CHAPTER 10

Principles of Problem Solving

Chapter Objectives

This chapter presents an overview of framing the solutions to problems:

- We begin with simple problems that can be solved in a single step
- We continue to strategies for solving more complex problems involving data collections by dividing the solution into the following fundamental operations that can be performed on any collection of data:
  - Inserting
  - Traversing
  - Building
  - Mapping
  - Filtering
  - Summarizing
  - Searching
  - Sorting

Then we will briefly discuss how to combine these fundamental tools to solve more complex data manipulation problems.

Introduction

Programming is really all about applying the computer as a tool to solve problems. One of the most difficult tasks facing novice programmers is the blank sheet of paper. Faced with a problem you have never seen before, how do you start to solve it? The problem-solving style recommended in this text is first to identify the basic character of the data and the basic operation(s) we are asked to perform. If these two ideas are clear, we can create a template or outline of the solution and begin to fill in the blanks.

As we gain more experience with the language, we have more computing tools to apply, and we can attack larger, more complex problems. We now have sufficient tools available to consider a more principled approach to data manipulation and problem solving. We will begin with the typical plan for solving simple problems in one step and then continue to consider assembling multiple steps to solve more complex problems.

10.1 Solving Simple Problems

In Chapter 2 we saw the basic plan for solving simple problems:

- Define the input data
- Define the output data
- Discover the underlying equations to solve the problem
- Implement the solution
- Test the results
- Repair the code until it conforms to the specifications

This plan works whenever the problem is simple enough to be able to visualize the complete solution. Typically, however, problems are more complex and require a number of steps to be assembled.

10.2 Assembling Solution Steps

Problem complexity frequently comes in the form of data collections that need to be transformed into other collections or summarized as intermediate results. Identifying the operation(s) that will create the output from the input requires some experience. The rest of this chapter provides some guidelines for identifying elementary steps whose solutions can be combined to create solutions to many complex problems.

10.3 Summary of Operations

First, we document the operations we expect to be able to perform on collections. Table 10.1 lists the generic operations, a brief description of each, and a discussion of the consequences. The following paragraphs illustrate these fundamental operations, using the array of structures from Chapter 7 as examples. The discussion of each step takes the form of a written description, a flowchart, and a template for writing the code.

Style Points 10.1

It is conceivable—and in fact, a common practice—to combine multiple operations into one computing module, but it is poor abstraction and leads to code that is hard to understand and/or debug.

Table 10.1 Taxonomy of solution steps

Operation	Description	Consequence
Insert	Inserts one item into a collection	Collection with one more item
Build	Creates a collection from a data source (external file or traversing another collection); usually accomplished by starting with an empty collection and inserting one item at a time	A new collection of data
Traverse	Touches each item of data in the collection—frequently used to display or copy a collection	The collection is unchanged
Map	Changes the content of some or all of the items in the collection	A new collection of the same length, but the content of some or all items is changed
Filter	Removes some items from the collection	A new collection with reduced length, but the content of the items remains unchanged
Fold	Traverses the collection, summarizing the contents with a single result (e.g., sum, max, or mean)	A single result summarizing the collection in some way; the collection is unchanged
Search	Traverses the collection until an item matches a given search criterion and then stops, returning the result	A single result or the indication that the desired match was not achieved; the collection is unchanged
Sort	Puts the collection in order by some specific criterion	A new collection of the same length

10.3.1 Basic Arithmetic Operations

The simple problem solution described in Section 10.1 frequently needs to be used as part of a larger problem solution. We include this activity in this list for completeness.

10.3.2 Inserting into a Collection

Inserting an item into a collection is a process usually used to build or maintain a collection of information. In this text, we have seen four basic data collection types to which insertion applies: vectors, arrays, cell arrays, and structure arrays. We will discuss the peculiarities of each collection and then the common processing algorithm that can be used to insert a new entry into the collection.

