In this question, notice that you have to return a list of lists.
k is the *number* of integers you can use to sum up to n.
n is the *integer* upto which you have to sum up.

Base Case:
Obtaining this clarity, let's start with the base case. We have two options here:

Option A)
```
if k == 1:
    return [[n]]
```

In other words, here, we need to return all lists of 1 numbers that sum to `n`. There is only one such list, `[n]` . Therefore, we return a list of *that* one list, `[[n]]`.

Option B)
```
if k == 0 and n == 0:
    return [[ ]]
```

In this case, we need to return all lists of 0 numbers that sum to 0 .
There is only one such list, `[ ]`. Therefore, we return a list of *that* one list, `[[ ]]`.

Option A & B are two ways you can approach the base case. Option A catches the recursion when you're making lists of `k = 1` number and Option B catches the recursion when you're making lists of `k = 0` numbers.

In general, it's better to prefer the `k == 0` version i.e. Option B. Heuristically speaking, things work out better when you catch the recursion as late as possible. To be clear, not catching the recursion at `k == 1` and instead waiting to catch it one step later at `k == 0` is "later". This can sometimes lead to solutions that are simpler or require fewer special cases. For example, if we chose Option A i.e. `k == 1` case, we would've also needed to support Option B at some point i.e. `k == 0` case, and then we'd need a second special case. Moreover, if we wrote `k == 0` instead of `k == 0 and n == 0`, we'd be wrong. There is 1 list of 0 numbers that adds to 0 , `[ ]`, hence the return value should be `[[ ]]`. On the other hand, when `k == 0` but **`n != 0`** , the return value must be `[ ]` i.e. just an empty list, because there is *no* list of zero numbers that sums up to an integer other than 0. In other words, there is ZERO list of 0 numbers that adds up to some `n != 0`.

This is analogous to the success and failure base cases from `count_change`. When `coin == 0 and total > 0` , we return 0 because there are no ways to make change for `total > 0` with coins of size 0 or smaller. When `total == 0` , we return 1 because there is one way: no coins.

Recursive Case:

Since we hope to address every single integer before `n` (and including `n`), we take the range of `1` (since that's where we start from) until `n+1` to include the `n`th element.

Now, notice that `s` is one of the `k-1`st element lists summing to `n`. And since the core issue in *any* recursive problem is to build a recursive solution into a solution for your arguments, we wish to "*append*" `x` to that `k-1`st element lists summing to `n` to obtain a `k` element list of lists. We "append" `x` to `s` by writing `[x]`[1]. Since we want the concatenation to be valid, we write `[x]`.

[1] `s` is a list and `x` is an integer.