“Statistics” Case Study

Problem

Write and test functions to compute the following statistics for a nonempty list of numeric values:

- The *mean*, or average value, is computed by dividing the sum of the values by the number of values in the list. In mathematical notation, this is represented as

  $$\frac{x_1 + x_2 + \ldots + x_n}{n}$$

  where `n` is the number of values in the list—`n` is at least 1—and `x`$_1$, `x`$_2$, …, `x`$_n$ are the values in the list.
- The *median* is the middle value in the list when the list’s values are arranged in numerical order. If the list has an even number of values, either of the two middle values may be chosen as the median. (This may differ from conventions you have used in other contexts.)
- The *range* of values in the list is a pair of values from the list, namely the minimum and maximum values.
- The *standard deviation*, a measure of how the values are spread out, is computed using one of the two formulas described below.

  Let the variables `m` and `sqm` represent the following quantities:

  `m` = the mean value in the list;

  `sqm` = the mean of the squares of values in the list, that is,

  $$\frac{x_1^2 + x_2^2 + \ldots + x_n^2}{n}$$

  in mathematical notation.

  One formula for the standard deviation is represented by the mathematical expression

  $$\sqrt{sqm - m^2}$$

  The other formula is

  $$\sqrt{\frac{(x_1 - m)^2 + (x_2 - m)^2 + \ldots + (x_n - m)^2}{n}}$$

  Intuitively, this is measuring how far, on the average, the values are from the mean.

Preparation

The reader should have experience with defining Scheme functions and constants and with using the following Scheme constructs: conditional expressions, lists, the built-in functions first, second, member, and assoc, the lambda special form, and the functionals map, apply, accumulate, find-if, remove-if, and keep-if.

Exercises

Application 1. Compute the mean, median, and standard deviation for the following set of values:

```
5  1  4  2  3
```

Use both formulas to compute the standard deviation.

Application 2. Do the same for the following:

```
4  3  3  3  4  4
```

As in problem 1, use both formulas to compute the standard deviation.

Analysis 3. Devise two sets of five values with the same mean, one having a very small standard deviation, the other having a very large standard deviation.

Reflection 4. Which of the two formulas for the standard deviation is easier for a person to use? Which is likely to be easier to implement in a program?

Analysis 5. Convince someone else that the two formulas for the standard deviation compute the same result. Why is your argument convincing?

Analysis 6. Which statistic—the mean or the median—is more informative for the following set of values?

```
1  1  1  1  2  2  2  2  5000
```

Analysis 7. Give a set of values for which the average of the squares of the values is different from the squared average of the values. Give a set of values for which the two quantities are equal.

Application 8. A shorthand called "sigma notation" or "summation notation" is often used in mathematics textbooks to represent sums. In this notation,the sum `x1` + `x2` + … + `xn` is represented as

$$\sum_{1 \le k \le n} x_k$$

The **Σ** is the Greek letter sigma, corresponding to our letter S. It stands for "Summation". The expression below the **Σ** represents the sequence of subscripts used in the expressions being added; in the above example, that's the integers starting at 1 and ending at `n`. One reads the above expression as "the sum of all `xk` for `k` running from 1 to `n`".

Use summation notation to represent the formula for `sqm` and the second formula for standard deviation.

Analysis 9. Discuss, for each statistic, the accuracy of the information it conveys about the list of values, and describe lists of values for which the statistic may be misleading.

Analysis 10. Give a reason for *squaring* the distance each value is from the mean in the second formula for standard deviation. Hint: Compute

$$\frac{(x_1 - m) + (x_2 - m) + (x_3 - m)}{3}$$

for the list 1, 2, 3.

Solving the "easy" parts of the problem

How can the problem be subdivided?

The problem statement says to compute four statistics for a list of values: the mean, the median, the range (this is actually a list of two values), and the standard deviation. There are no obvious dependencies among them, except that the formula for standard deviation involves computing the mean, so we decide to solve the problem as four separate subproblems.

What are the easy parts of this problem?

It makes sense to do the easy subproblems first. A problem is "easy" to solve if a programmer has a pattern for its solution that can be recycled. Solving the easy subproblems first provides a sense of security and yields partial solutions quickly.