- **Vectors** are very flexible collections in the MATLAB language, and suffer only from the obvious limitation that one can add only numbers to a vector

- **Arrays** are as flexible as vectors, except that they require that new data be inserted a row or column at a time, and that the size of the new item must match the existing array dimensions
- **Cell arrays** can be indexed like numerical arrays and can contain any object; however, to compare one element to another usually requires a special-purpose comparison function
- **Structure arrays** as a collection behave like cell arrays, except that any structure inserted must have the same fields as those in the existing structure

In general, inserting into any of these collections involves insertion into the front of the collection, the back of the collection, or at some position in the middle in order to keep the collection in order by a specific comparison method.

Inserting into the front is accomplished by concatenating the new element before the existing collection. For example, adding `item` to the front of an existing cell array, `ca`, is accomplished as follows:

```
>> ca = [{item} ca] % note the braces needed for a cell array
```

Inserting at the back is accomplished by concatenating the new element after the existing collection. For example, adding `item` to the back of an existing cell array, `ca`, is accomplished as follows:

```
>> ca = [ca {item}] % note the braces needed for a cell array
```

Inserting in order is usually accomplished using a `while` loop. If we are inserting `item` into a collection `coll`, we will use a `while` loop to find the index of the insertion point, `ins`, and then concatenate the three parts of the new collection. Figure 10.1 shows the flowchart that applies here, and Template 10.1 shows the template for the general solution.

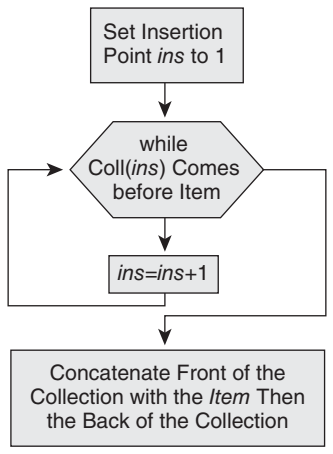

Figure 10.1 *Inserting in order*

Template 10.1 Template for inserting

```
%inserting item into a collection coll

<set insert point, ins, at the front>
while <insertion point in coll and
       item comes before coll(ins)>
   <move insertion point forward>
<end of the while loop>
<concatenate coll before ins with item and
     coll at and beyond ins>
```

For example, adding `item` in order to a vector, `v`, is accomplished as follows:

```
ins = 1;
while ins <= length(v) && before(item, v(ins))
      ins = ins + 1;
end
v = [v(1:ins-1) item v(ins:end)]
```

where `before(a,b)` is a generic comparator that determines whether `a` comes before `b` in the ordering scheme. Notice that this covers the cases where `item` must be the first or last item in the collection. Consequently, we could include the case of front or back insertion by having `before(a,b)` return `true` for inserting in the front and `false` for inserting at the back.

10.3.3 Traversing a Collection

Traversal involves moving across all elements of a collection and performing some step (not necessarily the same step) on each element without changing that element. Figure 10.2 and Template 10.2 illustrate the flowchart and basic template for traversing a collection. They assume that you are doing something like writing a file that needs to be initialized and finalized. These two steps may not always be required.

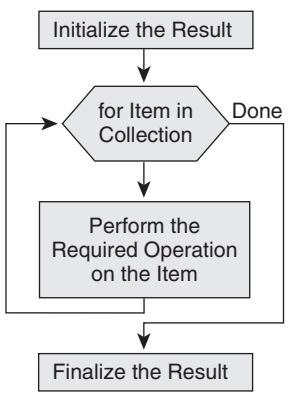

Figure 10.2 *Traversing a collection*

Template 10.2 Template for traversing

```
<initialize the result>
for item <across the whole collection>
   <operate on the item>
<end of the loop>
<finalize the result>
```

10.3.4 Building a Collection

In practice, frequently we combine traversal of one collection and building of another to copy data from one collection into another. Building a collection is the process of beginning with an empty collection and assembling data elements by inserting them one at a time into the new collection. The size of the collection increases continually until the process is finished. Figure 10.3 and Template 10.3 illustrate the algorithm for building a collection.

