3 Change-making

Given a money amount and a list of coin denominations, provide the combination of coins adding up to that amount, that uses the fewest coins.

Example 1: Pay amount 9 using coin denominations `[1, 2, 5]`. The combination having the fewest coins is `[5, 2, 2]`. A suboptimal combination is `[5, 1, 1, 1, 1]`: it adds up to 9, but is using 5 coins instead of 3, thus it cannot be the solution.

Example 2: Pay amount 12 using coin denominations `[1, 6, 10]`. The combination having the fewest coins is `[6, 6]`. A suboptimal combination is `[10, 1, 1]`.

Clarification questions

Q: What result should be returned for total amount 0?
A: The empty list `[]`.

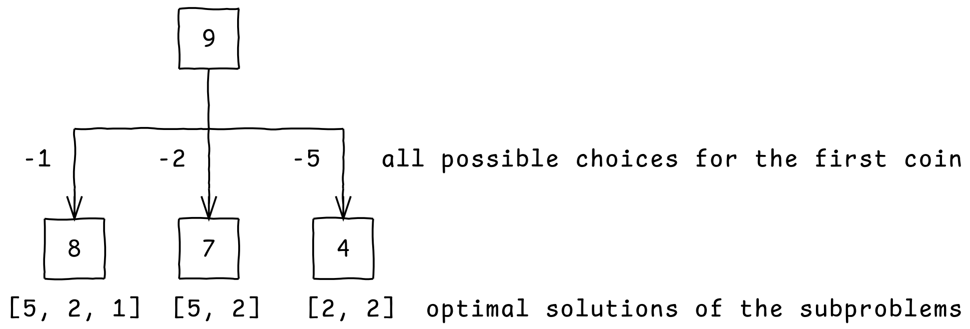
Q: Is it possible that the amount cannot be paid with the given coins?
A: Yes. For example, 5 cannot be paid with coins `[2, 4]`. In such a situation, return `None`.

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

We can formulate a top-down dynamic programming solution if we model the problem as a recurrence. For any non-zero amount that has to be paid optimally using the given coins, we know that at least one of the coins has to be used. The problem is that we do not know which one. If we knew, we could use that as a starting point to reach a subproblem: we could subtract its value from the amount, then solve an instance of the problem for the remaining, smaller amount. We would continue in this way until the remaining amount becomes 0.

However, we do not know which coin to choose first optimally. In this situation, we have no other choice but try all possible options in brute-force style. For each coin, we subtract its value from the amount, then solve by recurrence the subproblem—this leads to a candidate solution for each choice of the first coin. Once we are done, we compare the candidate solutions and choose the one using the fewest coins.

Here is an example of how we would form the optimal change for amount 9, using coins `[1, 2, 5]`. We represent each amount as a node in a tree. Whenever we subtract a coin value from that amount, we add an edge to a new node with a smaller value. Please mind that the actual solution does not use trees, at least not explicitly: they are shown here only for clarity.

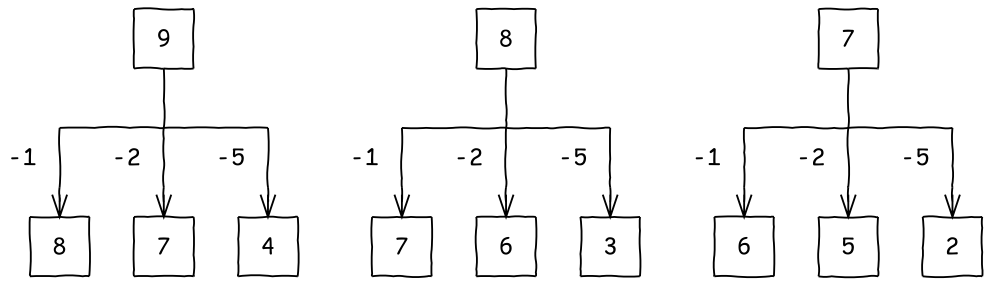

9
-1
-2
-5
all possible choices for the first coin
8
7
4
[5, 2, 1]
[5, 2]
[2, 2]
optimal solutions of the subproblems