Which of the statistics will be easy to compute? Probably the standard deviation will not be easy, since its formula is complicated. Computing the mode involves counting all the occurrences of each item, which seems somewhat difficult. Finding the median involves ordering the elements to find the middle one. We'll put that off too. The mean, maximum, and minimum, however, are easier to deal with, so we'll start with those.

How can the builtin `max` and `min` functions be used for this problem?

The `max` and `min` functions are already defined in Scheme. Each takes any number of numeric arguments; `max` returns the largest of its arguments, and `min` returns the smallest. For instance, we can evaluate

```
(max 3 -1 51 48 2)
```

and get 51.

For this problem, however, we're given not individual values but a list of values. The max function, when given a list, can't deal with it:

```
(max '(3 -1 51 48 2))
ERROR: number expected
```

If we knew there were, say, three numbers in a list L, we could provide those values as arguments:

```
(max (car L) (cadr L) (caddr L))
```

but the problem statement hasn't limited the size of the lists our solution is to handle.

How can the apply functional be used for this problem?

The apply functional is designed precisely for this purpose. It takes two arguments, a function and a list of arguments for that function. It essentially returns the result of inserting the function at the beginning of the list and then evaluating the result. Thus

```
(apply max '(3 -1 51 48 2))
```

does the equivalent of moving the max into the list, giving

```
(max 3 -1 51 48 2)
```

and then evaluating that, giving 51. Apply can be used in a function list-max as follows:

```
; Return the largest value in the list L.
(define (list-max L)
   (apply max L) )
```

The computation of the smallest element in a list is similar; we call it list-min.

Stop and help (*Provide the definition for* list-min.

Stop and predict (*Give an expression that computes the sum of the values in a list L of numbers.*

Stop and consider (*Suppose a function* my-max *were defined as follows:*

```
(define (my-max a b)
   (if (> a b) a b) )
```

What would be the result of evaluating the expression

```
(apply my-max '(3 -1 51 48 2))
```

How is the mean computed?

We move to the mean. As described in the problem statement, the mean is the sum of the values divided by the number of values. The sum of the values can be computed with apply in the same way as the maximum was:

```
(apply + '(3 -1 51 48 2))
```

has the same result as

```
(+ 3 -1 51 48 2)
```

namely, 103. We therefore code a function list-sum,

```
(define (list-sum L)
   (apply + L) )
```

and use it in a function to compute the mean:

```
(define (mean L)
   (/ (list-sum L) (length L)))
```

How should these functions be tested?

All these functions are short. Even so, we may have made typing errors, so we stop to test them here.

We test mean on a "typical" list for which the answer is easy to compute by hand, and test list-max and list-min in the same way. We also check what happens for all the functions in the "extreme" situation where the list of values contains only one element. (The problem statement guarantees that the list of values contains at least one element.)

Stop and consider (*For which list of ten values is the mean most easily computed by hand?*

Stop and help (*Test the* mean *function.*

Stop and consider (*What does* mean *return, if anything, when given an empty list as argument?*

What is the next step?

Appendix A contains the code that computes the mean and the range. Looking down the list of remaining functions, we note that the standard deviation merely involves arithmetic on means. We already have a function to compute the mean, so we work on standard deviation next.

How should the standard deviation be computed?

The problem statement gives two formulas for the standard deviation. The second formula appears to be long and complicated. The first formula, however, is the square root of the difference of two quantities:

the mean of the squared values, and
the square of the mean of the values.