10.3.5 Mapping a Collection

The purpose of mapping is to transform a collection by changing the data in some or all of its elements according to some functional description without changing its length. It is distinct from traversal because its intent is to change

Template 10.3 Template for building

```
<initialize the new collection>
for item <across the data source>
   <extract the item>
   <insert item in new collection>
<end of the loop>
<finalize the new collection>
```

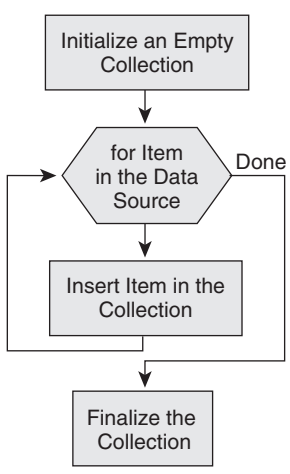

Figure 10.3 *Building a collection*

Template 10.4 Template for mapping

```
<initialize the result>
for item <across the whole collection>
   <extract the item>
   <modify the item>
   <insert modified item in the result>
<end of the loop>
<finalize the result>
```

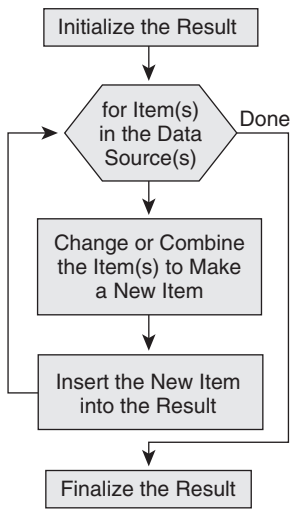

Figure 10.4 *Mapping a collection*

Style Points 10.2

A simpler example of collection building occurred when we built the CD collection initially by repeated calls to the `makeCD` method, as shown in Chapter 7 when we were inserting each item at the end of the collection. However, while that example seems to simplify the process of building the collection, it really did not. The data for the function calls had to be extracted from a CD listing and edited to construct the function calls—normally not an efficient or effective way to compose a collection. Such hard-wiring should generally be avoided.

the data elements. While many languages permit collections to be modified in place, the MATLAB language usually requires you to create a new collection. However, this is still considered mapping. The scalar mathematical and logical operations on vectors are good examples of mapping. Figure 10.4 and Template 10.4 illustrate the basic algorithm for mapping. As illustrated in the example of operations on vectors, mapping may involve combining two or more collections of the same length.

10.3.6 Filtering a Collection

Filtering involves removing items from a collection according to specified selection criteria. The data contents of the remaining items in the collection should not be changed, and the collection will usually be shorter than before.

Template 10.5 Template for filtering

```
<initialize the new collection>
for item <across the whole collection>
   <extract the item>
   if <keep the item>
      <insert item in new collection>
   <end if>
<end for>
<finalize the new collection>
```

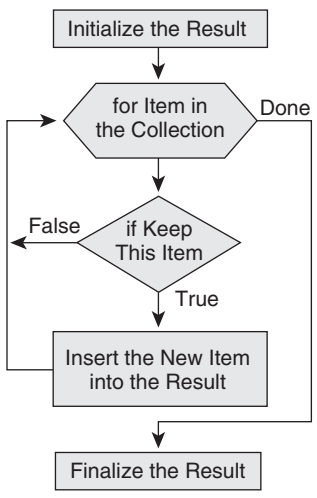

Figure 10.5 *Filtering a collection*

For example, we filter vectors by applying built-in logical operations and then indexing with the results to produce new, shorter arrays. Figure 10.5 and Template 10.5 illustrate the general algorithm for filtering a collection.

10.3.7 Summarizing a Collection

Folding is the name given to summarizing a collection. It is a special case of traversal where all of the items in the collection are summarized as a single result. The collection is not altered in size or values by the operation. Totaling, averaging, and finding the largest element in a vector are typical examples of folding. For example, we might want to find the CD with the best value in a collection. Figure 10.6 and Template 10.6 show the basic algorithm for folding a collection. The general form of a fold should be to initialize the summary value and then traverse the whole collection, updating the summary when necessary.