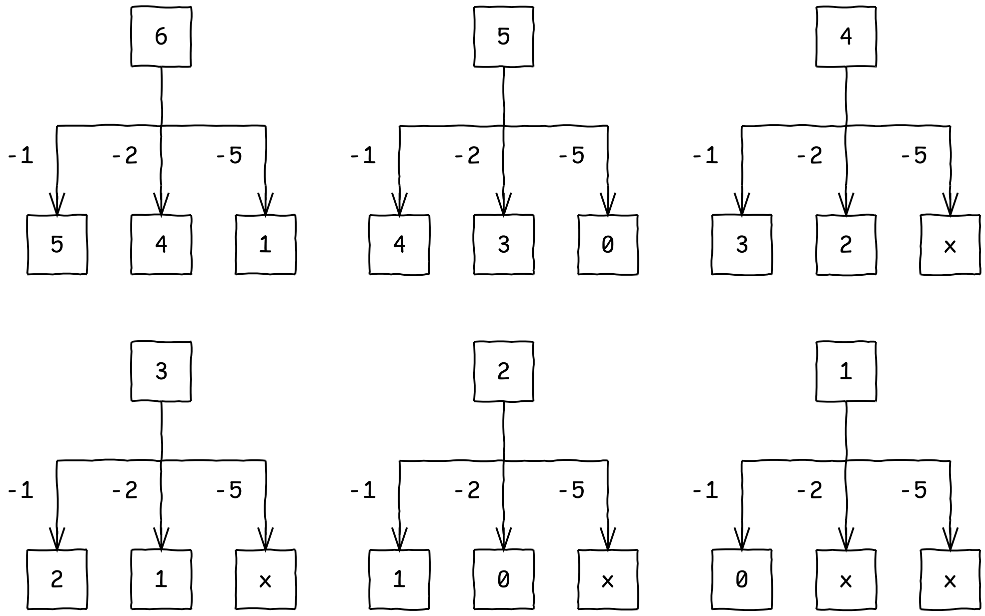
This diagram shows that for value 9, we have three options for choosing the first coin:

- We choose coin 1. We now have to solve the subproblem for value 9 − 1 = 8. Suppose its optimal result is `[5, 2, 1]`. Then we add coin 1 to create the candidate solution `[5, 2, 1, 1]` for 9.
- We choose coin 2. We now have to solve the subproblem for value 9 − 2 = 7. Suppose the optimal result is `[5, 2]`. We add to it coin 2 to create the candidate solution `[5, 2, 2]` for 9.
- We choose coin 5. We now have to solve the subproblem for value 9 − 5 = 4. The optimal result is `[2, 2]`. We add to it coin 5 to create the candidate solution `[5, 2, 2]` for 9.

Now that we are done solving the subproblems, we compare the candidate solutions and choose the one using the fewest coins: `[5, 2, 2]`.

For solving the subproblems, we use the same procedure. The only difference is that we need to pay attention to two edge cases: the terminating condition (reaching amount 0), and avoiding choices where we are paying too much (reaching negative amounts). To understand these better, let's take a look at the subproblems:





Drawing the subproblems helps us see clearly the edge cases:

- When the amount becomes 0, we have to stop the iteration, since the subproblem is solved and there is nothing left to be paid. We can see the 0 node in several of the subproblem trees.
- When the amount goes below zero, the given amount cannot be formed with that coin. For example, trying to pay amount 3 using coin 5 would require solving the subproblem for amount -2, which does not make sense.

In addition to that, we can also notice that there are many redundancies among the subproblems. This is very important, as it affects the performance of the algorithm. We originally thought about solving the problem using brute force, which results in a slow algorithm: $O(n^v)$, where n is the number of coins and v is the value of the amount we have to pay. In other words, the brute-force solution has exponential complexity, with very poor performance.

However, now that we know that there are redundancies among the subproblems, we can cache their results to reduce the complexity to only $O(nv)$: in each step we must try n coins, and we have up to v steps (in the worst case, we pay v with v coins of value 1). This result is great: we reduced the time complexity from exponential to polynomial!