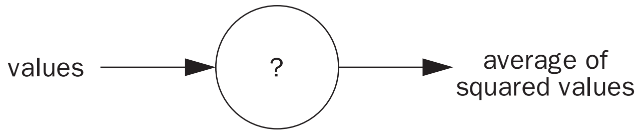
In Scheme, we have the following:

```
(define (std-dev values)
   (sqrt
      (- (mean-of-squared values)
         (square (mean values)) ) ) )
```

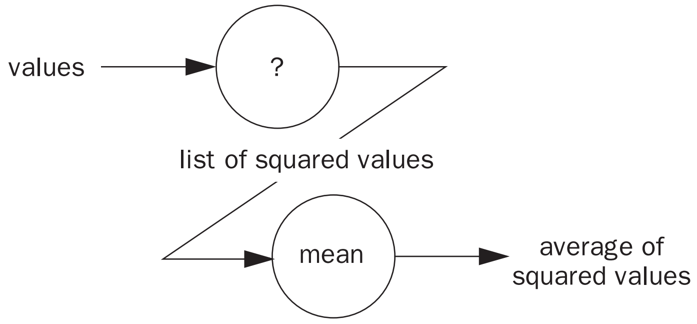
where the sqrt function is built in, the square function is easy to define, and the mean-of-squared function requires further design.

How should the mean of the squared values be computed?

A ***data flow diagram*** helps to keep track of the action. In a data flow diagram, functions are represented as circles and their arguments and returned results are represented as labeled arrows. The diagram appears below.



We already have a function that computes a mean. Reasoning backward from the mean function, we see that to compute the average of the squared values we need a list of squared values to provide as argument to mean. Here is a diagram:



Review of the functionals suggests that map will be appropriate. The list of squared values has just as many elements as the values list, and each element of the list of squared values is the result of applying the square function to its counterpart in values. Here is a function that uses map to compute the list of squared values:

```
(define (squared values)
   (map square values) )
```

This allows us to complete the data flow diagram and code std**-dev** as follows:

```
(define (std-dev values)
   (sqrt
      (- (mean (squared values))
         (square (mean values)) ) ) )
```

Appendix B contains the code.

Stop and consider (*Why should* squared *be coded as a separate function rather than using the expression* (map square values) *within* std-dev*?*

How should std-dev **be tested?**

There are two keys to testing a complicated function like std-dev. The first is to test the parts individually before testing them in combination. Just as a contractor or engineer requires reliable parts to build a house or a bridge, so should a programmer develop reliable ***components*** of a program before assembling them. The second key is to select test values with easily checked answers.

To test the parts individually, we test the square function and the squared function by themselves. (We've already tested mean.)

For std-dev, an easy-to-compute case is an extreme case. The problem statement notes that the standard deviation is a measure of how much the values are spread out, so it should be 0 when all the values are the same. We should check two or three such cases, including a case with a value list of just one value.

Next we consider value lists whose standard deviation should be 1. Such lists are more easily found by using the second formula from the problem statement. After a bit of algebraic manipulation, we note that when each value differs from the average by 1 or –1, the standard deviation should be 1. The values 1, 1, 3, 3 make up one such set.

Finally, we consider lists of values whose standard deviation is not an integer. We use the first formula to compute the answers by hand, since it provides an *independent* way to check the answer (the program implements the second formula, so redoing its computation by hand would provide somewhat less convincing evidence for correctness). We use small lists of values—three or four values each—to minimize the pain of the hand computation.

Stop and help (*Compute the standard deviation for the values 1, 4, 1, then use these values as test data for the* `std-dev` *function.*

Note that correct results on the test data do not ***prove*** that the function is indeed correct. Chosen systematically, however, with both extreme cases and "typical" cases, they can provide substantial evidence.

Exercises

Analysis 11. Suppose that three values each differs from its mean by the same amount. Show that they must all be identical.

Analysis 12. Briefly explain the result of evaluating

```
(apply > L)
```

for a list `L` of numbers.

Application 13. Write a set of functions that computes the standard deviation using the second formula given in the problem statement:

$$\sqrt{\frac{(x_1 - m)^2 + (x_2 - m)^2 + \ldots + (x_n - m)^2}{n}}$$

Analysis 14. How can the functions from the previous exercise help in testing the code designed so far in the case study?

Application 15. `Length` is a predefined function in Scheme. If it were not, however, it could be implemented using `map` and `apply`. Supply the function that could be passed as an argument to `map` to help compute the length of a list, by filling in the blank below.

```
(define (length L)
   (apply +
      (map _____ L) ) )
```

One way to compute the median

How should the median be computed?

