Chapter 17

Iteration

Problem 17.1: Iterative versions of \texttt{insert} and \texttt{isort}

Recall the \texttt{insert} and \texttt{insertion-sort} functions defined in Section 16.5.1. First, define an iterative version of the \texttt{isort} function, called \texttt{insert-iter}.

- \textbf{Hint:} It may help to review the accumulator-based \texttt{isort-acc} function discussed in In-Class Problem 16.5.3.

- \textbf{Hint:} Use \texttt{let} to create a local variable \texttt{acc} that is initially empty. Then use a \texttt{while} loop to accumulate all of the numbers of the sorted list that are smaller than \texttt{item}. After the \texttt{while} loop, transfer all of the accumulated numbers back onto the sorted list... but with \texttt{item} in its proper place!

Next, define an iterative version of the \texttt{insertion-sort} function, called \texttt{isort-iter}, that uses \texttt{insert-iter} as a helper function.

- \textbf{Hint:} Use \texttt{let} to create a local variable called \texttt{sorted} that is initially empty, then use \texttt{dolist} to insert each element of the unsorted \texttt{listy} into the sorted \texttt{accumulator}.

Problem 17.2: Iterative versions of \texttt{merge} and \texttt{split}

Recall the \texttt{merge} and \texttt{split} functions described in Section 16.5.2. Define iterative versions of these functions, called \texttt{merge-iter} and \texttt{split-iter}, respectively.

- \textbf{Hint:} For \texttt{merge-iter}, use \texttt{let} to create a local variable called \texttt{acc}, then use a \texttt{while} loop to accumulate numbers from the two input lists, as long as both lists remain non-empty. After the \texttt{while} loop, transfer all of the accumulated numbers onto whichever of the two input lists is non-empty. (Why?)

- \textbf{Hint:} For \texttt{split-iter}, use \texttt{let} to create local variables called \texttt{lefty} and \texttt{righty}, that are both initially empty. Then use a \texttt{while} loop to process the numbers in the input list, \texttt{listy}. On each iteration, push one element of \texttt{listy} onto \texttt{lefty}, and another onto \texttt{righty}. Beware the case where \texttt{listy} only has one element!

\textbf{Note:} The Merge Sort algorithm is inherently recursive (why?); so we don’t attempt to implement an interactive version of it. However, you could define a version that uses \texttt{merge-iter} and \texttt{split-iter}. 
Problem 17.3: More fun with iteration

(A) Recall the is-elt-of? predicate from Example 16.2.3, which is similar to the built-in member function from and the index-of function from Example 16.2.4. Use a while loop to define an iterative version of the member function, called member-iter.

(B) Recall the fetch-nth-element function from In-Class Problem 16.2.3, which is equivalent to the built-in list-ref function. Use a while loop to define an iterative version, called list-ref-incr.

(C) Recall the print-n-dashes function from Example 14.2.1. Use dotimes to define an iterative version, called print-n-dashes-iter.

(D) Recall the print-rectangle function from Problem 14.6. Define an iterative version, called print-rectangle-iter, that does not use the print-dashes-iter function, but instead uses two dotimes loops, one nested inside the other. The outer loop should control the number of rows, the inner loop should control the number of dashes printed in each row.

(E) Recall the print-histy function from Example 16.2.6. Define an iterative version that uses dolist to walk through the given input list and, for each number in that list, uses dotimes to print out the desired number of asterisks.

(F) Recall the list-of-n-random-numbers function from In-Class Problem 16.5.2. Use dotimes to define an iterative version, called list-of-n-random-numbers-iter. (Use let to define a local variable called acc, initially empty.)

(G) Recall the function, num-occurs-in-n-tosses, from Problem 15.3. Define an iterative version, called num-occurs-in-n-tosses-iter, that uses a dotimes special form and an accumulator.

(H) Recall the function, toss-until-doubles, from Problem 15.7. Define an iterative version using a while loop.

Problem 17.4: Iteratively approximating logarithms

Recall the approx-log-acc and approx-log-wr functions from Problem 14.13.

(A) Define an iterative function, approx-log-iter, that takes the same inputs as approx-log-wr. It should use let to create local variables named from and acc, with appropriate initial values. Then it should use a while loop to iteratively accumulate terms in the sum, \( x - \frac{x^2}{2} + \frac{x^3}{3} - \ldots \pm \frac{x^n}{n} \). Alternatively, your function could use let to create only the acc local variable, leaving the management of from to a dotimes special form—instead of a while loop.