Template 10.6 Template for folding

```
<initialize the summary value>
for item <across the whole collection>
   <extract the item>
   <update the summary value>
<end for>
<finalize the summary value>
```

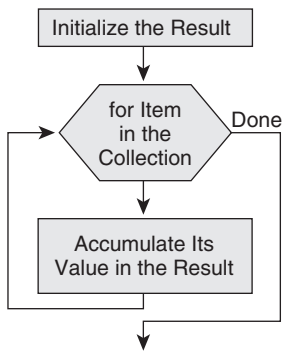

Figure 10.6 *Folding a collection*

10.3.8 Searching a Collection

Searching is the process of traversing the collection and applying a specified test to each element in turn, terminating the process as soon as the test is satisfied. This is superficially similar to filtering, except that it is not necessary to touch all the elements of the collection; the search stops as soon as one element of the collection matches the search criteria. If the criteria are extremely complex, it is sometimes advisable to perform a mapping or folding before the search is performed. Figure 10.7 and Template 10.7 illustrate one way to

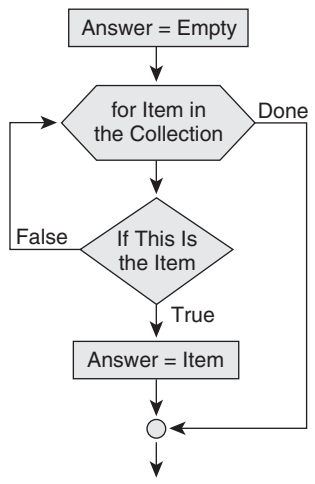

Figure 10.7 *Searching a collection*

Template 10.7 Template for searching

```
<initialize result to not succeeded>
for <item in the collection>
   if <found criteria>
       <set result to succeeded>
       <break the loop>
   <end if>
<end for>
<check for failure>
```

implement searching a collection using a `for` loop with a break exit. There are always two exit criteria from a search—finding what you seek and failing to find it. Searching can also be implemented with a `while` loop, but the multiple exit criteria make the code generally more complex.

10.3.9 Sorting a Collection

Sorting involves reordering the elements in a collection according to a specified ranking function that defines which item "comes before" another. Sorting is computationally expensive. However, if a large collection of data is stable—items are added or removed infrequently—but is frequently searched for specific items, keeping the data sorted can greatly improve the efficiency of the searches. Chapter 16 is devoted to the details of sorting algorithms, but the concept is included here to complete the list of operations we can perform on a collection.

10.4 Solving Larger Problems

Problem statements are rarely simple enough to be able to seize one of the above steps and solve the whole problem. Usually, the solution involves choosing a number of known operations and performing those operations in order to solve the complete problem. Solution steps are combined in one of two ways—in sequence or nested. When considering the overall strategy for solving a problem, one might identify steps A and B as contributing to the solution. Your logical statement might say either "do A and then B" sequential steps—or "for each part of A, do B"—nested steps.

For example, consider the baseball card problem originally proposed in Chapter 1. You have collected over the years a huge number of baseball cards, and you wish to find the names of the 10 "qualified" players with the highest lifetime batting average. To qualify, the players must have been in the league at least five years, had at least 100 plate appearances per year, and made less than 10 errors per year.

The first step is to build a collection containing the relevant information on the cards for each player, and the use of a structure array seems a good

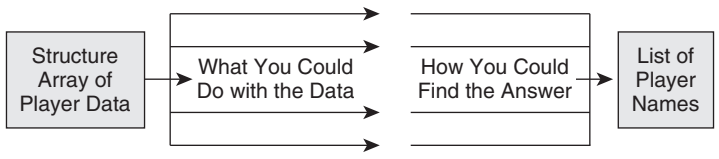

Figure 10.8 *Generalized problem solving*

choice. Next, we need to operate on this collection to solve the problem. Consider again the overall problem situation, as shown in Figure 10.8. The original data are the structure array containing all the player data. The final result is a list of 10 names of the qualified players with the highest batting averages. There may be more than one sequence of operations to solve this problem, and some may be more efficient than others.

First, we consider the operations that could be performed on the original data. Since the end result is a collection, it is unlikely that the first step would reduce the collection to one answer. This eliminates folding and searching. Since the collection is already built, we do not need to insert or build, leaving four possible operations to consider—traversal, mapping, filtering, and sorting.

Now, consider the last operation—it seems reasonable that the last thing to do is a mapping—taking the 10 selected structures and extracting the names.