The median was defined as the middle value when the values are arranged in order. In a list without duplicate values, it's the value that's greater than half the other values in the list. Each definition suggests an approach to computing the median.

Stop and predict (*Which of the two definitions of the median is likely to be easier to implement in Scheme?*

Stop and predict (*Write a function called* half-length *to find the number of values that will be less than the median.*

The first definition requires putting the values in order, while the second definition does not. That suggests that a solution based on the second definition is likely to be somewhat easier. Thus, even though the solution will only work with a list with no duplicate values, we will still start with that approach. We are confident that the solution will be easy to modify to work with lists with duplicates as well.

How can the value that's larger than exactly half the other values in the list be found?

"Finding the value that …" suggests using the find-if functional:

```
(define (median values)
   (find-if
      ________________
      values))
```

Find-if returns the first value in the list that satisfies its predicate argument. In this situation, the predicate should return true if a value is greater than half the elements of values. Said differently, it returns true if the number of elements that are less than the value is equal to half the length of the list.

How is a function to find the number of list elements less than a given value used to determine the median?

We thus need a function to find the number of list elements less than a given value. We'll call it how-many-< (read "how many less than"), and use it to design a predicate function possible-median? as follows:

```
(define (half-length L)
   (truncate (/ (length L) 2)) )

(define (possible-median? x values)
   (= (how-many-< x values)
      (half-length values) ) )
```

```
(define (median values)
   (find-if
      (lambda (x)
         (possible-median? x values) )
      values) )
```

Stop and help (*Why is it necessary to use* truncate *in* half-length*?*

Stop and help (*Why is it necessary to use* lambda *notation for the predicate function in* median*? I.e., why couldn't* median *be written as*

```
(define (median values)
   (find-if possible-median? values) )
```

How is the number of list elements less than a given value found?

To find the number of list elements less than a given value, we first define a function smaller-vals that selects those elements from the list. It uses keep-if as follows:

```
(define (smaller-vals x values)
   (keep-if
      (lambda (y) (< y x))
      values) )
```

The number of smaller values is just the length of the resulting list:

```
(define (how-many-< x values)
   (length (smaller-vals x values)) )
```

How should all these functions work together?

Summarizing, we have the following. Finding the median consists of using find-if to find a possible median. The function possible-median? checks for a possible median, which is an element for which the number of smaller elements in the list is equal to half the length of the list. The function how-many-< finds the number of elements in the list smaller than a given value, and the function half-length returns half the length of a list. Appendix C contains the code.

We check this by hand on an example list, big enough to involve a significant amount of computation, but not so big that we lose track of what's going on. The list (5 1 4 3 2) seems large enough; since its median is the fourth element, that should provide enough evidence that our approach at least is on the right track. Here are the steps of the "desk check". Indentation shows calls of subsidiary functions.

Evaluate (median '(5 1 4 3 2)):
Find a possible median by checking first 5, then 1, and so on until one is found.

1. Evaluate (possible-median? 5 '(5 1 4 3 2)):
   Check how many elements in the list are less than 5, and see if that's half the number of elements.
   - Evaluate (how-many-< 5 '(5 1 4 3 2)):
     Using smaller-vals, construct the list (1 4 3 2), and return its length, 4.
   - Evaluate (half-length '(5 1 4 3 2)):
     Return 2.
   - $4 \neq 2$, so 5 isn't a possible median.
2. Evaluate (possible-median? 1 '(5 1 4 3 2)):
   Check how many elements in the list are less than 1, using how-many-<. How-many-< returns 0; half-length returns the same thing it did before, which was 2. (Details are omitted.) $0 \neq 2$, so 1 isn't a possible median.
3. Evaluate (possible-median? 4 '(5 1 4 3 2)):
   There are 3 elements in (5 1 4 3 2) that are less than 4. $3 \neq 2$, so 1 isn't a possible median.
4. Evaluate (possible-median? 3 '(5 1 4 3 2)):
   There are 2 elements in (5 1 4 3 2) that are less than 4, so possible-median? returns true, and median correspondingly returns 3.

Stop and help (*Trace through the application of* median *to the list* (4 1 3 2).