(B) Define a more efficient version of approx-log-iter, called approx-log-better. Instead of computing each power of \( x \) from scratch on each iteration, introduce an extra local variable called curr-power, whose value on successive iterations is \( x \), then \( x^2 \), then \( x^3 \), etc.

To see the difference in performance, you may wish to compare the results of evaluating the following expressions:

(time (approx-log-iter 1 10000))
(time (approx-log-better 1 10000))
Problem 17.5: Iteratively approximating $\pi$

Recall the approx-pi-acc and approx-pi-wr functions from Examples 14.3.5 and 14.4.2, respectively. Define an iterative function called approx-pi-iter that, like approx-pi-wr, takes a single input $n$. Your function should use let to create local variables called from, sign and acc, with suitable initial values. It should then use a while loop to accumulate the desired terms in the sum $4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} \ldots \pm \frac{4}{n}$.

Problem 17.6: Iteratively approximating $e$

(A) Recall the approx-e-acc and approx-e-wr functions from Examples 14.3.6 and 14.4.3, respectively. Define an iterative function called approx-e-iter that, like approx-e-wr, takes a single input $n$. Your function should use let to create local variables called indy, curr-denom and acc, with suitable initial values. It should then use a while loop to accumulate the desired terms in the sum, $1 + \frac{1}{2!} + \frac{1}{3!} + \ldots + \frac{1}{n!}$. Alternatively, your function could use let to create only curr-denom and acc, leaving the management of indy to a dotimes loop (instead of using while).

(B) More generally, mathematicians tell us that the function $e^x$ is very well approximated by sums of the form, $1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!}$. Define an iterative function called approx-e-x-iter, that computes such sums. It should be similar to approx-e-iter, except that:

- It takes an extra input $x$, that can be any real number; and
- It creates an extra local variable curr-power, whose values on successive iterations is 1, then $x$, then $x^2$, etc.
Chapter 18

Vectors

Problem 18.1

Define a (destructive) function that satisfies the following contract.

;;; DOUBLE-ALL!
;;; -----------------------------------------------------------------
;;; INPUT: VECKY, a vector of numbers
;;; OUTPUT: The same vector, modified as described below
;;; SIDE EFFECT: Doubles the contents of each slot (destructively)

> (define vecky (vector 10 20 30 40 50))
> vecky
#(10 20 30 40 50)
> (double-all! vecky)
#(20 40 60 80 100)
> vecky
#(20 40 60 80 100)

Problem 18.2

Define a (destructive) function, called roll-em!, that satisfies the following contract:

;;; ROLL-EM!
;;; -----------------------------------------------------------------
;;; INPUT: DICE, a vector of numbers
;;; OUTPUT: The same vector, but with contents destructively
;;; modified, as follows
;;; SIDE EFFECT: Replaces each slot with a random toss of a
;;; six-sided die

> (define dice (make-vector 5))
> (roll-em! dice)
#(3 2 5 3 4)
> dice
#(3 2 5 3 4)
> (roll-em! dice)
#(1 6 1 5 5)
> dice
#(1 6 1 5 5)

Hint: Use the toss-die function from the previous chapter.

Problem 18.3

Define a (destructive) function called roll-some! that satisfies the following contract:

```scheme
;; ROLL-SOME!
;; -------------------
;; INPUTS: DICE-VECK, a vector of dice values
;;         ROLLER, a vector of the same length as DICE-VECK,
;;         but consisting solely of 1s and 0s
;; OUTPUT: DICE-VECK, modified as described below
;; SIDE EFFECT: Walks through the two input vectors in parallel.
;; For each index I, if the Ith element of ROLLER is a 1, then
;; the Ith element of DICE-VECK is replaced by a random toss
;; of a 6-sided die; otherwise, it is unchanged.
```

Here are some examples:

```scheme
> my-dice
#(1 2 6 6 3 2 3)
> (roll-some! my-dice #(1 1 0 0 1 1 1))
#(5 4 2 6 6 3 4)
> (roll-some! my-dice #(1 1 0 0 0 1 1))
#(1 6 3 6 6 2 2)
> (roll-some! my-dice #(1 0 1 0 0 1 1))
#(4 6 5 6 6 1 6)
> my-dice
#(4 6 5 6 6 1 6)
```

Problem 18.4

Define a non-destructive function, called vector-reverse, that satisfies the following contract. Because it is non-destructive, it must create a new vector, instead of modifying the given vector.

```scheme
;; VECTOR-REVERSE
;; ----------------------------------------------------------------------
;; INPUT: VECK, a vector
;; OUTPUT: A new vector that is just like VECK, except
;;         that its elements are in the reverse order.
;; SIDE EFFECTS: none
```
Here are some examples:

```
> (vector-reverse #(a b c d))
#(d c b a)
> (vector-reverse #(1 2 3))
#(3 2 1)
```

Notice that the inputs in the above examples are immutable! So, if the function had tried to modify them, it would have caused an error!

