

CHAPTER TWELVE

Dynamic Programming

12.1 Introduction

In the previous chapter, we introduced the notion of storage, embodied in Scheme by vectors. The point of storage is to allow one part of a computation to store a value that a later part of the computation will retrieve. However, this leaves open the question of what ultimate objectives can be served by this “squirreling away” of information.

In this chapter, we’ll introduce one important use for storage, a technique that can be used to make some computations dramatically more efficient. In particular, we’ll look at tree-recursive processes that naturally wind up repeatedly recomputing the solution to identical subproblems. By using storage, we can keep track of which computations have already been done and what the results were so that we can reuse the results rather than recompute them. We’ll look at two variants on this theme, one in which results are stashed away on an “opportunistic” basis as they need to be computed and another in which we systematically precompute results before we have any need for them. Although we’ll look at the differences between these two variants, we’ll also emphasize their commonality. Both have the power to turn a computation that takes too long to be feasible into one that completes in an instant.

We’ll start with an overview section, in which we take a single simple example problem through all three different versions: the original tree-recursive form, the version that opportunistically stores results (memoization), and the version that precomputes results (dynamic programming). We’ll then have a section apiece devoted to memoization and dynamic programming, in which we work through more examples, addressing some side issues that don’t arise in our simple introductory example. A short section brings the two variants back together in order to compare their relative advantages. Finally, you will apply these techniques to the problem of breaking paragraphs into lines in a visually pleasing way.

12.2 Revisiting Tree Recursion

During summer vacation this year, two of us decided to add a path from our back door to the garden. We had two different lengths of paving stones available to us; one was 1 foot long and the other was 2 feet. (Both were wide enough for the kind of path we had in mind.) We were willing to use any combination of the two sizes, provided that the total length worked out correctly. When we tried to decide the exact pattern of stones, however, we found that we could not agree on which pattern to follow. At that point, the third author suggested that we use Scheme to generate all the possible patterns, present them to him, and he would pick the most pleasing one. For example, if the path was only 4 feet long, there would be five patterns that look like these:



As we headed indoors to try this idea out, one of us had second thoughts and became concerned about the number of pictures we might wind up generating. We decided to first write a procedure that would count the number of different ways to make a path using these pavers.

To write this procedure, we can concentrate on the length of the path. If the length is 1 foot, there is only one way to form it, because we must use one of the pavers that are 1 foot long. If the length is 0 feet, there is only one way to form it, by using no pavers at all. On the other hand, if the length is 2 or more feet, we have two choices for the first paver. If we use a 1-foot paver as the first paver, the number of ways to pave the remainder of the walk is the same as the number of ways to construct a walk that is 1 foot shorter. Similarly, if the first stone is a 2-foot paver, the number of ways to complete the path is the number of ways to construct a path that is 2 feet shorter. Thus, the total number of ways (starting with either size paver) to construct a path that is a given length will be the sum of the number of ways to construct a path that is 1 foot shorter and the number of ways to construct a walk 2 feet shorter:

```
(define walk-count
  (lambda (feet)
    (cond ((= feet 0) 1)
          ((= feet 1) 1)
          (else (+ (walk-count (- feet 1))
                   (walk-count (- feet 2)))))))
```

We tested this out on some relatively small numbers, and it seemed to work. For example, it correctly told us that there are five ways to pave a 4-foot walk, as shown by the preceding pictures. So, we moved to our real question, which was how many ways there were to pave the 30-foot path in the back yard. After waiting over a minute

without getting an answer, we got impatient and gave up. Interestingly, the answer was nowhere near as slow in arriving for only slightly shorter paths; for example, we were able to find in under a second that there are 10,946 different ways to pave a 20-foot path. Later, when we were feeling more patient, we timed the procedure for all lengths from 20 to 30 feet and constructed the following table:

Path length	Ways to pave	Seconds to find
20	10,946	0.83
21	17,711	1.32
22	28,657	2.14
23	46,368	3.49
24	75,025	5.61
25	121,393	9.08
26	196,418	14.51
27	317,811	23.56
28	514,229	37.87
29	832,040	61.28
30	1,346,269	99.67

As you can see from the table, although our tree-recursive procedure is very elegant, it also takes an enormous amount of time except for very small path lengths. Remembering the asymptotic outlook from Chapter 4, what is happening is that the time is growing quickly as the path length increases. You may notice that the number of ways to pave the path is also growing similarly quickly. If you compare adjacent rows of the table, you'll see that neither the number of ways to pave nor the number of seconds taken is growing so fast as to double with each additional foot of path. On the other hand, if you compare a row with the one two rows down from it, you'll see that both the number of pavings and the number of seconds are growing quickly enough that they more than double with each additional 2 feet of path. So, if we use n to denote the path length in feet, the empirical evidence seems to suggest that both the number of pavings and the time taken have order of growth $\Theta(b^n)$ where the base of the exponent, b is somewhat more than $\sqrt{2}$ but less than 2. (*Exponential growth* with a base of 2 would double each foot, whereas with a base of $\sqrt{2}$, it would double every 2 feet.) It turns out that it isn't hard to show that the number of pavings and the time grow faster than $(\sqrt{2})^n$ and slower than 2^n . We'll take a minute to do the necessary math and then move on to our real goal, which is to show how to use memory in the form of vectors to dramatically reduce the order of growth of tree-recursive processes like this one.

Before doing the mathematical analysis, we'll give a more compact name to the number of ways to pave an n -foot path using any mixture of 1- and 2-foot pavers.

Traditionally mathematicians use the symbol F_{n+1} for this number. That is, F_1 is the number of ways to pave a 0-foot walk, F_2 is the number of ways to pave a 1-foot walk, etc. The letter F stands for Fibonacci, because these numbers are known as the *Fibonacci numbers*. These numbers were originally popularized by a problem concerning rabbit reproduction in an early thirteenth-century arithmetic book whose author, Leonardo Pisano, was sometimes known by the patronymic Fibonacci.

At any rate, our definition is that $F_1 = 1$, $F_2 = 1$, and for any $n > 2$, $F_n = F_{n-1} + F_{n-2}$. You can convince yourself that after the first two Fibonacci numbers, they get steadily larger, that is, $F_n > F_{n-1}$ for all $n > 2$. (All the Fibonacci numbers are positive because adding positives yields a positive; thus, because F_n is the sum of F_{n-1} and the positive number F_{n-2} , it must be larger than F_{n-1} .) We can get some handle on how fast the Fibonacci numbers grow by observing that $F_n = F_{n-1} + F_{n-2}$, and because the numbers are increasing in size, we have $F_{n-1} > F_{n-2}$ (for $n > 3$). Therefore, F_n must be larger than twice F_{n-2} but not as large as twice F_{n-1} (again, for $n > 2$). Thus we have shown that the number of ways to pave a path more than doubles with every 2 feet in length but doesn't grow quickly enough to double every foot, in keeping with the empirical evidence.

All that remains is to show that the time our tree-recursive procedure takes to compute F_{n+1} grows at the same order of growth as F_{n+1} itself. If you look at the procedure, you can see that all it does is add up lots of 1s to get its answer. The base case invocations supply the 1s, and the remaining procedure invocations add them up. To get a total answer of F_{n+1} , there must be F_{n+1} 1s being added together, so the recursion tree must have F_{n+1} base-case leaves. Imagine starting with those F_{n+1} 1s in a pile and repeatedly taking two numbers out of the pile, adding them together, and putting the sum back into the pile. Continue until there is only one number left in the pile. How many additions did you do? Well, each addition reduced the number of numbers by one because you took two numbers out of the pile and put one back in. You started with F_{n+1} numbers (the 1s) and ended with one number (F_{n+1} itself). Thus, the number of numbers went down by $F_{n+1} - 1$, and that must be the number of additions. This tells us how many invocations of the `walk-count` procedure there are when you start by doing (`walk-count n`). The answer is F_{n+1} base-case invocations plus $F_{n+1} - 1$ adding-up invocations, for a grand total of $2F_{n+1} - 1$. Thus, the amount of work done has the same order of growth as F_{n+1} does, namely, a quickly growing exponential growth.