How should median **be tested?**

So far, so good. We must now thoroughly test median online. Several aspects of the code suggest possibilities for error:

- The integer division in half-length provides the possibility of an "off-by-one" error. Testing this involves using value lists both of odd and of even length.
- Another source of off-by-one errors is the comparison in smaller-vals. Sometimes programmers say "<" when they mean "<=" or vice-versa. Guarding against this error requires devising ***boundary*** test values.
- Finally, programmers (even experts!) sometimes get confused, and accidentally reverse the sense of a comparison, using "<" when they mean ">" or vice-versa. Test data that displays this error will be easy to devise, but we must be sure to test the functions individually to ensure that we find the error quickly.

Good testing practice in general suggests using extreme values as well as typical values. There are several ways in which a value can be "extreme" in a list: it can be at the beginning or end of the list, or it can be the largest or smallest value of the list. We test median with value lists containing the median at various positions.

Stop and predict ⟨ *Is it true that the median can't also be the largest value? Explain.*

From the principles above, we generate the following test arguments for the various functions:

function	*test argument*
half-length	*list of odd length; list of even length; list of length 1*
smaller-vals	*a value no smaller than the largest value in the list; a value no larger than the smallest value in the list; some other value in the list*
how-many-<	*same as* smaller-vals
median	*median at the beginning of the list; median at the end of the list; median somewhere in the middle*

How can map **be used to test** median **more easily?**

A good way to test possible-median? is to wrap a map around it. The resulting function will look just like median, except for having map in place of find-if:

```
(define (test values)
   (map
      (lambda (k)
         (possible-median? k values) )
      values) )
```

Thus

```
(test '(5 1 4 3 2))
```

should return

```
(#f #f #f #t #f)
```

This shows the result for *all* elements of the values list, allowing us both to check several test cases at once and to make sure that possible-median? doesn't ***always*** return true.

Stop and help ⟨ *Design test data for the various functions used to compute the median, and test the functions.*

Now we try the function on a list that contains *duplicate* values. The extreme case for this situation is where the values are all the same, and for this the median fails miserably. The fix is not too difficult, however; it is left as a study question.

Exercises

Analysis

16. Why does median not work on some lists that contain duplicate values?

Testing, analysis

17. Provide an argument to median that does contain duplicate values but for which the correct median value is returned.

Analysis

18. Identify, as completely as possible, those lists for which median doesn't work.

Debugging

19. Fix median. Hint: Think about using both a "greater than or equal to" and a "less than or equal to" test in the argument to find-if.

Analysis

20. If there are an even number of values in the list passed to median, which of the two "middle" elements gets returned as the median?

Analysis

21. Possible-median? as written does some needless computation. Describe the source of the inefficiency.

Application

22. Use map to test the mean function on all of the lists (1 2 3), (4 5), and (2) at once.

An alternate way to compute the median

How can the other approach to finding the median be implemented in Scheme?

The other approach to finding the median was to arrange the values in order, then return the middle element of the list. In Scheme, we have

```
(define (median-by-sorting values)
   (list-ref
      (sort values)
      (half-length values) ) )
```

where sort is a function we have to write.

Stop and help (

Which of the two "middle" elements is returned by half-length *applied to a list with an even number of elements?*

Why even worry about finding another way to compute the median?

One might ask what good another method of computing the median would do. There are several reasons to look for an alternative implementation:

a. One reason is merely to acquire more experience with Scheme, and with functionals in particular. The functionals are so powerful and flexible that even experts find new uses for them after playing around for awhile.

b. We saw in computing the standard deviation that two implementations could be used to check each other's results.

c. Finally, we should always aim for code that's easier to test and understand. The `median-by-sorting` function looks promising in this respect, as long as the `sort` function isn't too complicated.

How should the list be sorted?