Hints: Create a new vector of the appropriate length. Then use `dotimes` to walk thru the vector, setting its elements to appropriate values. Recall the `print-in-reverse` function from Example 18.4.3 for ideas. Don’t forget to return the new vector as output.

**Problem 18.5**

Define a (destructive) function that satisfies the following contract:

```
;; VECTOR-REVERSE!
;; ------------------------------
;; INPUT: VECKY, a vector
;; OUTPUT: The same vector, modified as described below
;; SIDE EFFECT: Destructively reverses the order of the
;; elements in VECKY.

> (vector-reverse! (vector 1 2 3 4 5))
#(5 4 3 2 1)
> (define vecky (vector 'a 'b 'c 'd))
> vecky
#(a b c d)
> (vector-reverse! vecky)
> vecky
#(d c b a)
```

**Problem 18.6**

Define a non-destructive function that satisfies the following contract:

```
;; VECTOR-MAP
;; ------------------------------
;; INPUTS: FUNK, a function that expects one input
;; VEKK, a vector of suitable inputs for FUNK
;; OUTPUT: A *new* vector of the same length as VEQUE
;; each of whose elements is obtained by applying FUNK to
;; the corresponding element of VEKK.
;; SIDE EFFECT: *NONE*
```
Here is an example that assumes that the `faky` function has already been defined.

```scheme
> (define vek #(1 2 3 4 5))  ← #(1 2 3 4 5) is an immutable vector
> (vector-map faky vek)
#(1 2 6 24 120)  ← New vector created by vector-map
> vek
#(1 2 3 4 5)  ← vek hasn’t changed
```

Notice that `vek` has been defined to be an immutable vector (i.e., its contents can’t be changed) and, thus, even if `vector-map` wanted to change its contents, it could not. (Attempting to do so would cause an error.) After `vector-map` is finished, `vek` remains the same (i.e., `vector-map` is non-destructive).

Hints: Use `make-vector` to create a new vector of the same length as `vek`. Then use `dotimes` to walk thru the new vector, setting each element to the result obtained by applying `funk` to the corresponding element of `vek`.

---

**Problem 18.7**

Define a destructive function that satisfies the following contract:

```scheme
;; VECTOR-MAP!
;; -----------------------------------------------
;; INPUTS: FUNK, a function that expects one input
;; VEQUE, a vector of suitable inputs for FUNK
;; OUTPUT: VEQUE, destructively modified...
;; SIDE EFFECT: Destructively modifies VEQUE by replacing
;; each of its elements by the result of applying FUNK to
;; the corresponding element of VEQUE.
```

Here is an example:

```scheme
> (define vec (vector 1 2 3 4 5))  ← VEC is a mutable vector
> (vector-map! faky vec)
#(1 2 6 24 120)  ← VECTOR-MAP! does its thing
> vec
#(1 2 6 24 120)  ← VEC has been changed!
```

Recall that the built-in `vector` function creates a mutable vector (i.e., one whose contents can be changed). `vector-map!` walks through the input vector, destructively modifying its contents. Afterward, `vec` is shown to be modified. Thus, `vector-map!` is destructive!

Hints: No need to create a new vector. Just use `dotimes` to walk through the given vector, destructively modifying its contents as you go.
Problem 18.8

Define a destructive function that satisfies the following contract: Define a function that satisfies the following contract:

;;; VECK-REPLACE!
;;; -----------------------------------
;;; INPUTS: OLD, anything
;;; NEW, anything
;;; VECKY, a vector
;;; OUTPUT: VECKY, destructively modified as follows.
;;; SIDE EFFECT: Replaces each occurrence of OLD in VECKY
;;; by NEW, where equality is as judged by EQ?.

Here is an example:

> (define vequi (vector 1 2 1 3 1 4)) ← VEQUI is mutable
> vequi
#(1 2 1 3 1 4)
> (veck-replace! 1 111 vequi)
#(111 2 111 3 111 4)
> vequi
#(111 2 111 3 111 4) ← VEQUI has changed!