From the foregoing analysis, and our earlier table of ways to pave paths of length 20 through 30 feet, we can see that evaluating (`walk-count 30`) winds up involving over 2.6 million invocations of `walk-count`. On the other hand, there isn't much variety in these invocations. Because the `walk-count` for a particular path length is computed only from the counts for smaller path lengths, there are at most 31 different `walk-counts` that can be involved in the computation of (`walk-count 30`): (`walk-count 0`) up through (`walk-count 30`). Since we have over 2.6 million invocations of `walk-count`, but at most 31 different ones, there clearly must be a

great deal of repetition. This is our sign that there is a substantial opportunity for improvement. This is a general sign, worth being on the lookout for in the future. Therefore, we'll give it a name:

The much computation, little variety sign: If a procedure does a great number of subcomputations, but there isn't much room for variety in the subcomputations that can arise, a lot of repeated computations must occur. That is a sign that a large performance improvement may be possible.

With this sign to motivate us, we'll turn to how we can use memory to get a more reasonable order of growth. We will look at two techniques, memoization and dynamic programming. We don't actually need either of these techniques to write an efficient version of the `walk-count` procedure—we could instead just write an iterative version. However, we're going to need memoization and dynamic programming in our tool kit when we encounter harder problems. Rather than waiting for those problems, we'll learn the essential ideas of memoization and dynamic programming in the comparatively simple context of the `walk-count` procedure. Moreover, the iterative version we're choosing not to write isn't fundamentally very different from the dynamic programming version.

The first technique we look at is called *memoization*. The basic idea behind memoization is to use memory to store a table of the values that we've already calculated. Then we can take advantage of the fact that once we've calculated a particular Fibonacci number, we will never need to recalculate it.

With memoization, we need to make a slight change in how we calculate Fibonacci numbers. Whenever we need to know a smaller Fibonacci number, as a subproblem, we don't want to just blindly go ahead and calculate it. Instead, we want to first check to see if it is already in our table of values; if not, we compute it and put it in the table. To accommodate this change, we'll slightly modify `walk-count` so that rather than directly invoking itself for the recursive calls, it invokes a separate `walk-count-subproblem` procedure:

```
(define walk-count
  (lambda (n)
    (cond ((= n 0) 1)
          ((= n 1) 1)
          (else (+ (walk-count-subproblem (- n 1))
                   (walk-count-subproblem (- n 2)))))))
```

Before we define `walk-count-subproblem`, and see how it stores the subproblem values in a table for reuse, let's address the question of where the table comes from. What we can do is define a `memoized-walk-count` procedure that creates

the table (as an n -element vector), then has internal definitions of our modified `walk-count`, `walk-count-subproblem`, and other helping procedures, and finally calls the modified `walk-count` to do the actual computation. That is, the overall template looks like this:

```
(define memoized-walk-count
  (lambda (n)
    (let ((table (make-vector n)))
      (define walk-count
        (lambda (n)
          (cond ((= n 0) 1)
                ((= n 1) 1)
                (else (+ (walk-count-subproblem (- n 1))
                        (walk-count-subproblem (- n 2)))))))
      (define walk-count-subproblem
        (lambda (n)
          we still need to define this)
          there will be some other helping stuff here
          (walk-count n))))))
```

Of course, we could have named the outer `memoized-walk-count` procedure `walk-count` as well, which would have the advantage that any callers of the old, slow, tree-recursive `walk-count` procedure would automatically now be calling the sleek, new, memoized version. We've chosen to give it a distinctive name so that we can more easily compare the two versions.

We can write `walk-count-subproblem` as a two-step process: first ensure that the value is in the table (i.e., put it there if it isn't already), and then in any case return the value found in the table:

```
(define walk-count-subproblem
  (lambda (n)
    (ensure-in-table! n)
    (vector-ref table n)))
```

To ensure that a value is in the table, we will first check to see if the table has a value already stored in the position n ; if not, we store one there. This leaves the question: How do we check? We'll take the approach of initially storing the false value, `#f`, into all the table entries. That way when an actual value is stored into one of the entries, it will change from being false to being something nonfalse. Because Scheme interprets anything other than false as being true, we can write the procedure as follows:

```
(define ensure-in-table!
  (lambda (n)
    (if (vector-ref table n) ; anything but #f ?
        'done
        (store-into-table! n))))
```

The `store-into-table!` procedure itself is trivial:

```
(define store-into-table!
  (lambda (n)
    (vector-set! table n (walk-count n))))
```

The only missing step is initially filling the whole table with `#f` values. That can be done using a procedure called `vector-fill!`, which we would use as follows:

```
(vector-fill! table #f)
```

The procedure `vector-fill!` is very similar to the procedure `zero-out-vector!` in Section 11.6 in that it puts a particular value in each location of a vector. This procedure is specified by the R⁴RS Scheme standard as being “inessential” (i.e., not every Scheme implementation is required to provide it).

Exercise 12.1

Write the procedure `vector-fill!`

At this point, we’ve seen all the pieces of `memoized-walk-count` and merely need to put them together in one place:

```
(define memoized-walk-count
  (lambda (n)
    (let ((table (make-vector n)))
      (define walk-count
        (lambda (n)
          (cond ((= n 0) 1)
                ((= n 1) 1)
                (else (+ (walk-count-subproblem (- n 1))
                        (walk-count-subproblem (- n 2)))))))
      (define walk-count-subproblem
        (lambda (n)
          (ensure-in-table! n)
          (vector-ref table n))))))
```

```

(define ensure-in-table!
  (lambda (n)
    (if (vector-ref table n)
        'done
        (store-into-table! n))))
(define store-into-table!
  (lambda (n)
    (vector-set! table n (walk-count n))))
(vector-fill! table #f)
(walk-count n)))

```

Now let's take a careful look at how the table gets filled in. As soon as a value is calculated for the first time, it is placed in the table. On the other hand, we can't compute a value until we have computed the previous two values. Thus, the very first values that go in the table are F_1 and F_2 , then F_3 , F_4 , and so on. This suggests an iterative way of computing Fibonacci numbers, using a vector. The basic idea is to construct a vector of values as we did in the memoized approach, but instead of filling this vector in as needed, we start at the beginning (at index 0) and fill in the whole vector systematically, start to finish. This second approach to using memory to improve the time complexity of a procedure is called *dynamic programming*; we'll abbreviate dynamic programming to *dp* and call this version of the procedure *dp-walk-count*:

```

(define dp-walk-count
  (lambda (n)
    (let ((table (make-vector n)))
      (define walk-count
        (lambda (n)
          (cond ((= n 0) 1)
                ((= n 1) 1)
                (else (+ (walk-count-subproblem (- n 1))
                        (walk-count-subproblem (- n 2)))))))
      (define walk-count-subproblem
        (lambda (n)
          ;; no need to ensure in table, given systematic filling in
          (vector-ref table n)))
      (define store-into-table!
        (lambda (n)
          (vector-set! table n (walk-count n))))
      (define store-into-table-from!
        ;; does store-into-table! for values from start through n-1

```



```

(define walk-count-subproblem
  (lambda (n)
    ;; no need to ensure in table
    (vector-ref table n)))
(define store-into-table!
  (lambda (n)
    (vector-set! table n (walk-count n))))
(from-to-do 0 (- n 1) store-into-table!)
(walk-count n)))

```

The most important point to make is that both the memoized and the dynamic programming versions are *huge* improvements over the original tree recursive version. You may recall that the original version took times ranging from 0.83 seconds to compute F_{21} up to 99.67 seconds to compute F_{31} on our computer. By contrast, the memoized and dynamic programming versions ranged from 0.002 or 0.003 seconds for F_{21} only up to 0.003 or 0.004 seconds for F_{31} .

Because the two techniques are so similar and lead to such similarly dramatic performance improvements, you may wonder why we've bothered to present both. Each technique turns out to have its own advantages; however, they don't show up very clearly in this simple example. Therefore, we'll wait until we've seen more examples of memoization and dynamic programming (in the next two sections) before presenting a comparison of the relative strengths of these two techniques. The key point to remember, though, is that the differences between memoization and dynamic programming are nowhere near as dramatic as those between either one of them and tree recursion.

Although we probably could have figured out the dynamic programming version of `walk-count` without writing the memoized version first, we will find that with more complicated problems, doing a memoized version helps us visualize the table without having to worry about how it gets filled in. In the rest of this chapter, we will look at several problems, first concentrating on how to make a memoized version and then looking at how to write dynamic programming solutions.