Now, we must think about how to find these 10 structures. If we had a collection of qualified players sorted by their batting average, we could accomplish this with a special filter taking the first 10 from these sorted, qualified players. Backing up one more step, we can see that the sorted collection we need is just a sort of the qualified players, and we can chain these steps together to solve the whole problem.

10.5 Engineering Example—Processing Geopolitical Data

Imagine that you have decided to move your prosperous business overseas to the country with the most business-friendly environment. After considerable study, you decide that the best measure of friendliness would be to compute the rate of growth of the gross domestic product for candidate countries, subtract their rate of population growth, and use this measure to choose the best country. An Internet search provides an interesting source of data. Figure 10.9 shows an excerpt from a spreadsheet containing historical data for 154 countries from Penn World Table Version 6.1.[1] The data columns of interest to us contain the following information:

[1]Credit: Alan Heston, Robert Summers, and Bettina Aten, Penn World Table Version 6.1, Center for International Comparisons at the University of Pennsylvania (CICUP), October 2002.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	Country	Code	Year	POP	XRAT	PPP	cgdp	cc	ci	cg	P	pc	pg	pi	openc	cgnp
2	Angola	AGO	1960	4816.00	0.03	0.01	542.68	76.75	8.84	9.45	17.51	13.07	24.03	49.11	36.98	na
3	Angola	AGO	1961	4884.19	0.03	0.00	564.37	74.23	7.92	9.85	17.36	13.18	23.65	48.67	35.23	na
4	Angola	AGO	1962	4955.35	0.03	0.00	573.94	75.48	6.76	10.55	17.28	13.41	23.65	50.44	38.79	na
5	Angola	AGO	1963	5028.69	0.03	0.01	593.72	73.68	5.72	13.56	17.73	13.90	24.04	52.06	38.69	na
•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
36	Angola	AGO	1994	10627.18	59.51	53.32	1095.94	34.66	9.09	46.75	89.59	70.34	76.84	228.63	160.87	47.86
37	Angola	AGO	1995	10972.00	2750.23	1007.53	1244.73	41.86	9.43	57.65	36.63	29.58	31.91	96.85	146.58	48.19
38	Angola	AGO	1996	11316.94	128029.20	54873.28	1362.32	37.17	8.57	56.75	42.86	34.35	37.77	113.51	134.87	52.97
39	Angola	AGO	1997	na	na	na	na	na	na	na	na	na	na	na	na	59.09
40	Angola	AGO	1998	na	na	na	na	na	na	na	na	na	na	na	na	50.08
41	Angola	AGO	1999	na	na	na	na	na	na	na	na	na	na	na	na	44.52
42	Angola	AGO	2000	na	na	na	na	na	na	na	na	na	na	na	na	53.81
43	Albania	ALB	1991	3277.00	15.63	3.13	1605.36	82.81	6.92	36.66	20.04	23.78	12.02	17.70	54.89	98.10
44	Albania	ALB	1992	3225.00	75.03	10.53	1566.99	136.94	5.11	35.08	14.03	15.59	7.88	14.42	108.94	95.94
45	Albania	ALB	1993	3179.00	102.06	19.33	2031.94	109.39	12.54	25.73	18.94	20.80	10.51	20.01	77.14	99.02
•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
5795	Zambia	ZMB	1998	9665.71	1862.07	744.91	800.69	85.12	13.75	14.14	40.00	39.54	33.22	49.87	68.86	93.36
5796	Zambia	ZMB	1999	9881.21	2388.02	941.87	765.24	91.82	15.30	12.54	39.44	39.02	31.89	48.14	66.55	94.97
5797	Zambia	ZMB	2000	10089.00	3110.84	1157.63	840.97	86.33	15.38	12.34	37.21	37.70	29.54	40.65	70.45	95.88
5798	Zimbabwe	ZWE	1954	3011.69	0.71	0.37	400.19	66.89	41.48	4.03	50.97	60.59	135.99	29.92	77.30	na
5799	Zimbabwe	ZWE	1955	3127.52	0.71	0.36	429.04	65.87	50.95	3.47	50.40	60.80	136.52	31.10	78.43	na
5800	Zimbabwe	ZWE	1956	3264.42	0.71	0.36	471.08	63.51	54.77	3.53	50.08	62.11	136.86	30.53	74.27	na
•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
5842	Zimbabwe	ZWE	1998	12153.85	23.68	4.06	2799.85	77.66	10.75	13.39	17.16	14.97	22.03	26.87	91.96	93.25
5843	Zimbabwe	ZWE	1999	12388.32	38.30	6.12	2770.48	76.89	10.73	12.81	15.98	14.35	19.01	24.02	92.99	93.75
5844	Zimbabwe	ZWE	2000	12627.00	44.42	9.48	2607.03	69.23	8.62	22.44	21.33	19.26	23.63	31.96	62.61	96.62