We are now ready to write a first implementation of the recurrence, without caching (the brute-force, exponential complexity solution). This is because we want to focus on the logic of the recurrence, without distractions; we will add caching afterwards.

```
def make_change(coins, amount):
    """
    Given a list of coin values, and an amount to be paid,
    returns the shortest list of coins that add up to that amount.
    If the amount to be paid is zero, the empty list is returned.
    If the amount cannot be paid using the coins, None is returned.
    """
    # Handle payment of amount zero.
    if not amount:
        return []
    # Negative amounts cannot be paid.
    if amount < 0:
        return None
    optimal_result = None
    # Consider all the possible ways to choose the last coin.
    for coin in coins:
        # Solve a subproblem for the rest of the amount.
        partial_result = make_change(coins, amount - coin)
        # Skip this coin if the payment failed:
        if partial_result is None:
            continue
        candidate = partial_result + [coin]
        # Check if the candidate solution is better than the
        # optimal solution known so far, and update it if needed.
        if (optimal_result is None or
            len(candidate) < len(optimal_result)):
            optimal_result = candidate
    return optimal_result
```

This algorithm implements the recurrence relation as explained above. Notice how we handle the edge cases at the beginning of the function, before making any recursive calls, to avoid infinite recursion and to keep the recursion logic as simple as possible.

Handle and eliminate edge cases early

Generally, it is preferrable to get edge cases out of the way as soon as possible, so that the rest of the implementation is kept simple. This improves substantially the readability of the code.

This implementation has exponential time complexity, since we have not implemented caching yet. Let's do that now.

Unfortunately, if we try to add caching simply adding the `lru_cache` decorator, we will have a nasty surprise:

```
from functools import lru_cache

@lru_cache(maxsize=None)
def make_change(coins, amount):
    ...
```

This code throws the exception: `TypeError: unhashable type: 'list'`. This is caused by the inability to cache the argument `coins`. As a list, it is mutable, and the `lru_cache` decorator rejects it. The decorator supports caching only arguments with immutable types, such as numbers, strings or tuples. This is for a very good reason: to prevent bugs in case mutable arguments are changed later (in case of lists, via `append`, `del` or changing its elements), which would require invalidating the cache.

A joke circulating in software engineering circles says that there are only two hard problems in computer science: cache invalidation, naming things, and off-by-one errors. To address the former, the design of the `lru_cache` decorator takes the easy way out: it avoids having to implement cache invalidation at all by only allowing immutable arguments.

Still, we need to add caching one way or another. We can work around the `lru_cache` limitation if we notice that we do not actually need to cache the `coins` list—that is shared among all subproblems. We only need to pass the remaining amount to be paid. So we write a helper function that solves the subproblem, taking as argument only the amount. The coin list is shared between invocations.

One way to implement this is to make the helper function nested inside the `make_change` function, so that it has access to the `coins` argument of the outer function:

```python
def make_change(coins, amount):
    @lru_cache(maxsize=None)
    def helper(amount):
        ...
    return helper(amount)
```

Another way is to transform the `make_change` function into a method of a class, that stores the list of coins in a class member. The helper could then be written as another method of the class, ideally a private one. I think adding classes is overkill for what we have to do, so we will not discuss this approach.

Here is the full solution that uses nested functions:

```python
def make_change(coins, amount):
    """
    Given a list of coin values, and an amount to be paid,
    returns the shortest list of coins that add up to that amount.
    If the amount to be paid is zero, the empty list is returned.
    If the amount cannot be paid using the coins, None is returned.
    """
    @lru_cache(maxsize=None)
    def helper(amount):
        # Handle payment of amount zero.
        if not amount:
            return []
        # Negative amounts cannot be paid.
        if amount < 0:
            return None
        optimal_result = None
        # Consider all the possible ways to choose the last coin.
        for coin in coins:
            # Solve a subproblem for the rest of the amount.
            partial_result = helper(amount - coin)
            # Skip this coin if the payment failed:
            if partial_result is None:
                continue
            candidate = partial_result + [coin]
            # Check if the candidate solution is better than the
            # optimal solution known so far, and update it if needed.
            if (optimal_result is None or
                len(candidate) < len(optimal_result)):
                optimal_result = candidate
        return optimal_result
    return helper(amount)
```

How many comments should we write?

Well-written comments improve code readability, however there is a trade-off: too many comments can be distracting, are a maintainability burden, and a sign that the code might not be clear enough. However, for interviews, I prefer leaning towards more verbosity, since we normally do not have time to refactor the code to perfection. Comments can compensate for that.