12.3 Memoization

In this section, we consider the *binomial coefficients* that are calculated by the solution to Exercise 4.17 on page 103. These numbers describe the number of ways to select a subset of k objects from a set of n distinct objects, for values of k and n such that $0 \leq k \leq n$. To choose k of n items, we can either choose the first item and $k - 1$ of the remaining $n - 1$ or not choose the first item and choose k of the remaining $n - 1$. This provides the recursive case of our procedure:

```
(define choose
  (lambda (n k)
    (cond ((= n k) 1)
          ((= k 0) 1)
          (else (+ (choose (- n 1) (- k 1))
                   (choose (- n 1) k))))))
```

We can measure the time complexity of `choose` by counting the number of additions that are needed to compute `(choose n k)`. Because the base cases only contribute a value of 1, we can use the same argument as we did with `walk-count` to show that the number of additions needed to compute `(choose n k)` must be one less than its actual value. However, finding bounds on the size of $C(n, k)$ is very complicated (and beyond the scope of this text). Even if n grows large, $C(n, k)$ won't grow rapidly larger if k stays small. In particular, $C(n, 0) = 1$ and $C(n, 1) = n$, neither of which is rapidly growing. Similarly, if k stays close to n as n grows, $C(n, k)$ again won't grow rapidly; as examples, consider that $C(n, n) = 1$ and $C(n, n - 1) = n$. The most rapidly growing case is when k is midway in between these extremes, that is, when we're looking at $C(2k, k)$ as k grows large. This case grows very large very fast, as shown in the following table:

k	$C(2k, k)$
1	2
2	6
3	20
4	70
5	252
6	924
7	3,432
8	12,870
9	48,620
10	184,756
11	705,432
12	2,704,156
13	10,400,600
14	40,116,600
15	155,117,520
16	601,080,390
17	2,333,606,220
18	9,075,135,300
19	35,345,263,800
20	137,846,528,820

Thus, it is clear that our tree-recursive `choose` procedure, which adds up 1s one at a time, is going to take an unreasonable amount of time even to solve such a small problem as $C(40, 20)$.

▶ Exercise 12.4

Draw a tree, analogous to Figure 4.4 on page 92, showing how the value of $C(4, 2)$ is computed by `choose`. Your tree should have $C(4, 2)$ at the root and six 1s as its leaves.

▶ Exercise 12.5

Explain how the “much computation, little variety sign” applies to this problem. In particular, we showed that $C(40, 20) = 137,846,528,820$, which the tree-recursive `choose` procedure computes by adding up 137,846,528,820 1s, clearly a great deal of computation. What about the variety side of the picture? In the course of computing $C(40, 20)$, the tree-recursive `choose` procedure winds up computing $C(n, k)$ for other values of n and k . Can you say anything about how many different combinations of n and k might arise?

We will first improve the time complexity of `choose` by using memoization, just as we did with the Fibonacci numbers. In the next section, we will look at a dynamic programming solution to computing `choose`.

There is one major difference between `choose` and `walk-count`. Whereas `walk-count` has only one parameter and thus can easily use a vector to store the calculated values, `choose` has two parameters. If we think of the `walk-count` value vector as being a table, we see that we were using a one-dimensional table there. For `choose` we’ll need a two-dimensional table, or a grid. Such a table would have two sets of indices, one for the rows and one for the columns. Each element in the table can be located by two numbers—one that identifies which row the element is in and one that identifies which column it’s in. A typical picture would then be something like the following

	0	1	2	3	4	5
0						
1						
2					6	
3						

In this example, the element in row 2 and column 4 is 6, and the height of the table (number of rows) is 4, whereas the width (number of columns) is 6. Note that the rows and columns are numbered starting from 0.

Unfortunately, Scheme does not provide two-dimensional tables for us. We can define an abstract data type for them; we'll need to have a procedure that constructs tables, a procedure that looks up a value in a table, and a procedure that changes the value of a given location in a table. We could also have selectors that tell us how many rows or columns a particular table has. Assume that these procedures are specified by the following definitions:

```
(define make-table
  (lambda (number-of-rows number-of-columns)
    ...))

(define table-ref
  (lambda (table row column)
    ...))

(define table-set!
  (lambda (table row column value)
    ...))

(define table-height
  (lambda (table)
    ...))

(define table-width
  (lambda (table)
    ...))
```

Exercise 12.6

Using these procedures, write

- a. A procedure called `table-fill!` that takes a table and an element and sets every entry in the table to the given element. For example, `(table-fill! table 0)` would have a similar effect to that of `zero-out-vector!` in Section 11.6.
- b. A procedure called `display-table` that nicely displays its table parameter.

How do we implement tables? We want to use vectors because they allow us to store results. But somehow we need to create a two-dimensional table out of a one-dimensional vector. One way to do this is to think of a table as a sequence of rows (i.e., a vector of rows). Each row is then divided up into a sequence of elements, one per column; in other words, each row is itself a vector. When we want the element

in row r and column c , we look at the vector in position r of our table and find the element in position c of that vector. Thus, the procedure `table-ref` is defined by

```
(define table-ref
  (lambda (table row col)
    (vector-ref (vector-ref table row) col)))
```

▶ Exercise 12.7

Write the procedure `table-set!`!

▶ Exercise 12.8

Write the procedures `table-height` and `table-width`. Each of these should take a table as a parameter and return the number of rows or columns in that table.

Creating the table is fairly straightforward. We want to first create a vector for the rows. Then we want to fill this vector with vectors:

```
(define make-table
  (lambda (r c)
    (let ((table (make-vector r)))
      (from-to-do 0 (- r 1)
        (lambda (i)
          (vector-set! table i (make-vector c))))
      table)))
```

Now that we have tables, we can make a memoized version of `choose`, which is very similar to making a memoized version of `walk-count`. As before, we will construct a table, although this one is a two-dimensional table instead of a one-dimensional one. We will initially set all of the entries of this table to false. Before using any element of the table, we ensure that it has been filled in:

```
(define memoized-choose
  (lambda (n k)
    (let ((table (make-table n (+ k 1))))
      (define choose
        (lambda (n k)
          (cond ((= n k) 1)
                ((= k 0) 1)
                (else (table-ref table n k)))))
      table)))
```

```

      (else (+ (choose-subproblem (- n 1) (- k 1))
              (choose-subproblem (- n 1) k))))))
(define choose-subproblem
  (lambda (n k)
    (ensure-in-table! n k)
    (table-ref table n k)))
(define ensure-in-table!
  (lambda (n k)
    (if (table-ref table n k)
        'done
        (store-into-table! n k))))
(define store-into-table!
  (lambda (n k)
    (table-set! table n k (choose n k))))
(table-fill! table #f)
(choose n k)))

```

As you can see, the relationship between `memoized-choose` and `choose` is essentially identical to the relationship between `memoized-walk-count` and `walk-count`, except `make-table`, `table-ref`, `table-set!`, and `table-fill!` are used in place of `make-vector`, `vector-ref`, `vector-set!`, and `vector-fill!`.

One subtle point is the size chosen for the table: n rows and $k + 1$ columns. The n rows are enough because the first argument to `choose-subproblem` will never be any larger than $n - 1$. (Remember, with n rows, the indices can run from 0 to $n - 1$.) By contrast, the second argument to `choose-subproblem` can be as large as k . Therefore, $k + 1$ columns are needed. That way the legal range for column indices includes k .

One of the nice features about the memoized version of `choose` is the relationship between the indices of the table entries and the values of those entries. More precisely, `(choose i j)` has the same value as `(table-ref table i j)`, assuming that the table has been filled in at that position. In the next example, we consider a situation where this relationship is not as direct.

This second example is a chocolate version of a famous problem in computer science. Suppose we are at a chocolate candy store and want to assemble a kilogram box of chocolates. Some of the chocolates (such as the caramels) at this store are absolutely the best in the world, and others are only so-so. In fact, we've rated each one on a scale of 1 to 10, with 10 being the highest. Furthermore, we know the weight of a piece of each kind; for example, a caramel weighs 13 grams. How do we put together the best box weighing at most 1 kilogram? (The more well-known version of this problem is known as the *knapsack problem*, but we have trouble imagining packing chocolates into a knapsack rather than a box and trouble imagining anything other than chocolate being of sufficient value to warrant optimization.)

Before we start writing a Scheme procedure to solve this problem, we need to construct abstract data types for chocolates and boxes of chocolates and to define what we mean by “the best” box.