Problem 18.9

Define a non-destructive function, called veck-index-of, that satisfies the following contract:

;;; VECK-INDEX-OF
;;; -----------------------------------
;;; INPUTS: ITEM, anything
;;; VECKY, a vector
;;; OUTPUT: The index of the first slot of VECKY that contains
;;; an first occurrence of ITEM; or #f if ITEM does
;;; not appear in VECKY.

Here are some examples of its use:

> (veck-index-of 'x #(a b c x y z x x))
3
> (veck-index-of 'z #(a b c x y z x x))
5
> (veck-index-of 'w #(a b c x y z x x))
#f

Hint: Define a helper function, called veck-index-of-helper, that includes an extra input I, that serves as an index into the given vector. Make recursive function calls, incrementing I, until you find an occurrence of ITEM, or I gets too big. Here’s the contract for the helper function:
Problem 18.10: Converting a vector into a list

The goal of this problem is to define a function, called veck-to-list, that takes a vector as its only input, and returns as its output a list containing the same elements, in the same order, as illustrated below.

> (veck-to-list #(a b c d))
(a b c d)
> (veck-to-list (make-vector 5))
(0 0 0 0)
> (veck-to-list (vector 1 #t ()))
(1 #t ())

Here is the contract for veck-to-list:

;;; VECK-TO-LIST
;;; -------------------------------------
;;; INPUT: VECK, a vector
;;; OUTPUT: A list having the same elements as VECK, and in the same order.

Given the tools that we have seen so far, for this problem, it is probably easiest to define a recursive helper function, called veck-to-list-helper, that includes an extra input I that identifies the current element of the vector. Here is the contract:

;;; VECK-TO-LIST-HELPER
;;; -------------------------------------
;;; INPUTS: VECK, a vector
;;; I, an index
;;; OUTPUT: A list containing the elements of VECK from the index I onward

Here are some examples:

> (veck-to-list-helper #(a b c d e) 2)
(c d e)
> (veck-to-list-helper #(a b c d e) 3)
(d e)
Hints:

* When the index I reaches the length of the vector, then it is no longer a valid index for that vector, indicating that you have reached the base case of the recursion.

* Remember, you are returning a list in the base case, and a list in the recursive case!

* For this problem, it is probably easier not to use an accumulator.

Incidentally, after you have implemented this function, you can compare it to the built-in function, vector->list, that does the same thing!

Problem 18.11: Converting a list into a vector

The goal of this problem is to define a function, called list-to-veck, that takes a list as its only input, and returns as its output a vector containing the same elements, in the same order, as illustrated below.

> (list-to-veck '(a b c d))
(a b c d)
> (list-to-veck (list 1 #t ()))
(#(1 #t ()))

Here is the contract for list-to-veck:

;; LIST-TO-VECK
;; ---------------------------------  
;; INPUT: LISTY, a list
;; OUTPUT: A vector containing the same elements as LISTY, and in the same order
;; SIDE EFFECTS: none

Hints:

* You can call the built-in length function once to find out how long the list is—but don’t call it more than once! Use it to create a vector of the same length. Then, you might think about using dotimes to walk thru the vector, but that would be inefficient if you are planning to use the fetch-nth-element or list-ref functions (cf. In-class problem 16.2.3) to access the successive elements of the list. Instead, define a recursive helper function that satisfies the following contract. It will use list-based recursion to walk thru the list. On each recursive function call, the first element of the list will be the one you want to give to the vector-set! function. This approach will be much more efficient.

;; LIST-TO-VECK-HELPER
;; -----------   
;; INPUTS: LISTY, a list
;; VECK, a vector
;; I, an index (non-negative integer)
;; OUTPUT: The vector, VECK, modified as described below
;; SIDE EFFECTS: When called with I=0, copies the contents
Problem 18.12

Define a non-destructive function, called every-other-one-vector, that satisfies the following contract:

```scheme
;; EVERY-OTHER-ONE-VECTOR
;; --------------------------------------------------------------
;; INPUT: VECKY, a vector
;; OUTPUT: A vector containing every other element of VECKY.
;; Note: The output vector should contain roughly half the
;; elements of VECKY.
```

Here are some examples of its behavior:

```scheme
> (every-other-one #(0 1 2 3 4 5 6))
#(0 2 4 6)
> (every-other-one #(1 2 3 4 5 6))
#(1 3 5)
```

Hints:

- Create a new vector whose length is roughly half that of the input vector. Then use `do_times` to walk through that new vector, copying relevant elements from the input vector to the new vector.
- You may find the built-in `even?`, `odd?`, or `quotient` functions helpful. (`even?` and `odd?` were introduced in Problem 14.2; `quotient` was introduced in Section 5.3.)
- You should not use lists for this function; use vectors!