A good rule of thumb is to use comments to explain *why* the code is doing something. If you feel the need to explain *what* it is doing, it might be time to refactor that piece of code into an appropriately-named function. If you do not have time to refactor during the interview, you can add a comment like: “TODO: refactor into helper function.”

Solution 2: dynamic programming, bottom-up, $O(nv)$ time

Once we have implemented the top-down solution, we can rewrite it as bottom-up: we start from the amount 0, and keep adding coins in all possible ways until reaching the amount to be paid:

```
def make_change(coins, amount):
    # solutions[k] = optimal list of coins that add up to k,
    #                or None if no solution is known for k
    solutions = [None] * (amount + 1)
    # Initial state: no coins needed to pay amount 0
    solutions[0] = []
    # Starting from amount 0, find ways to pay higher amounts
    # by adding coins.
    paid = 0
    while paid < amount:
        if solutions[paid] is not None:
            for coin in coins:
                next_paid = paid + coin
                if next_paid > amount:
                    continue
                if (solutions[next_paid] is None or
                    len(solutions[next_paid]) >
                    len(solutions[paid]) + 1):
                    solutions[next_paid] = solutions[paid] + [coin]
        paid += 1
    return solutions[amount]
```

This solution is iterative, which is an advantage compared to the top-down solution, since it avoids the overheads of recursive calls. However, for certain denominations, it wastes time by increasing the amount very slowly, in steps of 1 unit. For example, if `coins = [100, 200, 500]` (suppose we use bills), it does not make sense to advance in steps of 1.

In addition, a lot of space is wasted for amounts that cannot be paid. Let's see if we can come up with a better solution.

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

We can optimize the bottom-up solution by using a queue of amounts to be handled, instead of an array. This helps in two ways: Firstly, it allows us to skip amounts that cannot be formed, thus reducing execution time. Secondly, by not having to store amounts that cannot be paid, memory usage is reduced.

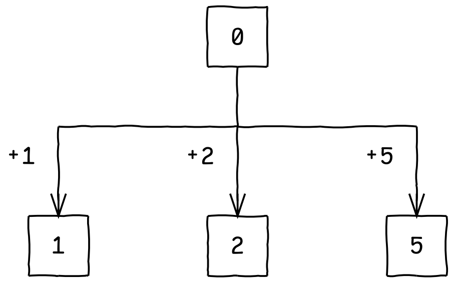
Let's implement it:

```
def simplest_change(coins, amount):
    # solutions[k] = optimal list of coins that add up to amount k
    # Amounts that cannot be paid are not stored.
    solutions = {0: []}
    # List of amounts that can be paid but have not been handled yet.
    amounts_to_be_handled = collections.deque([0])
    # Iterate over amounts in breadth-first order.
    while amounts_to_be_handled:
        paid = amounts_to_be_handled.popleft()
        solution = solutions[paid]
        if paid == amount:
            # Due to BFS order, the first path reaching the
            # required amount is the one using the smallest number
            # of coins. Thus it is the optimal solution.
            return solution
        for coin in coins:
            next_paid = paid + coin
            if next_paid > amount:
```

```
        # We can safely ignore amounts overshooting the
        # target amount to be paid.
        continue
      if next_paid not in solutions:
        solutions[next_paid] = solution + [coin]
        amounts_to_be_handled.append(next_paid)
  # No combination of coins could match the required amount,
  # thus it cannot be paid.
  return None
```