Defining chocolates is quite easy. At the candy store, each chocolate has a filling (e.g., caramel, marshmallow, maple cream) and a coating of dark, milk, or white chocolate. We also know the weight of an individual piece of chocolate as well as a number that describes its desirability. We will glue these four attributes together in a list and use `car` and `cdrs` for selectors:

```
(define make-chocolate
  (lambda (filling covering weight desirability)
    (list filling covering weight desirability)))

(define chocolate-filling car)
(define chocolate-covering cadr)
(define chocolate-weight caddr)
(define chocolate-desirability caddr)
```

Boxes of chocolates are just collections of pieces of chocolate. Some boxes are empty (indeed, in one author’s office, all the boxes are empty); some contain several pieces. The weight of a box of chocolates is the sum of the weights of the pieces. Similarly, the desirability of a box is the sum of the desirabilities of the pieces. The best box, then, is the one with a maximum desirability.

As an abstract data type, we will want to know what chocolates are in the box, what the weight of a box is, and what the desirability of a box is. We will also need to compare two boxes to see which one is better. For constructors, we will need to be able to make an empty box, and we will also need to be able to add a chocolate to a box. Because we’re concentrating on buying the box, we won’t worry about taking chocolates out of the box.

Initially, we might be tempted to use lists of chocolates to implement our box ADT. To find the weight of a box, we would just `cdr` down the list that represents it and add up the weights of the individual pieces, and we would find the desirability in a similar fashion. However, this approach can be time-consuming if we do a lot of `box-weight` or `box-desirability` operations. Because we want to use this ADT in a procedure that finds the most desirable box subject to a given weight limit, we are likely to be doing exactly that.

We can improve on using just lists to represent boxes by using a combination of a list of chocolates, a weight, and a desirability. We will need to be sure that the weight of a box is actually equal to the sum of the weights of the chocolates in the list and, similarly, that the desirability is the sum of the desirabilities of the chocolates. Therefore, whenever we add a piece of chocolate to a box, we will want to `cons` it onto the list, add its weight to the weight and add its desirability to the desirability. Furthermore an empty box will have a weight of 0 and a desirability of 0:


```

(define make-empty-box
  (lambda ()
    (list '() 0 0)))

(define box-chocolates car)
(define box-weight cadr)
(define box-desirability caddr)

(define add-chocolate-to-box
  (lambda (choc box)
    (list (cons choc
                (box-chocolates box))
          (+ (chocolate-weight choc)
              (box-weight box))
          (+ (chocolate-desirability choc)
              (box-desirability box))))))

```

▶ Exercise 12.9

Using these procedures, write a procedure called `better-box` that takes two boxes of chocolates and returns the one that is more desirable. If they are equally desirable, you should return whichever one you choose.

How do we find the most desirable box weighing 1 kilogram or less? As with `choose`, we will first concentrate on finding a tree-recursive procedure to pick a box and then will write a memoized version of that. The input to the procedure will be the weight of the box we would like to buy and a list of all the chocolates that are available in the store. Here is an example of one such list, constructed from the chocolates available at a small store in Ohio:

```

(define shirks-chocolates-rated-by-max
  (list (make-chocolate 'caramel 'dark 13 10)
        (make-chocolate 'caramel 'milk 13 3)
        (make-chocolate 'cherry 'dark 21 3)
        (make-chocolate 'cherry 'milk 21 1)
        (make-chocolate 'mint 'dark 7 3)
        (make-chocolate 'mint 'milk 7 2)
        (make-chocolate 'cashew-cluster 'dark 8 6)
        (make-chocolate 'cashew-cluster 'milk 8 4)
        (make-chocolate 'maple-cream 'dark 14 1)
        (make-chocolate 'maple-cream 'milk 14 1)))

```


This procedure is similar to `choose` in that the `else` clause has two recursive calls; thus we would expect a worse-case scenario where the time complexity is roughly exponential. To improve the time complexity, we will try to memoize `pick-chocolates`. As with `choose`, we will need a two-dimensional table. One dimension will correspond to the weight. In other words, we can use the numbers from zero up to and including the weight limit to index the columns, say. The other dimension will correspond to the list of available chocolates in some way. But we must use integers for indexing the rows; we can't use lists. One way to get around this problem is to use the length of each list. Thus, if we're using the list of Shirk's chocolates given above, the row with index 10 would correspond to the whole list of chocolates, the row with index 9 would correspond to the `cdr` of that list, and so on. The entries of the table will be boxes of chocolates. To be precise, the entry in the i th row and j th column will be the best box of chocolates weighing at most j grams and restricted to the last i elements of the list of available chocolates.

▶ Exercise 12.11

We assume that the weight limit and the weight of each kind of chocolate is an integer number of grams. Why is this assumption necessary?

Now that we know how our table works, writing the memoized version of `pick-chocolates` is very straightforward. As with `choose` and `walk-count`, we will want to construct a table and fill it with the value `#f`. The rest of the construction is also essentially the same as before. The one substantial novelty is that we will need to use the *length* of the chocolates list for indexing the rows of the table:

```
(define memoized-pick-chocolates
  (lambda (chocolates weight-limit)
    (let ((table (make-table (+ (length chocolates) 1)
                             (+ weight-limit 1))))
      (define pick-chocolates
        (lambda (chocolates weight-limit)
          (cond ((null? chocolates) (make-empty-box))
                ((= weight-limit 0) (make-empty-box))
                ((> (chocolate-weight (car chocolates))
                    weight-limit) ; first too heavy
                 (pick-chocolates-subproblem (cdr chocolates)
                                              weight-limit))
                (else
                 ;;(continued))
```

```

(better-box
  (pick-chocolates-subproblem
    (cdr chocolates) ; none of first kind
    weight-limit)
  (add-chocolate-to-box
    (car chocolates) ; at least one of the first kind
    (pick-chocolates-subproblem
      chocolates
      (- weight-limit
         (chocolate-weight (car chocolates))))))))))
(define pick-chocolates-subproblem
  (lambda (chocolates weight-limit)
    (ensure-in-table! chocolates weight-limit)
    (table-ref table (length chocolates) weight-limit)))
(define ensure-in-table!
  (lambda (chocolates weight-limit)
    (if (table-ref table (length chocolates) weight-limit)
        'done
        (store-into-table! chocolates weight-limit))))
(define store-into-table!
  (lambda (chocolates weight-limit)
    (table-set! table (length chocolates) weight-limit
                (pick-chocolates chocolates weight-limit))))
(table-fill! table #f)
(pick-chocolates chocolates weight-limit)))

```

▶ Exercise 12.12

We used a tree-recursive procedure, `count-combos`, in Section 7.5 to determine how many ways there were to redeem for prizes the 10 tickets one of our sons won playing Whacky Gator at the local arcade.

- a. Since we wrote Chapter 7, our children have grown older and are better Gator Whackers. Empirically see how well (or poorly) the tree-recursive `count-combos` procedure you wrote in Chapter 7 can accommodate this by seeing how the time grows as the number of tickets to redeem grows.
- b. Write a memoized version, and empirically compare it with your prior version.

12.4 Dynamic Programming

Although memoization can dramatically improve the performance of a tree-recursive procedure, the memoized procedure still generates a recursive process. We saw that

we could fill out the table in an iterative fashion with the Fibonacci example, using dynamic programming. In this section, we will show how to use dynamic programming to rewrite `choose`. Then we will consider another example, with applications ranging from document management to molecular biology.

First, let's look at the binomial coefficients, that is, the numbers calculated by the procedure `choose`. If we look at the table of values at the end of computing (`memoized-choose 9 4`), the table is as follows:

```
#f #f #f #f #f
1 1 #f #f #f
1 2 1 #f #f
1 3 3 1 #f
1 4 6 4 1
1 5 10 10 5
#f 6 15 20 15
#f #f 21 35 35
#f #f #f 56 70
```

As you can see, not all of the entries in the table wound up getting filled in because not all of them had any bearing on computing $C(9, 4)$. For example, the `#f` in the lower left corner corresponds to $C(8, 0)$; although this entry could legally be filled in with the value 1, there was no reason to do so, because it did not arise as a subproblem in computing $C(9, 4)$. The `#f` values in the upper right portion of the table are more interesting. These correspond to values of $C(i, j)$ where $i < j$. In particular, the far upper right entry is for $C(0, 4)$, the number of ways of choosing four items when you don't have any to choose from. As before, these entries play no role in computing $C(9, 4)$. However, they are a little different from the values in the lower left: Up until now we haven't specified what the correct value is for $C(i, j)$ when $i < j$; our `choose` procedure doesn't handle this case. To keep our dynamic programming version of `choose` simple, we'll have it fill in the whole table. To make this possible, we'll have to add one more case to the definition of `choose`. If you ask how many ways there are to choose k items out of n , and $k > n$, there are 0 ways to do it. Thus, the table as filled in by the dynamic programming version will be

```
1 0 0 0 0
1 1 0 0 0
1 2 1 0 0
1 3 3 1 0
1 4 6 4 1
1 5 10 10 5
1 6 15 20 15
1 7 21 35 35
1 8 28 56 70
```

Now that we've straightened out what needs doing—and in particular, that the upper right triangular portion of the table gets filled in with zeros—we can write the dynamic programming version much as before:

```
(define dp-choose
  (lambda (n k)
    (let ((table (make-table n (+ k 1))))
      (define choose
        (lambda (n k)
          (cond ((< n k) 0) ; this is the new case
                ((= n k) 1)
                ((= k 0) 1)
                (else (+ (choose-subproblem (- n 1) (- k 1))
                        (choose-subproblem (- n 1) k))))))
      (define choose-subproblem
        (lambda (n k)
          (table-ref table n k)))
      (define store-into-table!
        (lambda (n k)
          (table-set! table n k (choose n k))))
      (from-to-do 1 (- n 1)
        (lambda (row)
          (from-to-do 0 k
            (lambda (col)
              (store-into-table! row col))))))
      (choose n k))))
```

▶ Exercise 12.13

Now that we have added a case for $n < k$, we could eliminate the case for $n = k$. Explain why.