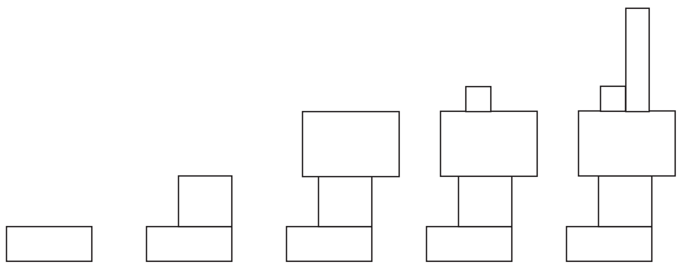
Sorting can be done recursively. To gain more practice with functionals, however, we choose to explore ways to use them to sort, and survey the functionals to see which ones might be helpful. `Find-if` returns a single element of a list. It could be given a list of all orderings of `values` and return one in which all the values are in order, but generating the list of all orderings seems like too much trouble. `Remove-if` and `keep-if` return only parts of a list, and thus seem inappropriate. `Map` returns a list of the correct length; however, it involves the application of a one-argument function to each element, and the sorting process will involve *comparison* of *pairs* of elements.

How can `accumulate` be used to help sort the list?

There are many ways to sort a list (entire books have been written on the subject!). One way is as an accumulation.

How can the computation of the mean, maximum, and minimum be viewed as accumulations?

An accumulation in real life is a collection of things, amassed one by one. Raindrops accumulate into a puddle. Blocks accumulate into a stack, as shown below.



successive accumulation of blocks

In programming, an accumulation is a quantity that we build out of data elements, one by one. Thus a sum can be an accumulation, built by adding elements one by one. The accumulate functional performs that one-by-one accumulation. (We noted before that apply essentially adds the numbers all at once.) For example, (accumulate + '(1 5 4 3 2)) performs the computation ((((1+5)+4)+3)+2), which, expressed in words, is

> Add 1 and 5.
> Add 4 to the result (the accumulated sum so far).
> Add 3 to that result.
> Add 2 to that result.

The maximum (or minimum) of a list of values can also be an accumulation, built by successively comparing each value to the maximum (or minimum) found so far. To compute the maximum value in the list, we might do the following:

> Compare 1 and 5.
> Compare 4 with 5 (the largest value so far).
> Compare 3 with 5 (the result of the second comparison).
> Compare 2 with 5 (the result of the third comparison).

That's just what (accumulate max '(1 5 4 3 2)) does.

How can sorting be viewed as an accumulation?

To see how this approach applies to sorting, we consider how a person might arrange a hand of cards. He or she would pick them up, one by one, and insert each on into the hand. The cards are thus being accumulated into the hand in order. When all cards are picked up, the entire hand is sorted.

Applying this technique to a list of numbers, say 3, 1, 9, 5, and 4, yields the following steps:

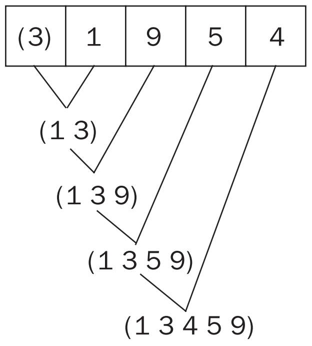
> Start out with 3.
> Insert 1 into the list (3), giving (1 3).
> Insert 9 into (1 3), giving (1 3 9).
> Insert 5 into (1 3 9), giving (1 3 5 9).
> Insert 4 into (1 3 5 9), giving (1 3 4 5 9).

(We arbitrarily assume that the list will be sorted in *increasing* order.)

We just saw accumulate used to find the sum of values and the maximum value in a list. Accumulate here is used somewhat differently. The accumulation isn't a number, it's a ***list***. Thus the order of arguments to the accumulating function is important (the value accumulated so far comes first). Also, the first element of the list argument must itself be an accumulated value, in this case a list. Here's the code:

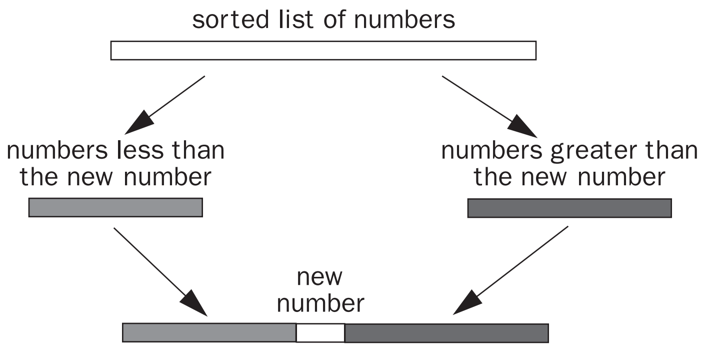
```
(define (insert L k)
   ______________ )

(define (sort L)
   (accumulate
      insert
      (cons
         (list (first L))
         (rest L) ) ) )
```

The insertion pattern is displayed in the tree diagram below.