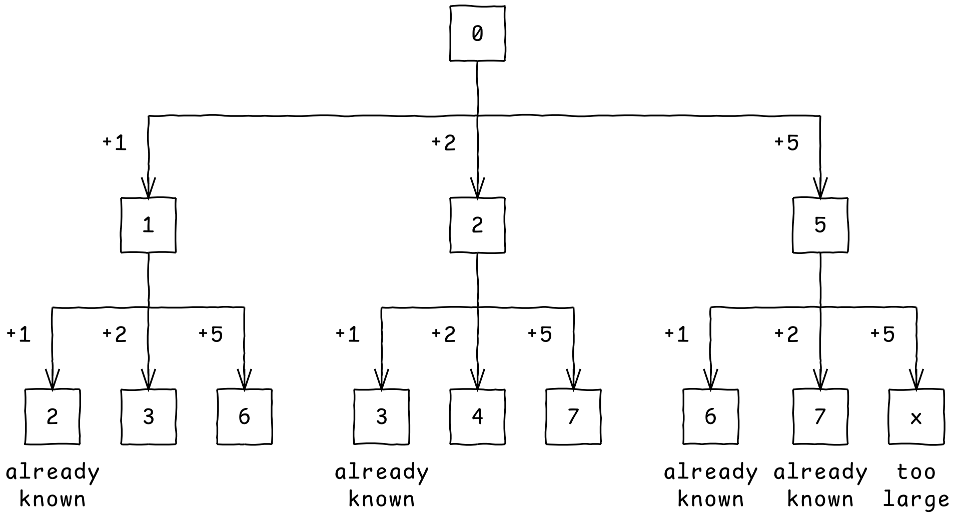
Notice how we are treating the (sub)problem space as a graph, which is explored in breadth-first order (BFS). We start from the subproblem of forming the amount 0. From there, we keep expanding using all the possible coin choices until we reach the required amount.

Let's look at an example showing how the problem space exploration works for paying amount 9 with coins `[1, 2, 5]`. We start with amount 0 and explore by adding a coin in all possible ways. Here is the first level of the problem space graph:



The first level contains all amounts that can be paid using a single coin. After this step, we have `[1, 2, 5]` in our BFS queue, which is exactly the list of nodes on the first level—hence the name of breadth-first search.

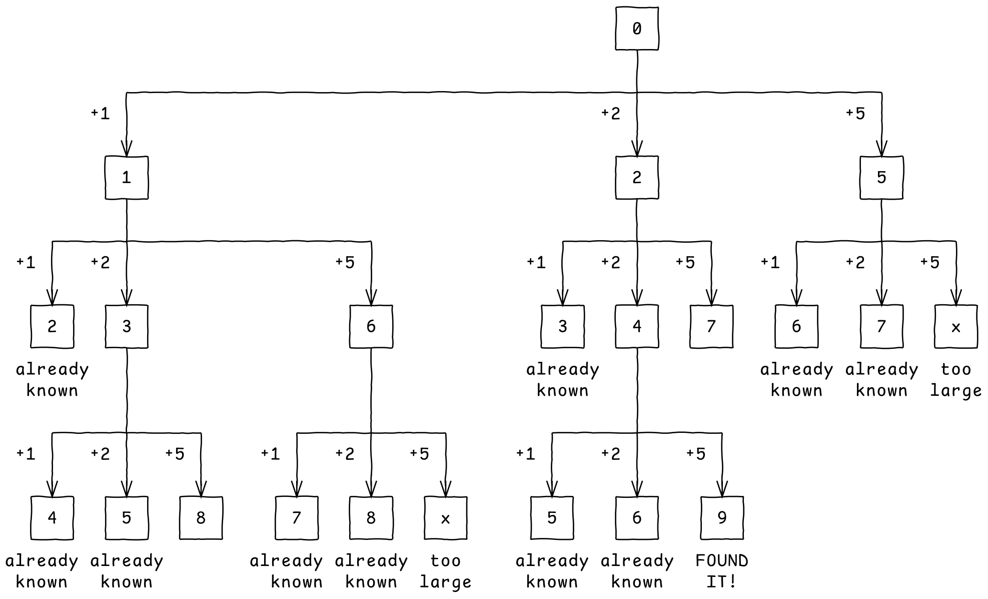
To fill in the second level of the problem space graph, we continue the exploration a step further for amounts 1, 2 and 5:



The second level contains all possible amounts that can be paid with 2 coins. After this step, we have `[3, 6, 4, 7]` in our BFS queue.

Notice that we discarded nodes corresponding to already known amounts, since these can be paid with a similar or better solution (fewer coins). We have also discarded amounts that exceed the value we have to pay (such as 10). This ensures that the graph size is bounded and that it contains only valid/optimal nodes.

We still have not found a way to pay our target amount, 9. Let's explore further, adding the third level of the problem space graph:



Note that as we reached the target amount 9, we stopped drawing the rest of the level. As it is on the third level of the tree, the amount can be paid with 3 coins. The combination is

2, 2 and 5, which can be found by walking the path from 0 to 9. This is the solution to the problem.