Figure 10.9 *Spreadsheet samples*

- Country—Country name
- Code—Country code
- Year—Year in which the data in this row were recorded
- POP—Population that year
- XRAT—Exchange rate versus U.S. currency that year
- PPP—Purchasing power parity over GDP that year
- CGDP —Real gross domestic product per capita that year

Figure 10.9 also illustrates one of the weaknesses of spreadsheets: they are inherently two dimensional, and the data in this case are three dimensional; each country has several sets of data as functions of the year when the information was recorded. Therefore, the data must be massaged into a form more useful to us. A careful examination of the data also reveals the following challenges:

- The years in which the data were available vary from country to country—most have data from 1950 to 2000

Listing 10.1 Country analysis

```
    % build the country array
1.  worldData = buildData('World_data.xls');
2.  best = findBest(worldData);
3.  fprintf('best country is %s\n', ...
4.      worldData(best).name)
```

- There are some places within the numerical data where the values are not available, signified by the letters "na" at those locations

Our algorithm must take into account the variable number of years and the potential presence of strings within the data. Fortunately, the `cvsread(...)` function discussed earlier[2] recognizes this situation and inserts `NaN` in the numerical data fields. To ensure clarity and reliability in our solution, we need a careful design for this data processing task as follows.

- Looking at the end result desired, eventually we need to fold a collection of data about each country and choose the friendliest one.
- The information describing each country must include not only its name, but also vectors of the population and CGDP as a function of the year. It seems that a structure array by country would be an appropriate form for the data.
- Therefore, before actually solving the problem, we have to build this structure.
- Having built the structure, the folding operation to find the friendliest country follows the folding template shown in Section 10.3.7.

Listing 10.1 shows the script that accomplishes this analysis, although most of the work is actually done in the following functions.

In Listing 10.1:

Line 1: `worldData` will be a structure array containing the relevant data from the spreadsheet.

Line 2: `best` will be the index of the friendliest country according to the criteria defined in the function `findBest(...)`.

Lines 3–4: Here we can look up and print the name of the best country.

Listing 10.2 lists the function that builds the country data. The algorithm violates the best style by taking advantage of the logical ordering of the data in the spreadsheet to traverse the data from the spreadsheets simultaneously, filter out the data for each country in turn, and then map the available data for that country into the emerging structure array.

[2]For MATLAB users, xlsread(…) should be used here.

Listing 10.2 Building the country data

```
 1. function worldData = buildData(name)
    % read the spreadsheet into a data array
    % and a text cell array
 2.     [data txt] = xlsread(name);
 3.     country = ' '; % force the first data row
                       % to change the country
 4.     cntry_index = 0;
        % Traverse the data and cell arrays producing
        % an array of structures,
        % one for each country
 5.     for row = 1:length(data)
            % Because the text data in txt contains
            % the header row of the spreadsheet,
            % the data at a given row belongs to the country
            % whose name is at txt{row+1}.
            % if the country name changes,
            % begin a new structure.
 6.         if ~strcmp(txt{row+1}, country)
 7.             col = 1;
 8.             country = txt{row+1};
 9.             cntry_index = cntry_index + 1;
10.             cntry.year = 1;
11.             cntry.pop = 1;
12.             cntry.gdp = 1;
13.         end
14.         cntry.name = country;
15.         cntry.year(col) = data(row, 1);
16.         cntry.pop(col) = data(row, 2);
17.         cntry.gdp(col) = data(row, 5);
18.         col = col + 1;
19.         worldData(cntry_index) = cntry;
20.     end
21. end
```

In Listing 10.2:

Line 2: Reads the Excel spreadsheet—we need the numerical data and the text part for the names of the countries.