▶ Exercise 12.14

Write a dynamic programming version of the chocolate box problem in the previous section. You'll find it helpful to first write a procedure that when given a number, n , returns the last n elements of the list of chocolates.

For our second example, consider a problem that occurs in systems that keep multiple old versions of large files. For example, in writing this book, we used a program that kept each draft of a chapter. After the first few versions of a given chapter, the number of changes from one draft to the next was relatively small,

whereas the size of each draft was relatively large. Rather than storing all of the different versions, our system stores only the current one. For each prior draft, it stores a list of the changes between that draft and the next. Now, the smaller this list of changes is, the less space we'll need to store it. Thus we'd like to find the smallest possible list of changes to convert one version into the next.

We will look at a somewhat simplified version of this problem. To be more precise, suppose we have two vectors of symbols, and we want to convert the first vector to the second by doing one of three things. We can insert a symbol into the first vector (in any one position), we can delete one occurrence of a symbol from the first vector, or we can replace one occurrence of a symbol with another symbol. What is the minimal number of changes we need to make to convert the first vector to the second? What are the changes that need to be made? We will answer the first question using dynamic programming and then outline a way of modifying that solution to find an answer to the second. Even without the modifications our procedure could be useful—it could be used to determine how similar two documents are, or, equally well, how similar two DNA sequences are.

We'll start by concentrating on the sizes of the two vectors, which we'll call *vector1* and *vector2*. Suppose *vector1* has n elements and *vector2* has m elements. If $n = 0$, the minimal number of changes we need to make is m because we will have to insert each element of that second vector into the first one. Similarly, if $m = 0$, the minimal number of changes we need to make is n because we'll need to do n deletions.

Now suppose both sizes are nonzero. We can look at the last element in each vector and determine what to do by seeing if these elements are the same or not. If they are the same, we simply need to find out how many changes are needed to convert the first $n - 1$ elements of *vector1* into the first $m - 1$ elements of *vector2*.

If, on the other hand, the last elements differ, we have three options:

1. We could delete the last element of *vector1* and then find the minimum number of changes needed to convert the first $n - 1$ elements of *vector1* into all of *vector2*.
2. We could find the minimum number of changes needed to convert all of *vector1* into the first $m - 1$ elements of *vector2* and then insert the last element of *vector2* at the end.
3. We could replace the last element of *vector1* with the last element of *vector2* and then find the minimum number of changes needed to convert the first $n - 1$ elements of *vector1* into the first $m - 1$ elements of *vector2*.

Note that in each of these cases, we decrease the size of at least one of the vectors, in the sense that we are looking at one fewer element. The vectors themselves don't shrink; we just focus attention on the first $n - 1$ or $m - 1$ elements rather than all n or m . For this reason, the **changes** procedure that follows is written in terms of an internal procedure named **changes-in-first-and-elements**,

where (`changes-in-first-and-elements i j`) computes the minimum number of changes needed to turn the first i elements of `vector1` into the first j elements of `vector2`. That is, it determines the number of changes needed in the first i and j elements (of vectors 1 and 2, respectively), hence the name. This will involve comparing the i th element of `vector1` with the j th element of `vector2` rather than comparing the last elements of the vectors. (Note that the i th and j th elements are in locations $i - 1$ and $j - 1$ because the locations are numbered from 0.)

Returning to the three possibilities listed above, let's quantify how many changes are needed in each case. We'll use $D(i, j)$ as a notation for the number of changes needed to transform the first i elements of `vector1` into the first j elements of `vector2`. Then in the first case we have the one deletion of the i th element of `vector1` plus the $D(i - 1, j)$ changes needed to finish the job, for a total of $1 + D(i - 1, j)$. Similarly, in the other two cases we get $1 + D(i, j - 1)$ and $1 + D(i - 1, j - 1)$ as the number of changes. Because we are interested in finding the *minimum* number of changes, we simply need to select whichever of these three possibilities is smallest; the built-in Scheme procedure `min` can do this for us.

In summary, here is the Scheme version of this algorithm:

```
(define changes
  (lambda (vector1 vector2)
    (let ((n (vector-length vector1))
          (m (vector-length vector2)))
      (define changes-in-first-and-elements
        (lambda (i j)
          (cond
            ((= i 0) j)
            ((= j 0) i)
            (else
             (if (equal? (vector-ref vector1 (- i 1))
                         (vector-ref vector2 (- j 1)))
                 (changes-in-first-and-elements (- i 1) (- j 1))
                 (min (+ 1 (changes-in-first-and-elements
                           (- i 1) j))
                      (+ 1 (changes-in-first-and-elements
                           i (- j 1)))
                      (+ 1 (changes-in-first-and-elements
                           (- i 1) (- j 1))))))))))
      (changes-in-first-and-elements n m))))
```

Because of those three recursive calls in the `else` clause, this algorithm is a very good candidate for either memoization or dynamic programming. In both cases,

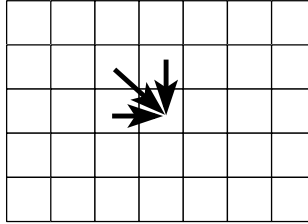


Figure 12.1 The three locations that influence a table entry are those above, to the left, and diagonally up and to the left.

we'll need to construct a table. We can use i and j to index the table, so we'll use a table with $n + 1$ rows and $m + 1$ columns, and we'll assume that the table entry in position (i, j) is $D(i, j)$, the minimal number of changes needed to convert the first i elements of the first vector to the first j elements of the second.

▶ Exercise 12.15

Write a memoized version of `changes`.

To write the dynamic programming version of this procedure, note that to compute one element of the table, we need to have already computed the element immediately above it, the one immediately to its left, and the one that is one row above and one column to the left of it. In other words, the table entry at position (i, j) is computed from the entries in positions $(i, j - 1)$, $(i - 1, j)$, and $(i - 1, j - 1)$, as shown in Figure 12.1. This means that we should fill out the table in an order such that the three entries at the tails of the three arrows are filled in *before* the entry at the heads of the arrows. There are several such orders; the most obvious two are either to go left to right across the top row, then left to right across the next row, etc., or alternatively to go top to bottom down the leftmost column, then top to bottom down the second column, etc. If we arbitrarily choose the former of these options (the row by row approach), we get this program:

```
(define dp-changes
  (lambda (vector1 vector2)
    (let ((n (vector-length vector1))
          (m (vector-length vector2)))
      (let ((table (make-table (+ n 1) (+ m 1))))
        (define changes-in-first-and-elements
          (lambda (i j)
            (cond
              ((= i 0) j)
              (else
               (let ((top (table-ref table (- i 1) j))
                     (left (table-ref table i (- j 1)))
                     (diag (table-ref table (- i 1) (- j 1))))
                 (min top left diag)))))))
          table)))
```

```

      ((= j 0) i)
      (else
       (if (equal? (vector-ref vector1 (- i 1))
                  (vector-ref vector2 (- j 1)))
           (changes-in-first-and-elements-subproblem (- i 1)
                                                      (- j 1))
           (min
            (+ 1 (changes-in-first-and-elements-subproblem
                  (- i 1) j))
            (+ 1 (changes-in-first-and-elements-subproblem
                  i (- j 1)))
            (+ 1 (changes-in-first-and-elements-subproblem
                  (- i 1) (- j 1))))))))))
(define changes-in-first-and-elements-subproblem
  (lambda (i j)
    (table-ref table i j)))
(define store-in-table!
  (lambda (i j)
    (table-set! table i j
                (changes-in-first-and-elements i j))))
(from-to-do 0 n
  (lambda (row)
    (from-to-do 0 m
      (lambda (col)
        (store-in-table! row col))))))
(changes-in-first-and-elements n m))))

```

▶ Exercise 12.16

We mentioned that this procedure uses just one possible valid ordering.

