, thus checking if the solution has good performance. In this case, it is too difficult to check the result by hand.

We use instead a very simple input: all numbers equal to one, and target result zero. For this input, the number of expressions having a target result can be computed using combinatorics. To generate expressions with n ones, with n even, and target sum zero, half of the operators must be "-" and the other half must be "+", such that they cancel each other out. The number of such expressions is $\binom{n-1}{n/2}$, since there are $n - 1$ operators in an expression, out of which $n/2$ have to be chosen as "-".

The tests can be written as:

```
class TestCountExpressions(unittest.TestCase):
    def test_1_simple(self):
        self.assertEqual(count_expressions([2, 1, 1], 2), 2)
        self.assertEqual(count_expressions([1, 2, 2, 3, 3], 7), 2)
        self.assertEqual(count_expressions([1, 2, 2, 3, 1], 3), 3)

    def test_2_single_number(self):
        self.assertEqual(count_expressions([3], 2), 0)
        self.assertEqual(count_expressions([3], 3), 1)

    def test_3_perf(self):
        self.assertEqual(count_expressions([1] * 20, 0), math.comb(19, 10))
        self.assertEqual(count_expressions([1] * 30, 0), math.comb(29, 15))
```