1. Consider again the problem of *making change*. Given a list of "denominations" $(d_1 d_2 \ldots d_k)$ values and a positive integer n, the problem is to determine the number of ways that n can be written as a sum of the d_i. For example, if we consider US coins, (1 5 10 25), the number 11 can be written in 4 ways:

$$10+1,\ 5+5+1,\ 5+\underbrace{1+\cdots+1}_{6},\quad \text{and}\quad \underbrace{1+\cdots+1}_{11}.$$

   Note that we do not consider $10+1$ and $1+10$ as "different" ways to write 11: all that matters is the number of occurrences of each denomination, not their order. Write a SCHEME function `(change k l)` that returns the number of ways to express `k` as a sum of the denominations appearing in the list ℓ. This is a generalization of the version seen in class early on this semester.

2. Now that you can *count* the number of ways to give change, it is time to compute each way to give change! Write a SCHEME function `(make-change n den)` which outputs a list of change specifications. Namely, a call `(make-change 11 (list 25 10 5 1))` produces the list

```
((1 1 1 1 1 1 1 1 1 1 1) (1 1 1 1 1 1 5) (1 5 5) (1 10))
```

   Hints: a helper function `(helper n den cur)` that takes as input *cur*, a list of coins given out so far will surely come in handy! Also, the order in which the "options" appear in the top list is immaterial.

3. While getting the output above is helpful, it is not particularly readable. Indeed, the list `(1 1 1 1 1 1 5)` that states 6 pennies and 1 nickel could be far more readable as `((6 . 1) (1 . 5))` also stating 6 pennies and 1 nickel. That is, this is a list of pairs telling us how many of each denomination. Thankfully, the former can be translated into the latter by a conversion known as *run length encoding* that replaces sequences of an identical value by a pair giving the number of repetition of the value. Write a SCHEME function `(rle coins)` which, given a list of coins, returns a list of pairs denoting the number of repetitions of each sequence of consecutive values. As a last example, the list

```
(list 1 1 1 1 1 1 1 5 5 5 5 1 1 1 10 10 10 1 1 25 25 25 25 25 25)
```

   would be encoded as

```
((7 . 1) (4 . 5) (3 . 1) (3 . 10) (2 . 1) (6 . 25))
```

   Note how the tree sub-sequences of pennies are **not** collapsed into a single value. Those are kept as separate pairs. Naturally, this function only works for one element of the output from `make-change`.

4. Naturally, it would be nice to convert the entire output of `make-change` to the RLE format. Write a SCHEME function `(rle-all lcoins)` which, when given a list of coin changes, produces a list of RLE encoded coin-changes. For instance the call

```
(rle-all (make-change 11 '(25 10 5 1)))
```

produces

```
(((11 . 1)) ((6 . 1) (1 . 5)) ((1 . 1) (2 . 5)) ((1 . 1) (1 . 10)))
```

which is a list of 4 lists (since there are four ways to give change on 11 cents) where each list is an RLE encoding.

5. *Pearson's Coefficient* is widely used in the natural sciences as a measure of "correlation" between two variables. For example, it is reasonable to expect that human height and weight are correlated, which is to say that—on average—taller humans tend to weigh more. As an example, consider the heights and weights of a small population of humans, as shown in the table below:

	A	B	C	D	E	F	G
height (cm)	160	186	172	202	177	186	191
weight (kg)	51	79	69	100	66	80	83

Examining the data you can see that, as a rule, weight *does* increase with height, though there are some exceptions.

The Pearson coefficient (of a collection of samples with two features, like height and weight above) is always between -1 and 1. A Pearson coefficient of exactly 1 (or exactly -1) indicates a perfectly linear relationship between the two quantities, whereas a coefficient near zero indicates that there is no such nice relationship. The data above has a Pearson coefficient of $\approx .98$, indicating an extremely strong relationship between the variables.

Given two lists, $X = (x_1\ x_2\ \ldots\ x_n)$ and $Y = (y_1\ y_2\ \ldots\ y_n)$, each containing n values, Pearson's Coefficient is defined by the expression:

$$\frac{\sum_{i=1}^{n}(x_i-\bar{X})(y_i-\bar{Y})}{\sqrt{\sum_{i=1}^{n}(x_i-\bar{X})^2}\sqrt{\sum_{i=1}^{n}(y_i-\bar{Y})^2}},$$

where $\bar{X}$ and $\bar{Y}$ denote the averages of the x_i and the y_i, which is to say that

$$\bar{X} = \frac{1}{n}\sum_i x_i \quad \text{and} \quad \bar{Y} = \frac{1}{n}\sum_i y_i\,.$$

(a) Write a function, `list-sum`, that takes a list and returns its sum.