- Change the procedure to the other (column by column) valid ordering we mentioned, and verify that it still works.
- Give an example of an invalid order in which to fill in the table.
- Give a third example of a valid order.

▶ Exercise 12.17

The last line of the `dp-changes` procedure calculates $D(n, m)$ using the expression

```
(changes-in-first-and-elements n m)
```

It would be possible to instead just look this value up from the already filled-in table by changing the above expression to

`(changes-in-first-and-elements-subproblem n m)`

The analogous modification would not have been legal in `dp-choose`, however. Explain what the relevant difference is between the two procedures.

▶ Exercise 12.18

The previous versions of `changes` all used as their base cases $D(0, j) = j$ and $D(i, 0) = i$. As preparation for the next exercise, modify `dp-changes` so that the only base case is $D(0, 0) = 0$. You'll need to recursively define $D(0, j)$ as $1 + D(0, j - 1)$ for $j > 0$ and similarly define $D(i, 0)$ as $1 + D(i - 1, 0)$ for $i > 0$.

▶ Exercise 12.19

In this problem, we outline a way to modify the `dp-changes` from the previous exercise so that it produces a list of the changes to make to `vector1` in order to get `vector2`.

- a. First we need an ADT for the changes. Each change will have a certain type (replace, insert, or delete) and a position at which to do the change. Insertions and replacements will also need to know what symbol to use for the insertion or replacement. Construct a suitable ADT, with three constructors and three selectors.
- b. Next, we'll need an ADT for *collections of changes*, which is like a box of chocolates in the previous section—the point of using a “collection” ADT rather than a list is so that the number of changes in the collection can be kept track of, rather than repeatedly counted with `length`. You'll need `make-empty-collection` and `add-change-to-collection` constructors and `collection-size` and `collection-list` selectors.
- c. We will change the values in the table so that they are collections of changes rather than integers that indicate the minimum number of changes. In that case, instead of using the procedure `min` to select the smallest of three numbers, we'll need to select the smallest of three collections of changes. Write a procedure that gets three collections of changes, determines which has the smallest size, and returns that collection.
- d. Finally, write a version of `dp-changes` that produces the list of changes to make rather than the number of them.

12.5 Comparing Memoization and Dynamic Programming

Having seen a number of examples of both memoization and dynamic programming, let's consider what their relative strong points are. Keep in mind, though, that we're talking here about relatively fine differences between two very similar and similarly powerful techniques.

One benefit of memoization is that only those table entries that are needed are computed, whereas in dynamic programming all table entries get systematically computed in case they turn out to be needed. For some procedures this systematicness makes no difference; for example, the `memoized-walk-count` procedure fills in the whole table anyhow because it is all needed. However, consider the memoized and dynamic programming versions of `choose`. As the example tables in Section 12.4 show, the dynamic programming version can compute significantly more table entries. For some other problems, the difference is even more substantial.

The other principal advantage of memoization, relative to dynamic programming, is that the programmer doesn't have to figure out in what order the table needs to be filled. For the procedures we've seen (and most encountered in practice), this wasn't difficult. Occasionally, however, one encounters a problem where the correct ordering of the subproblems is a stumper, and then it is wise not to bother figuring it out but rather to simply use memoization.

One of the biggest advantages of dynamic programming is that for some problems—such as computing Fibonacci numbers—the dynamic programming solution can be modified to use asymptotically less memory (in the case of `dp-walk-count`, $\Theta(1)$ instead of $\Theta(n)$). This is possible because the Fibonacci recurrence, $F_n = F_{n-1} + F_{n-2}$, is an example of a *limited history recurrence*, in which only a limited number of preceding values (here 2) need to be remembered in order to compute a new value. Thus, rather than using an n -element vector for the table of values, it is possible to just keep cycling through the positions in a two-element vector, reusing the same two storage locations over and over for the most recent two Fibonacci numbers. A similar savings is possible in other limited-history recurrences.

As you can see, the most substantial differences between memoization and dynamic programming only arise in some problems, not in all. For those problems where none of these special considerations apply, professional programmers generally choose dynamic programming because if one is careful about the programming details, it can be slightly more efficient (by a constant factor—not a better asymptotic order of growth). On the other hand, memoization is a tool you can quickly and easily reach for, with less careful analysis.

12.6 An Application: Formatting Paragraphs

In this section we consider a problem that is encountered by many text formatting or word processing programs. How do we find the best way to break a paragraph into

separate lines? To be more precise, suppose we have all the words and punctuation marks in a paragraph typed in somehow, and we need to decide how to break that paragraph into lines. We are only allowed to break lines in between words; that is, we can't hyphenate words. We are also concerned about the amount of white space that is left over on the lines. Ideally, the words on each line and the spaces between those words would fill the entire width of the line, with no leftover space. Moreover, if there is leftover space, we'd prefer that it was reasonably evenly spread among the lines, with just a little per line, rather than some lines having large amounts of excess space. Of course, we realize that the last line of the paragraph may need to have a large chunk of white space, so we won't count the amount of leftover white space that appears on that line.

Let's assume that the input to this problem will be a list of words, where each word is a string that can contain letters, digits, punctuation marks, etc., but not any spaces. We can find the width a word will occupy on the printed page by using a `string-width` procedure. A simple version of this procedure would simply say that each character occupies one unit of space; that works for fixed-width type fonts like `this one`. If we make that simplifying assumption, the width of a string would be the same as its length in characters, and we could simply do the following definition:

```
(define string-width string-length)
```

If you want to make our program work with type fonts in which the characters vary in width, you'll simply need to redefine `string-width` to take this fact into account. We will also assume that we know the maximum width of a line, measured in the same units as `string-width` uses. So, because our simple `string-width` returns the width measured in characters, the maximum line width would be expressed in characters too.

We would like the output from our formatting procedure to be a list of lists of words. Each list of words represents a line in the paragraph. The amount of space that a given line takes up will be the sum of the widths of the words in that line plus the width of the spaces that go in between the words. The width of each space can be given the name `space-width`; with widths measured in characters this would simply be

```
(define space-width 1)
```

If `string-width` were changed to report widths in some other unit to accommodate a variable-width font, `space-width` would need to be changed to reflect the width of the space character in that font. The leftover space on a line is simply the difference between the maximum line width and the amount of space that the line uses. The amount of space the line uses can be computed as follows, using the `sum` procedure from Chapter 7:

```
(define line-width
  (lambda (word-widths)
    (+ (sum word-widths) ; total width of words
       (* (- (length word-widths) 1) ; number of spaces
          space-width))) ; each this wide
```

If we didn't care about having a large chunk of white space on any one line, we could measure how good a particular solution to our problem is by simply adding up the amount of leftover space on all of the lines but the last one. The smaller this number is, the better the solution is. However, we really do care about huge chunks of white space. In other words, having nine excess spaces on one line and one on another is not as good as having five on each. One way to adjust for this problem would be to add up the cubes of the amounts of leftover space, because cubing would make that huge chunk count for a lot more. (We could take some other power, as well. In general, the higher the power, the greater the penalty is for having one or two big amounts of leftover space.) We call this cubed leftover space the *cost*; the following procedure computes the cost of a line:

```
(define line-cost
  (lambda (word-widths max-line-width)
    (expt (- max-line-width (line-width word-widths))
          3)))
```

In summary, we want to find the best way to break our paragraph into lines, where *best* means that we want to minimize the sum of the line costs on all of the lines except for the very last line. We will assume that we're given as input the maximum line width and a list of strings. Our output is a list of lists of strings.

One approach to this problem is to first simplify it somewhat by taking the list of strings and converting it into a list of numbers, namely, the widths of the strings. Working from this list of widths, we will initially produce as output simply a list of integers, where each of these integers specifies how many words are in the corresponding line. Using this list, we can then chop up the original list of strings into the final list of lists of strings. In other words, our overall formatting process has three phases: preprocessing the list of strings into a list of widths, doing the main decision-making about how many words to put on each line, and then postprocessing to get a list of lists of strings. The following two exercises take care of the preprocessing and postprocessing stages.