Problem 18.13: Computing a histogram

Define a function, called compute-histogram, that satisfies the following contract:
;; COMPUTE-HISTOGRAM
;; ---------------------------------------------------------------
;; INPUT: VECK-O-DICE, a vector of dice values (each 1 thru 6)
;; OUTPUT: A vector of length 7, where the ith slot contains
;; the number of occurrences of i in VECK-O-DICE.
;; (The 0th slot of the output vector is ignored.)

Here’s an example:

```scheme
> (define my-dice #(1 2 1 2 6 6 6 5 2))
> (compute-histogram my-dice)
#(0 2 3 0 0 1 3)
```

In this case, my-dice contains:

- two 1s
- three 2s
- no 3s
- no 4s
- one 5
- three 6s

These counts are reflected in the histogram computed by this function. Note that the histogram is a vector with seven slots, numbered 0 thru 6. The zeroeth slot is ignored. We only care about slots 1 thru 6. For each i > 0, the slot of the output vector at index i contains the number of occurrences of i in my-dice (or veck-o-dice). Here’s another example:

```scheme
> (define my-dice #(3 3 3 3 3 1 1 1))
> (compute-histogram my-dice)
#(0 4 0 5 0 0 0)
```

Hint: Create a vector of length seven called histy, then use dotimes to walk through the veck-o-dice (not histy). For each index i, look at the ith slot of veck-o-dice and use its value to figure out which slot of histy to increment. In effect, histy is a vector with seven accumulators, one of which we are ignoring.

Problem 18.14

* You may wish to review Problem 16.6 before starting this problem.

Define a function, called veck-has-satisfier?, that satisfies the following contract:

```scheme
;; VECK-HAS-SATISIFIER?
;; ---------------------------------------------------------------
;; INPUTS: FUNK, a predicate that expects one input
;; VECK, a vector of suitable inputs for FUNK
;; OUTPUT: #t, if VECK contains an element that satisfies
;; FUNK (i.e., that makes FUNK return #t)
;; #f, otherwise.
```
Here are some examples:

> (veck-has-satisfier? number? #(a #t 3 x))
#t
> (veck-has-satisfier? symbol? #(1 2 3 4))
#f

Note that it would be inefficient to implement this function using dotimes because dotimes invariably walks through the entire vector. This function should stop as soon as it finds an element of veck that satisfies funk. Therefore, you should define a recursive helper function that takes an extra input, i, that serves as an index into the vector veck.

**Problem 18.15**

**Define a function, called has-three-of-a-kind?, that satisfies the following contract:**

```
;; HAS-THREE-OF-A-KIND?
;; -----------------------------------------------
;; INPUT: VECK-O-DICE
;; OUTPUT: #t if the vector of dice contains *at least*
;;         three of one kind; #f otherwise.
```

Here are some examples:

> (has-three-of-a-kind? #(1 2 1 2 1)) ← has three ones
#t
> (has-three-of-a-kind? #(4 2 4 4 4)) ← has four fours
#t
> (has-three-of-a-kind? #(6 6 6 6 6)) ← has five sixes
#t
> (has-three-of-a-kind? #(6 5 6 5 2)) ← does not have three of a kind
#f

Notice that having four or five of a kind also counts as having three of a kind.

**Hint:** One way to solve this problem: Compute a histogram vector, then check whether it has an element that is 3 or bigger. Can you think of a way to use veck-has-satisfier? from Problem 18.14 to check whether the histogram vector contains an element that is 3 or bigger?

**Problem 18.16**

**Define a function, called has-large-straight?, that satisfies the following contract:**

```
;; HAS-LARGE-STRAIGHT?
;; -----------------------------------------------
;; INPUT: VECK-O-DICE, a vector of five dice values
;; OUTPUT: #t if the vector of dice contains all of the
```
Here are some examples:

```scheme
> (has-large-straight? #(2 4 5 3 1))
#t
> (has-large-straight? #(6 5 4 2 3))
#t
> (has-large-straight? #(6 4 1 2 3))
#f
```

Hints: You may assume that the input vector has exactly five slots. Compute a histogram and go from there! What must the histogram look like for a large straight? (Use the built-in `equal?` predicate (cf. In-class problem 18.4.2) to make your life easier!) Alternatively, convert the vector of dice into a list, then use the built-in `sort` function (cf. Example 16.5.6) to sort its contents into non-decreasing order. What must it look like at that point?