(b) Write a function that takes a list $X = (x_1\ x_2\ \ldots\ x_n)$ and returns the average $\bar{X}$. (Note: You will have to compute the length of the list in order to compute the average.) Call your function `average`.

(c) Write a function, `var-map`, that *maps* a list X to the "square of its deviation." Thus the list $X = (x_1\ x_2\ \ldots\ x_n)$ should be carried to the list

$$((x_1-\bar{X})^2\ \ldots\ (x_n-\bar{X})^2)\,.$$

You should use the `map` function. (For example, your function, when evaluated on the list (1 2 3 4 5), should return (4 1 0 1 4).)

(d) Write a function `stdev` that returns the *standard deviation* of a list. Specifically, given the list $(x_1 \ldots x_n)$, your functions should return

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{X})^2}.$$

This should be easy, you already have the functions you need.

(e) Write a new version (called `map2`) of the `map` function which operates on two lists. Your function should accept three parameters, a function f and two lists X and Y, and return a new list composed of the function f applied to corresponding elements of X and Y. For concreteness, place the function first among the parameters, so that a call to the function appears as `(map2 f X Y)`. Your function may assume that the two lists have the same length. In particular, given two lists $(x_1\ x_2\ \ldots\ x_n)$ and $(y_1\ y_2\ \ldots\ y_n)$ (and the function f), `map2` should return

$$(f(x_1, y_1) \ldots f(x_n, y_n)).$$

(f) Write a function, `covar-elements`, that, given two lists $X = (x_1\ x_2\ \ldots\ x_n)$ and $Y = (y_1\ y_2\ \ldots\ y_n)$, returns the covariance list:

$$((x_1 - \bar{X})(y_1 - \bar{Y})\ \ldots\ (X_n - \bar{X})(Y_n - \bar{Y})).$$

Note that the resulting list should have the same length as the two input lists; the ith element of the resulting list is the product $(x_i - \overline{X})(y_i - \overline{Y})$.

(g) Write a function, `pearson`, that computes Pearson's Coefficient. It might be useful to observe that an equivalent way to write Pearson's coefficient, by dividing the top and bottom by n, is

$$\frac{{}^1\!/_n \sum_{i=1}^{n}(x_i - \bar{X})(y_i - \bar{Y})}{\sigma(X)\sigma(Y)}.$$

(Thus, your function should take two lists as arguments, and return the value indicated above.)

6. *Least-Squares Line Fitting.* Line fitting refers to the process of finding the "best" fitting line for a set of points. This is one of the most fundamental tools that natural scientists use to fit mathematical models to experimental data.

As an example, consider the red points in the scatter plot of Figure 1. They do not lie on a line, but there is a line that "fits" them very well, shown in black. This line has been chosen—among all possible lines—to be the one that minimizes the sum of *squares* of the vertical distances from the points to the line. (This may seem like an odd thing to minimize, but it turns out that there are several reasons that it is the "right" thing minimize.)

Since the equation of a line is $y = ax + b$, this boils down to finding a slope a and x-intercept b for given lists X and Y corresponding to point data. It turns out that the best fitting line can be found with the following two equations:

$$a = r \cdot \frac{\sigma(Y)}{\sigma(X)}, \quad \text{and} \quad b = \bar{Y} - a\bar{X},$$

where $\sigma(X)$ and $\sigma(Y)$ refer to the standard deviations of X and Y and r is Pearson's Coefficient.

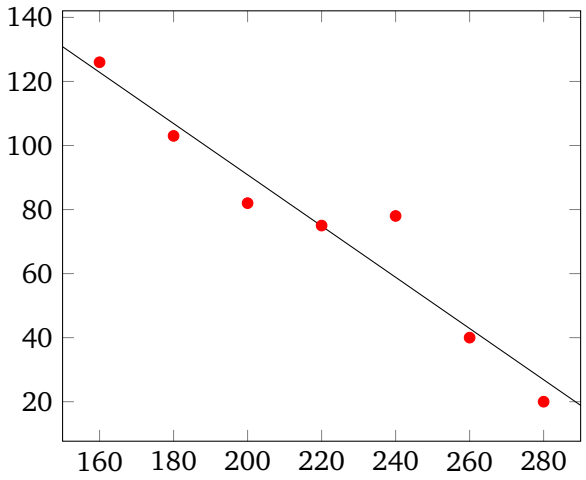

Figure 1: An example of a scatter plot and a best-fit line.