Exercise 12.20

Write a procedure that will take a list of strings and convert it into a list of numbers that correspond to the string widths.

 **Exercise 12.21**

Write a procedure called `make-lines` that takes a list of elements and a list of integers and breaks the list of elements into sublists. For example, `(make-lines '(a b c d e f g h i j) '(2 3 5))` has a value of `((a b) (c d e) (f g h i j))`. You should assume that the integers in the second list add up to the number of elements in the first list.

Now we can assume that our problem is to take a list of word widths and find the best number of words to put on each line. We will work recursively by concentrating on the first line. Suppose that we have n words total and we decide to put some number of them, k , on the first line. If we break the remaining $n - k$ words into lines in the best possible way, the overall line breaking will be the best that is possible given the decision to put k words on the first line. Therefore, all we have to do is experiment with different values of k and see which gives the best result, in each case recursively invoking the procedure to optimally divide up the remaining $n - k$ words.

As we experiment with different values of k , we are looking to see which results in the best solution to the line-breaking problem. How do we tell which solution is best? One way to do this is to associate a cost with each solution. The cost of a solution is the sum of the costs of all of the lines except for the last one. The best solution is one with a minimal cost.

Now, our solutions will consist of a list of numbers that tell us how many words to put onto each line. We can improve the efficiency of our program by connecting this list with its associated cost.

 **Exercise 12.22**

Construct an abstract data type for solutions that will glue together a list of integers, called `breaks`, and a number, called `cost`. You should have one constructor and two selectors:

```
(make-solution breaks cost)
(breaks solution)
(cost solution)
```

 **Exercise 12.23**

Write a procedure `better-solution` that when given two solutions returns the one with the lower cost. If the two have equal costs, either can be returned.

As the paragraph formatting procedure systematically experiments with varying numbers of words on the first line, it needs to determine the best solution that results. One way we can arrange for this is with a higher-order procedure, `best-solution-from-to-of`, much like `from-to-do`. Like `from-to-do`, it will apply a procedure to each integer in a range. Each time the procedure is applied, it is expected to produce a solution as its result, and the best of these (as determined by your `better-solution` procedure) gets returned:

```
(define best-solution-from-to-of
  (lambda (low high procedure)
    (if (= low high)
        (procedure low)
        (better-solution (procedure low)
                         (best-solution-from-to-of (+ low 1) high
                                                    procedure))))))
```

When this `best-solution-from-to-of` procedure is used to try out the different possibilities for how many words go on the first line, the first argument (`low`) will be 1, because that is the minimum number of words that can be put on a line. How about the second argument? What is the maximum number of words that can be put on the first line? This depends on how wide the words are and what the maximum line width is.

Exercise 12.24

Write a procedure `num-that-fit` that takes two arguments. The first is a list of word widths and the second is the maximum line width. Your procedure should determine how many of the words (from the beginning of the list) can be fit onto a line. Remember to account for the spaces between the words, each of which has the width named `space-width`.

At this point, we are ready to write the `format-paragraph` procedure itself. It takes as arguments the list of word widths for the paragraph and the maximum line width. It returns a solution object, which contains the best possible set of breaks together with the cost of that set of breaks. (Remember that the breaks are the number of words on each line.) This procedure first checks for two special cases:

1. If all the words can be fit on a single line, that is the best solution and has cost 0 because excess space on the last line isn't counted.
2. If no words can be put on the first line without overfilling it (i.e., even the first word alone is too wide for the maximum line width), an error message is given because the given formatting problem is insoluble.

Other than these two special cases, the procedure follows the strategy we outlined earlier of systematically trying all the possibilities for how many words to put on the first line:

```
(define format-paragraph ;returns solution
  (lambda (word-widths max-line-width)
    (let ((most-on-first (num-that-fit word-widths max-line-width)))
      (cond ((= most-on-first (length word-widths))
             (make-solution (list most-on-first) 0)) ; all on first
            ((= most-on-first 0)
             (error "impossible to format; use shorter words"))
            (else
             (best-solution-from-to-of
              1 most-on-first
              (lambda (num-on-first)
                (let ((solution-except-first-line
                      (format-paragraph (list-tail word-widths
                                                  num-on-first)
                                         max-line-width)))
                  (make-solution
                   (cons num-on-first
                        (breaks solution-except-first-line))
                   (+ (cost solution-except-first-line)
                     (line-cost (first-elements-of
                                num-on-first
                                word-widths)
                               max-line-width)))))))))))))
```

Exercise 12.25

Put this procedure together with the ones you wrote for computing the list of widths from a list of strings, the `breaks` selector you wrote, and the `make-lines` procedure you wrote. This should let you format a list of words into a list of lists of words, one per line. Try this out with some short lists of words and narrow maximum line widths. Try scaling up the number of words and/or the maximum line width; how tolerable is the growth in time?

Of all of the procedures we've considered so far, this one probably makes the most tree-recursive calls because it tries out all possible numbers of words to place on each line. This fact makes it a very good candidate for memoizing or for using dynamic programming.

The various subproblems solved in the course of formatting a paragraph consist of formatting tails of that paragraph, but with the same maximum line width. Thus only a one-dimensional table (vector) is needed, indexed by the number of words. That is, the entry at position n in the vector will contain the solution object showing the best way to break the last n words of the paragraph into lines, which means that the length of the `word-widths` list is used to index the table, much as the length of the `chocolates` list was used as an index in the chocolate box problem.

▶ Exercise 12.26

Write a memoized version of `format-paragraph`. Test your memoized version to be sure it produces the same answers as the nonmemoized version. Also, empirically compare their speed as the number of words and/or the maximum line width grows.

In writing a dynamic programming version, it will be helpful to have a procedure that given a number n computes the tail of length n of the `word-widths` list. Again, this problem is analogous to the situation in the chocolate-box problem.

▶ Exercise 12.27

Following the hint just given, write a dynamic programming version of `format-paragraph`. Again, test that it produces the same results as the other versions and empirically compare its performance.

Review Problems

▶ Exercise 12.28

We implemented two-dimensional tables in terms of one-dimensional vectors by representing each table as a vector of vectors, one per row. The most obvious alternative would be to continue to have a vector of vectors but make it be one per column. However, we have another, less obvious alternative: We can store all the elements of an $n \times m$ table in a single vector of length nm . For example, the 15 elements of a 3×5 table can be stored in the 15 elements of a vector of length 15. The two most straightforward ways to do this alternative are either to store first the elements from the first row, then those from the second row, etc., or to start with the elements from the first column, then those from the second column, etc. Note that in either case you will need to store the width and height of the table, not only because `table-width` and `table-height` are selectors for the table ADT, but more crucially in order to calculate the appropriate index into the vector of table values. One easy thing we can do is to let a table be represented by a three-element

vector, where the first two elements represent the width and height and the third element is the vector that stores the table values. Reimplement the table ADT using either of these variations (i.e., storing the elements row by row or column by column in a single vector).

▷ Exercise 12.29

Suppose you are the instructor of a course with n students, and you want to divide the students into k teams for a project. You don't care how many students are on each team (for example, that they be equal), except that each team must have at least one student on it, because otherwise there wouldn't really be k teams. How many ways can you divide the students up?

This problem can be analyzed much as we analyzed `choose`. The first student can either be in a team alone or in a team with one or more others. In the first case, the remaining $n - 1$ students need to be divided into $k - 1$ teams. In the second case, the remaining $n - 1$ students need to be divided into k teams, and then one of those k teams needs to be chosen to add the first student to. So, if we use $S(n, k)$ to denote our answer, we have $S(n, k) = S(n - 1, k - 1) + kS(n - 1, k)$. (We're using the letter S because numbers computed in this way are conventionally called *Stirling numbers of the second kind*.) Of course, this equation is only the recursive case; you'll need one or more base cases as well.

Write a tree-recursive procedure for computing $S(n, k)$ and then make it more efficient using memoization or dynamic programming. (Or you could first rewrite it using memoization and then using dynamic programming.)