Note that the exploration order was important: the breadth-first order guarantees that each time we reach a new amount, the path we followed is the shortest possible in terms of number of coins used. Had we used instead another graph search algorithm, such as depth-first search, the solution might not have been optimal.

Variant: count the number of ways to pay (permutations)

Given a money amount and a list of coin denominations, count in how many ways it is possible to pay that amount. The order matters, i.e. paying with a coin of 1 followed by a coin of 2 is considered distinct from paying with a coin of 2 followed by a coin of 1.

Example: To pay amount 3 with coins `[1, 2]`, there are 3 possible ways: `1 + 2`, `2 + 1` and `1 + 1 + 1`.

Solution: dynamic-programming, top-down, $O(nv)$

Let's analyze a slightly more complicated example: paying amount 6 with coins `[1, 2, 5]`. Here are all the possible ways to pay the amount:

- `6 = 5 + 1`
- `6 = 1 + 5`
- `6 = 2 + 2 + 2`
- `6 = 2 + 2 + 1 + 1`
- `6 = 2 + 1 + 2 + 1`
- `6 = 2 + 1 + 1 + 2`
- `6 = 1 + 2 + 2 + 1`
- `6 = 1 + 2 + 1 + 2`
- `6 = 1 + 1 + 2 + 2`
- `6 = 2 + 1 + 1 + 1 + 1`
- `6 = 1 + 2 + 1 + 1 + 1`
- `6 = 1 + 1 + 2 + 1 + 1`
- `6 = 1 + 1 + 1 + 2 + 1`
- `6 = 1 + 1 + 1 + 1 + 2`
- `6 = 1 + 1 + 1 + 1 + 1 + 1`

From this list we do not see any particular rule, other than enumerating all the permutations of the combinations of coins that add up to 6 (`1 + 5`, `2 + 2 + 2`, `2 + 2 + 1 + 1`, `2 + 1 + 1 + 1 + 1` and `1 + 1 + 1 + 1 + 1 +1`). This is not very helpful.

Let's list the payments again but sorted in lexicographic order:

- `6 = 1 + 1 + 1 + 1 + 1 + 1`
- `6 = 1 + 1 + 1 + 1 + 2`
- `6 = 1 + 1 + 1 + 2 + 1`
- `6 = 1 + 1 + 2 + 1 + 1`
- `6 = 1 + 1 + 2 + 2`

- `6 = 1 + 2 + 1 + 1 + 1`
- `6 = 1 + 2 + 1 + 2`
- `6 = 1 + 2 + 2 + 1`
- `6 = 1 + 5`
- `6 = 2 + 1 + 1 + 1 + 1`
- `6 = 2 + 1 + 1 + 2`
- `6 = 2 + 1 + 2 + 1`
- `6 = 2 + 2 + 1 + 1`
- `6 = 2 + 2 + 2`
- `6 = 5 + 1`

We can now see a pattern we can use to generate the list:

- Choose 1 as the first coin to pay and pay it. Then pay the rest of the amount (5) in all possible ways:
  - `5 = 1 + 1 + 1 + 1 + 1`
  - `5 = 1 + 1 + 1 + 2`
  - `5 = 1 + 1 + 2 + 1`
  - `5 = 1 + 2 + 1 + 1`
  - `5 = 1 + 2 + 2`
  - `5 = 2 + 1 + 1 + 1`
  - `5 = 2 + 1 + 2`
  - `5 = 2 + 2 + 1`
  - `5 = 5`
- Then choose 2 as the first coin to pay and pay it. Then pay the rest of the amount (4) in all possible ways:
  - `4 = 1 + 1 + 1 + 1`
  - `4 = 1 + 1 + 2`
  - `4 = 1 + 2 + 1`
  - `4 = 2 + 1 + 1`
  - `4 = 2 + 2`
- Finally, choose 5 as the first coin to pay and pay it. Then pay the rest of the amount (1) in all possible ways:
  - `1 = 1`

We found a rule that reduces the problem to a smaller subproblem. The subproblems can be solved using the same pattern. Let's show the second iteration:

- Choose 1 as the first coin to pay and pay it. Then pay the rest of the amount (5) in all possible ways:
  - Choose 1 as the second coin to pay and pay it. Then pay the rest of the amount (4) in all possible ways:
    - `4 = 1 + 1 + 1 + 1`
    - `4 = 1 + 1 + 2`
    - `4 = 1 + 2 + 1`
    - `4 = 2 + 1 + 1`
    - `4 = 2 + 2`

    - Choose 2 as the second coin to pay and pay it. Then pay the rest of the amount (3) in all possible ways:
        - `3 = 1 + 1 + 1`
        - `3 = 1 + 2`
        - `3 = 2 + 1`
    - Choose 5 as the second coin to pay and pay it. There is nothing left to be paid.
- Then choose 2 as the first coin and pay it. Then pay the rest of the amount (4) in all possible ways:
    - Choose 1 as the second coin to pay and pay it. Then pay the rest of the amount (3) in all possible ways:
        - `3 = 1 + 1 + 1`
        - `3 = 1 + 2`
        - `3 = 2 + 1`
    - Choose 2 as the second coin to pay and pay it. Then pay the rest of the amount (2) in all possible ways:
        - `2 = 1 + 1`
        - `2 = 2`
- Finally, choose 5 as the first coin and pay it. Then pay the rest of the amount (1) in all possible ways:
    - Choose 1 as the second coin to pay and pay it. There is nothing left to pay.

Notice that this method computes the number of ways to pay certain amounts, such as 4 or 3, several times. We have to cache the results to avoid exponential complexity due to the redundancies.

Now that we found a recursive rule, we can implement a top-down dynamic programming solution as follows:

```python
def count_ways_to_pay(coins, amount):
    @lru_cache
    def count_ways_helper(amount):
        # Nothing more left to pay: a single way.
        if amount == 0:
            return 1
        # Invalid payment: we paid too much so the amount that
        # remains is negative.
        if amount < 0:
            return 0
        num_ways = 0
        # Consider all the possible ways to choose the first coin:
        for first in coins:
            # Count the ways to pay the rest of the amount.
            num_ways += count_ways_helper(amount - first)
        return num_ways
    return helper(amount)
```

The time complexity is $O(nv)$, where n is the number of coins and v is the amount we have to pay.

Variant: count the number of ways to pay (combinations)

Given a money amount and a list of coin denominations, count in how many ways it is possible to pay that amount. The order does not matter, i.e. paying with a coin of 1 followed by a coin of 2 is considered the same as paying with a coin of 2 followed by a coin of 1, so it should be only counted once.

Example: To pay amount 3 with coins `[1, 2]`, there are 2 possible ways: `1 + 2` and `1 + 1 + 1`.

> **Counting combinations vs. permutations**
>
> Please pay attention and take your time to understand the difference between this problem and the permutation counting variant.

Solution: dynamic-programming, top-down, $O(nv)$

The given example is too short to find a clear pattern. Let's analyze a slightly more complicated example: paying amount 6 with coins `[1, 2, 5]`. There are 5 ways to pay the amount:

- `6 = 5 + 1`
- `6 = 2 + 2 + 2`
- `6 = 2 + 2 + 1 + 1`
- `6 = 2 + 1 + 1 + 1 + 1`
- `6 = 1 + 1 + 1 + 1 + 1 + 1`

Let's sort them in lexicographic order:

- `6 = 1 + 1 + 1 + 1 + 1 + 1`
- `6 = 1 + 1 + 1 + 1 + 2`
- `6 = 1 + 1 + 2 + 2`
- `6 = 1 + 5`
- `6 = 2 + 2 + 2`

> **Sorting solutions**
>
> Notice the solution sorting trick once again. This is very helpful to extract visually patterns that could be then generated through a recursive algorithm.