(a) Write a function `best-fit` that takes two lists X and Y (of the same length) and returns a pair (a, b) corresponding to the (slope and intercept of the) best fit line for the data.

(b) Write an alternate version of `best-fit` that returns the best fitting line as a *function*. Specifically, write a new function `best-fit-fn` so that `(best-fit-fn pX pY)` returns the *function* which, given a number x, returns $ax + b$ where a and b are the best fit coefficients.

Use this to the best fit line for the following data:

$$X = \{160, 180, 200, 220, 240, 260, 280\}, \qquad Y = \{126, 103, 82, 75, 78, 40, 20\}.$$

Call the function `fitline` so that you can graph it in the next problem.

7. Define a SCHEME function, `merge`, which takes two lists ℓ_1 and ℓ_2 as arguments. Assuming that each of ℓ_1 and ℓ_2 are *sorted* lists of integers (in increasing order, say), `merge` must return the sorted list containing all elements of ℓ_1 and ℓ_2.

   To carry out the merge, observe that since ℓ_1 and ℓ_2 are already sorted, it is easy to find the smallest element among all those in ℓ_1 and ℓ_2: it is simply the smaller of the first elements of ℓ_1 and ℓ_2. Removing this smallest element from whichever of the two lists it came from, we can recurse on the resulting two lists (which are still sorted), and place this smallest element at the beginning of the result.

8. There is a sorting algorithm that one can build from the SCHEME function `merge`. It is aptly named *merge sort*. Its architecture is based on the simple principle known as "divide and conquer" (like `quickSort`, covered in class). It works as follows: given a list ℓ, split the list into two sub-lists ℓ_1 and ℓ_2 of approximately the same length (a difference of 1 at most) such that all elements of ℓ appear in either ℓ_1 or ℓ_2. For instance, a list $\ell =$`(1 6 7 3 9 0 2)` could be split into $\ell_1 =$`(1 7 9 2)` and $\ell_2 =$`(6 3 0)`. Then one can recursively sort ℓ_1 and ℓ_2 to obtain sorted versions ℓ_1' and ℓ_2' and *merge* them to recover a fully sorted list. Write a SCHEME function `(mergeSort l)` which, given an unsorted list ℓ of integers, returns a sorted version of ℓ's content. **Hint:** writing a helper function to carry out the splitting would be a wise first step. Hiding that helper function in the bowels of `mergeSort` would be even better! A whimsical illustration of mergeSort is shown in Figure 2.

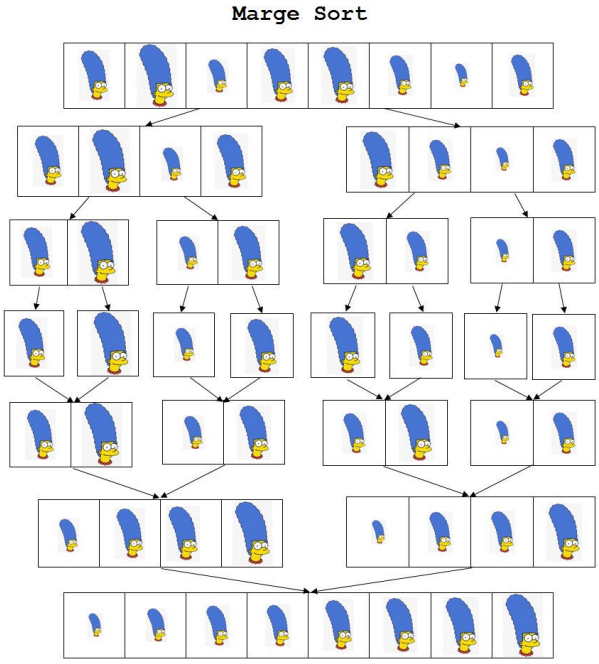

Figure 2: `mergeSort` at work!