How is insert **coded?**

Insert, given a list and a number, should return the result of inserting the number into the list in the correct position. Again, we could code it recursively but choose for this problem to find a way to use functionals. The insertion can be done by splitting the list into two pieces, the small elements and the large elements, and appending them around the number to insert, as in the diagram below.



sorted list of numbers
numbers less than the new number
numbers greater than the new number
new number

Stop and predict (*Have we forgotten anything? Hint: think back to the bug in the first version of* median.

A problem, the same one we encountered in the first version of median, is that there may be duplicate values in the list. Using "≤" for "less than" or "≥" for "greater than" (but not both!) solves the problem.

To break apart the list as just described, we use the function smaller-vals designed for the first version of median—slightly modified so that the order of arguments is consistent with that of insert—and a similar function derived from it:

```
(define (smaller+equal-vals L x)
   (keep-if
      (lambda (y) (<= y x))
      L))

(define (larger-vals L x)
   (keep-if
      (lambda (y) (> y x))
      L))
```

Insert, derived from the pseudocode and diagram, is coded as follows:

```
(define (insert L x)
   (append
      (smaller+equal-vals L x)
      (list x)
      (larger-vals L x)))
```

How should sort and its components be tested?

Appendix D contains the code for the rewritten median computation. We test in pieces, following good programming practice. By now, we've collected quite a number of "extreme test data" categories:

relevant element at the beginning;
relevant element at the end;
all elements identical;
element is the largest in the list;
element is the smallest in the list.

These should suggest test data for median-by-sorting, sort, insert, smaller+equal-vals, and larger-vals.

Stop and help (*Why is it necessary to specify* (list x) *rather than* x *as an argument to* append *in* insert*?*

Stop and help (*Design test data for all the functions just designed.*

Exercises

Analysis 23. What would be the undesired result of using "≤" for "less than" *and* "≥" for "greater than" in the insertion?

Analysis 24. Another way to solve the problem is to break the list into ***three*** parts: the values less than k, those greater than k, and those ***equal*** *to* k, These three lists can then be appended to form the proper result. Compare this approach to the one we took.

Application 25. Use map to simultaneously test the insertion of each of the values 1, 3, 5, and 7 into the list (2 4 6).

Modification 26. Rewrite the insert function to work with the following version of sort:

```
(define (sort L)
   (accumulate insert L) )
```

Outline of design and development questions

Solving the "easy" parts of the problem

How can the problem be subdivided?

What are the easy parts of this problem?

- How can the builtin max and min functions be used for this problem?
- How can the apply functional be used for this problem?
- How is the mean computed?
- How should these functions be tested?

What is the next step?

How should the standard deviation be computed?

- How should the mean of the squared values be computed?

How should std-dev be tested?

One way to compute the median

How should the median be computed?

- How can the value that's larger than exactly half the other values in the list be found?
- How is a function to find the number of list elements less than a given value used to determine the median?
- How is the number of list elements less than a given value found?
- How should all these functions work together?

How should median be tested?

- How can map be used to test median more easily?

An alternate way to compute the median

How can the other approach to finding the median be implemented in Scheme?

- Why even worry about finding another way to compute the median?
- How should the list be sorted?
- How can `accumulate` be used to help sort the list?
    - How can the computation of the mean, maximum, and minimum be viewed as accumulations?
    - How can sorting be viewed as an accumulation?
- How is `insert` coded?

How should `sort` and its components be tested?