We can formulate the following rule that generates these payments:

- Consider coin 1. It can appear in the payment between 0 and 6 times. Consider each option:
  - Do not pay with 1 at all. Count the number of ways to pay 6 with `[2, 5]`:
    - `6 = 2 + 2 + 2`

- Pay with 1 once. Count the number of ways to pay the remaining amount (5) with `[2, 5]`:
  - `5 = 5`
- Pay with 1 twice. Count the number of ways to pay the remaining amount (4) with `[2, 5]`:
  - `4 = 2 + 2`
- Pay with 1 three times. Count the number of ways to pay the remaining amount (3) with `[2, 5]`:
  - Impossible.
- Pay with 1 four times. Count the number of ways to pay the remaining amount (2) with `[2, 5]`:
  - `2 = 2`
- Pay with 1 five times. Count the number of ways to pay the remaining amount (1) with `[2, 5]`:
  - Impossible.
- Pay with 1 six times. There is nothing left to pay => `1 + 1 + 1 + 1 + 1 + 1`.

We found a rule that allows us to reduce the problem to several smaller subproblems: choose a coin, and use it some number of times to pay a part of the amount; then pay the remaining amount with a smaller set of coins.

If it's not clear yet, let's look at the first two iterations when applying this rule:

Consider coin 1. It can appear in the payment between 0 and 6 times. Consider each option:

- Do not pay with 1 at all. Count the number of ways to pay 6 with `[2, 5]`:
  - Consider coin 2. It can appear in the payment between 0 and 3 times. Consider each option:
    - Do not pay with 2 at all. Count the number of ways to pay the remaining amount (6) with `[5]`:
      - Impossible.
    - Pay with 2 once. Count the number of ways to pay the remaining amount (4) with `[5]`:
      - Impossible.
    - Pay with 2 twice. Count the number of ways to pay the remaining amount (2) with `[5]`:
      - Impossible.
    - Pay with 2 three times. There is nothing left to pay => `2 + 2 + 2`.
- Pay with 1 once. Count the number of ways to pay the remaining amount (5) with `[2, 5]`:
  - Consider coin 2. It can appear in the payment between 0 and 2 times. Consider each option:
    - Do not pay with 2 at all. Count the number of ways to pay the remaining amount (6) with `[5]`:
      - `5 = 5`
    - Pay with 2 once. Count the number of ways to pay the remaining amount (3) with `[5]`:

        - Impossible.
      - Pay with 2 twice. Count the number of ways to pay the remaining amount (1) with `[5]`:
        - Impossible.
- Pay with 1 twice. Count the number of ways to pay the remaining amount (4) with `[2, 5]`:
  - Consider coin 2. It can appear in the payment between 0 and 2 times. Consider each option:
    - Do not pay with 2 at all. Count the number of ways to pay the remaining amount (4) with `[5]`:
      - Impossible.
    - Pay with 2 once. Count the number of ways to pay the remaining amount (2) with `[5]`:
      - Impossible.
    - Pay with 2 twice. There is nothing left to pay => `1 + 1 + 2 + 2`;
- Pay with 1 three times. Count the number of ways to pay the remaining amount (3) with `[2, 5]`:
  - Impossible.
- Pay with 1 four times. Count the number of ways to pay the remaining amount (2) with `[2, 5]`:
  - `2 = 2`
- Pay with 1 five times. Count the number of ways to pay the remaining amount (1) with `[2, 5]`:
  - Impossible.
- Pay with 1 six times. There is nothing left to pay => `1 + 1 + 1 + 1 + 1 + 1`.

We can implement this rule using a helper function that takes as parameters the amount left to be paid and the index of the next coin to be considered (assuming coins are stored in a list).

We can see from the examples above that there are a lot of redundant computations, so we cache the results to reduce the time complexity. We will analyze how much after we write the solution.

We can write the solution as:

```python
def count_ways_to_pay(coins, amount):
    @lru_cache
    def count_ways_helper(index, amount):
        # Nothing more left to pay: a single way.
        if amount == 0:
            return 1
        # Invalid payment: we either paid too much so the amount that
        # remains is negative, or we still have something to pay but
        # we ran out of coins.
        if amount < 0 or index == len(coins):
            return 0
        num_ways = 0
        coin = coins[index]
        # Consider all the possible amounts to pay with the coin:
        for repeats in range(amount // coin + 1):
            payment = coin * repeats
            # Count the ways to pay the rest of the amount.
            num_ways += count_ways_helper(index + 1, amount - payment)
        return num_ways
    return helper(amount)
```

The time complexity is $O(nv)$, where n is the number of coins and v is the amount we have to pay.