1 Coin change

Imagine we have a cash register and an amount we are trying to reach by combining coins from this register. The coin change problem revolves around finding all possible combinations to reach a certain amount.

We can assume to have an infinite amount of each denomination available. We represent the denominations of integers (the amount of cents), and combine them in a list.

```
amountsEuro :: [Int]
amountsEuro = [1, 2, 5, 10, 20, 50, 100, 200]
```

Furthermore, we define the helper `changesEuro` which applies the (yet to be defined) `changes` function to `amountsEuro` giving a function of type `Int -> [[Int]]`.

```
changesEuro :: Int -> [[Int]]
changesEuro = changes amountsEuro
```

1.1 Calculating combinations

Define a function `changes :: [Int] -> Int ->[[Int]]` that takes a list of denominations `[Int]` and an amount to reach, and gives back a list of all (unique) combinations that add up to the given amount. You may assume the amount is not negative (≥ 0).

Hints

- How many base cases are there?
- What is the difference between the empty list `[]` and a list with an empty list as only element `[[]]`?

Examples

```
Main> changesEuro 0
[[]]

Main> changesEuro 1
[[1]]

Main> changesEuro 2
[[1,1],[2]]

Main> changesEuro 10
[[1,1,1,1,1,1,1,1,1,1],[1,1,1,1,1,1,1,1,2],[1,1,1,1,1,1,2,2],[1,1,1,1,1,5],
[1,1,1,1,2,2,2],[1,1,1,2,5],[1,1,2,2,2,2],[1,2,2,5],[2,2,2,2,2],[5,5],[10]]
```

1.2 Order of denominations

Changing the order of the input denominations may change the *order* of the outputted combinations, but not the *amount* of combinations.

Let `amountsEuroRev` be `amountsEuro` reversed, and `changesEuroRev` be `changes` applied to `amountsEuroRev`.

```
amountsEuroRev :: [Int]
amountsEuroRev = reverse amountsEuro

changesEuroRev :: Int -> [[Int]]
changesEuroRev = changes amountsEuroRev
```

Make sure the following returns `True` for any input `i`:

```
checkReverse :: Int -> Bool
checkReverse i = (length $ changesEuro i) == (length $ changesEuroRev i)
```