Exercises

Modification

27. Suppose the values list provided to the various statistics functions is a list of *pairs*, each containing a name and a value. Here's an example:

```
((clancy 100)
 (linn 100)
 (wirth 65))
```

Modify the program(s) to return the desired statistics for the new argument format.

Reflection

28. What makes testing a function difficult in general? What made testing the functions of this case study difficult or easy?

Modification

29. Modify the `mean` function to return the average of all but the largest and smallest scores in its argument list. (This is done in some athletic competitions.) Thus the revised function will return 2 for the list (`8 1 2 3 –16`) as well as for the list (`3 2 1 3 –7`).

Application

30. Write a function named `second-max` that, given a list of numbers, returns the second-largest number in the list. If the largest value occurs two or more times in the argument list, it should be returned.

Debugging

31. Deleting one character somewhere in the functions to compute the median in Appendix C produces code that gives the error message "wrong number of arguments to `possible-median?`" when tested. Which character could have been deleted? (There are two possibilities; indicate both.)

Testing, reflection

32. What "rules of thumb" help programmers design test data?

Application 33. The ***mode*** of a list of values is the value that occurs most often in the list. How might techniques described in this case study help in designing a function to compute the mode?

Analysis 34. It has been said that "Liars often figure and figures often lie." How might a liar choose among the mean, the mode, and the median to communicate false information.

Modification 35. Explain how the program might be modified to alert users to potentially misleading information in the statistics that are computed.

Appendix A—functions to compute the mean and the range

```
; Return the sum of values in the list of numbers L.
(define (list-sum L)
   (apply + L) )

; Return the average value in the list of numbers L.
(define (mean L)
   (/ (list-sum L) (length L)) )

; Return the largest value in the list L.
(define (list-max L)
   (apply max L) )

; Return the smallest value in the list L.
(define (list-min L)
   (apply min L) )

; Return the range of values in the list of numbers L.
(define (range L)
   (list (list-min L) (list-max L)) )
```

Appendix B—functions to compute the standard deviation

```
; Return the square of the number x.
(define (square x)
   (* x x) )

; Return the list of squares of values, a list of numbers.
(define (squared values)
   (map square values) )

; Return the standard deviation for the numbers in values,
; a list of numbers.
(define (std-dev values)
   (sqrt
      (- (mean (squared values)) (square (mean values)) ) ) )
```

Appendix C—functions to compute the median for lists of *distinct* values

```
; Return the list of numbers in values that are smaller
; than x. x is a number, values is a list of numbers.
(define (smaller-vals x values)
   (keep-if (lambda (y) (< y x)) values) )

; Return how many numbers in values are smaller than x.
; x is a number, values is a list of numbers.
(define (how-many-< x values)
   (length (smaller-vals x values)) )

; Return half the length of the list L.
```

```
(define (half-length L)
   (truncate (/ (length L) 2)) )

; Determine if x is a possible median of values,
; a list of numbers.
; It is assumed that x is an element of values.
(define (possible-median? x values)
   (= (how-many-< x values) (half-length values)) )

; Find the median of values, a list of numbers.
; *** NOTE:
; This function does not work correctly when values
; contains duplicate numbers.
; ***
(define (median values)
   (find-if
      (lambda (x) (possible-median? x values))
      values) )
```

Appendix D
Functions to compute the median, using a sorting function

```
; Return the list of numbers in values that are less than
; or equal to x. x is a number, values is a list of numbers.
(define (smaller+equal-vals L x)
   (keep-if (lambda (y) (<= y x)) L) )

; Return the list of numbers in values that are larger
; than x. x is a number, values is a list of numbers.
(define (larger-vals L x)
   (keep-if (lambda (y) (> y x)) L) )

; L is a sorted list.
; Return the result of inserting x into L in proper order.
(define (insert L x)
   (append
      (smaller+equal-vals L x)
      (list x)
      (larger-vals L x) ) )

; Return the result of sorting the values in the list L.
(define (sort L)
   (accumulate
      insert
      (cons (list (first L)) (rest L)) ) )

; Return the median of the numbers in values,
; a list of numbers.
(define (median-by-sorting values)
   (list-ref (sort values) (half-length values)) )
```