▷ Exercise 12.30

We defined the procedure `from-to-do` earlier in this chapter. This procedure is very similar to the so-called FOR loop from other languages. Because we learned in Chapter 10 how to add features to Scheme, it would be nice to add actual FOR loops to Mini-Scheme. To illustrate what we mean, consider the following use of the procedure `from-to-do`:

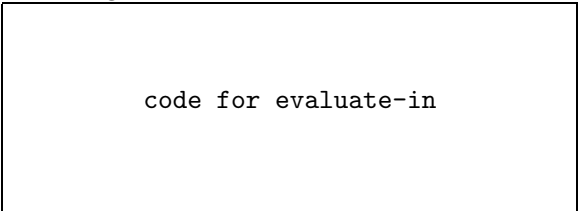
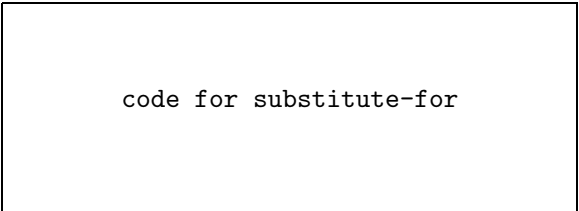
```
(from-to-do 2 (* 3 4)
           (lambda (n) (display (* 2 n))))
```

With FOR loops, we could instead write:

```
(for n = 2 to (* 3 4) do (display (* 2 n)))
```

This exercise will work through the details of adding FOR loops to Mini-Scheme.

Let's choose to implement FOR loops using a new AST constructor `make-for-ast`. The skeleton for `make-for-ast`, with the important code left out, is as follows:

```
(define make-for-ast
  (lambda (var start-ast stop-ast body-ast)
    (define the-ast
      (lambda (message)
        (cond ((equal? message 'evaluate-in)
              (lambda (global-environment)
                
                ))
              ((equal? message 'substitute-for)
              (lambda (value name)
                
                ))
              (else (error "unknown operation on a for AST"
                          message))))))
    the-ast))
```

- Write the pattern/action that needs to be added to `micro-scheme-parsing-p/a-list` for FOR loops.
- Add the code for `evaluate-in`. (*Hint*: You can use the procedure `from-to-do`.)
- Add the code for `substitute-for-in`.

Exercise 12.31

Imagine the following game: You are given a path that consists of white and black squares. The exact configuration of white and black squares varies with the game but might for example look as follows:



You start on the leftmost square (which we'll call square 0), and your goal is to move off the right end of the path in the least number of moves. However, the rules stipulate that

- If you are on a white square, you can move either 1 or 2 squares to the right.
- If you are on a black square, you can move either 1 or 4 squares to the right.

How can we determine, for a given path, the least number of moves we need? One way to compute this number is to write a procedure `fewest-moves` that takes a path and a position on the path and computes the minimum number of moves from that position. Thus, to determine the minimum number of moves for the preceding path, we would evaluate:

```
(fewest-moves (vector 'white 'black 'white 'white 'white
                    'black 'white 'white 'black 'white
                    'black 'white 'white 'black 'black
                    'white 'white 'white)
              0)
```

Note that we pass the path as a vector as well as the current square (in this case 0). Here is one way to implement `fewest-moves`:

```
(define fewest-moves
  (lambda (path i) ; path is a vector
                    ; i is the position within path
    (cond ((>= i (vector-length path)) ; right of path
           0)
          ((equal? (vector-ref path i) 'white)
           (+ 1 (min (fewest-moves path (+ i 1))
                     (fewest-moves path (+ i 2))))))
          (else
           (+ 1 (min (fewest-moves path (+ i 1))
                     (fewest-moves path (+ i 4))))))))
```

- a. Write a memoized version of `fewest-moves`.
- b. Write a dynamic programming version of `fewest-moves`. Be sure to remember that the simplest subproblems, in the sense of being closest to the base case, do *not* correspond to smaller values of the argument i in this problem.
- c. Modify `fewest-moves`, and your memoized and/or dynamic programming version of it, to produce a list of moves rather than just the number of moves that are necessary.

▶ Exercise 12.32

The `ps` procedure that follows calculates how many ways we can parenthesize an n -operand expression. For example, `(ps 4)` evaluates to 5 because there are five

ways to parenthesize a four-operand expression: $a - (b - (c - d))$, $a - ((b - c) - d)$, $(a - b) - (c - d)$, $(a - (b - c)) - d$, and $((a - b) - c) - d$.

```
(define from-to-add
  (lambda (start end f)
    (if (> start end)
        0
        (+ (f start)
            (from-to-add (+ start 1) end f)))))

(define ps
  (lambda (n)
    (cond ((= n 1) 1)
          ((= n 2) 1)
          (else
           (from-to-add 1 (- n 1)
                        (lambda (k)
                          (* (ps k) (ps (- n k))))))))))
```

- Write a memoized version of `ps`.
- Write a dynamic programming version of `ps`.

► Exercise 12.33

The function $h(n)$ is defined for nonnegative integers n as follows:

$$h(n) = \begin{cases} 1 & \text{if } n < 2 \\ h(n-1) + h(n-2) & \text{if } n > 2 \text{ and } n \text{ is odd} \\ h(n-1) + h(n/2) & \text{if } n \geq 2 \text{ and } n \text{ is even} \end{cases}$$

- Write a dynamic programming procedure for efficiently calculating $h(n)$.
- Is it possible to modify the procedure so that it stores all the values it needs in a vector of fixed size, as `walk-count` can be modified to store the values it needs in a two-element vector? (A vector of “fixed size” is one with a size that does not depend on the parameter, n .) Justify your answer.

► Exercise 12.34

The following `best` procedure determines the best score that is possible on the following puzzle. You are given a list of positive integers. Let’s say the first one is k .

You can either claim k points for yourself and then skip over k more numbers after the k , or you can just skip over the k itself without claiming any points. These options repeat until the numbers are all gone. When skipping over the next k numbers, if there aren't k left, you just stop. For example, given the list

```
(2 1 3 1 4 2)
```

your best bet is to first claim 2 points, which means you have to skip over the first 1 and the 3, then pass up the opportunity to take the second 1, so that you can take the 4, which then causes you to skip the final 2. Your total score in this case is 6; had you played less skillfully, you could have gotten a lower score. The `best` procedure returns the best score possible, so `(best '(2 1 3 1 4 2))` would return 6.

```
(define best
  (lambda (l)
    (if (null? l)
        0
        (let ((k (car l))
              (rest (cdr l)))
          (max (best rest)
               (+ k (best (skip-of k rest))))))))

(define skip-of
  (lambda (n l) ;skip first n elements of l
    (if (or (= n 0) (null? l)) ;l can be shorter than n
        l
        (skip-of (- n 1) (cdr l)))))
```

- a. Write a memoized version of `best`.
- b. Write a dynamic programming version of `best`.

Chapter Inventory

Vocabulary

exponential growth
Fibonacci numbers
memoization
dynamic programming

binomial coefficients
knapsack problem
Stirling numbers of the second kind
FOR loops

Slogans

Much computation, little variety sign

Abstract Data Types

two-dimensional tables
chocolates
boxes

changes
collections (of changes)
solutions (line breaking)

New Predefined Scheme Names

vector-fill!

Scheme Names Defined in This Chapter

walk-count
memoized-walk-count
walk-count-subproblem
ensure-in-table!
store-into-table!
dp-walk-count
from-to-do
dp-walk-count-2
choose
make-table
table-ref
table-set!
table-height
table-width
table-fill!
display-table
memoized-choose
make-chocolate
chocolate-filling
chocolate-covering
chocolate-weight
chocolate-desirability
make-empty-box
box-chocolates
box-weight
box-desirability
add-chocolate-to-box
better-box

shirks-chocolates-rated-by-max
pick-chocolates
memoized-pick-chocolates
dp-choose
changes
dp-changes
make-empty-collection
add-change-to-collection
collection-size
collection-list
string-width
space-width
line-width
line-cost
make-lines
make-solution
breaks
cost
better-solution
best-solution-from-to-of
num-that-fit
format-paragraph
make-for-ast
fewest-moves
from-to-add
ps
best
skip-of

Notes

We mentioned briefly that the `changes-dynamic` procedure for computing the minimum number of changes needed to convert one vector of symbols into another has applications in comparing the sequences occurring in biological molecules such as DNA. For a discussion of this application, as well as the application of this algorithm and its relatives to problems in speech processing and other areas, see [46].

The problem of breaking a paragraph into lines in a visually appealing way by minimizing the sum of the cubes of the amount of excess space on each line is a gross simplification of the actual approach used by the `TEX` program, which was used to format this book. That program uses a dynamic programming algorithm but takes into account not only the amount of excess space on each line but also the possibility of hyphenation and a number of esoteric considerations. For a complete description of the quantity that it minimizes, see [34]. For the program itself, see [33].