Lines 3 and 4: Initialize the results of the traversal, setting an unknown country name and the initial country count.

Lines 5–20: Traverse the rows of the numerical data.

Line 6: Since the numerical data skipped the header row, the name of the country corresponding to each row of data is in the text file at `row+1`. When the country changes, we step to the next country index, reset the year counter, `col`, for that country, and empty the structure used to accumulate the country data.

Line 7: Resets the counter that indexes the year storage for the current country.

Line 8: Saves the name of the new country to continue retrieving its data.

Line 9: Increases the country count.

Lines 10–12: Reset the structure used to store the vectors of data. This is crucial because the number of annual data items for all countries is not the same.

Lines 14–17: Add this row of data to the structure. Column 1 is the year, column 2 is the population, and column 5 is the CGDP.

Line 18: Moves to the next year.

Line 19: Saves all this in the structure array.

Listing 10.3 Folding the country data

```
 1. function besti = findbest(worldData)
    % find the index of the best country
    % according to the criterion in the function
    % fold
 2.     best = fold(worldData(1));
 3.     besti = 1;
 4.     for ndx = 2:length(worldData)
 5.         cntry = worldData(ndx);
 6.         tryThis = fold(cntry);
 7.         if tryThis > best
 8.             best = tryThis;
 9.             besti = ndx
10.         end
11.     end
12. end
13. function ans = fold(st)
    % s1 is the rate of growth of population
14.     pop = st.pop(~isnan(st.pop));
15.     yr = st.year(~isnan(st.pop));
16.     s1 = slope(yr, pop)/mean(pop);
        % s2 is the rate of growth of the GDP
17.     gdp = st.gdp(~isnan(st.gdp));
18.     yr = st.year(~isnan(st.gdp));
19.     s2 = slope(yr, gdp)/mean(gdp);
        % Measure of merit is how much faster
        % the gdp grows than the population
20.     ans = s2 - s1;
21. end
22. function sl = slope(x, y)
    % Estimate the slope of a curve
23.     if length(x) == 0 || x(end) == x(1)
24.         error('bad data')
25.     else
26.         sl = (y(end) - y(1))/(x(end) - x(1));
27.     end
28. end
```

Listing 10.3 shows the function that finds the best country by folding the country structure array, together with the two supporting functions that provide the comparison criteria. Notice that the complexity of the data has forced the solution into nested folds: to fold the country data array, we have to summarize (fold) the annual data for each country.

In Listing 10.3:

Lines 2 and 3: As with any folding function that is looking for the maximum or minimum of a collection, the best place to start is the first item in the collection. The remaining items can then be compared to this one.

Lines 4–11: Loop through the remaining countries in the array.

Line 5: Extracts one structure.

Line 6: Computes its friendliness value.

Lines 7–10: If the result is improved, these lines update the stored values. The index `besti` is returned when the loop finishes.

Line 13: This function computes the measure of friendliness for each country. The goal is to subtract the rate of population growth from the rate of growth of the GDP. So first we compute the rate of population growth.

Lines 14 and 15: These lines establish two local vectors containing the population value and the corresponding year without the values that are `NaN`, the places where `"na"` appears in the spreadsheet.

Line 16: Calls the helper function for the slope of this relationship, and non-dimensionalizes the result by dividing by the mean population.

Lines 17–19: Repeat the same logic for the non-dimensional rate of increase of the GDP.

Line 20: Returns the difference in growth rates.

Line 22: The function that estimates the rates of growth.

Lines 23 and 24: We have a problem if there is no data or if the value we will subsequently use as a divisor is zero.

Line 26: A very crude measure of the slope is to divide the difference between the first and last data points by the difference between the first and last `x` values. (We will be able to improve on this approach later.)

When we run this program, we see the following result:

```
>> best country is Equatorial Guinea
```

This may not be exactly the result we were hoping for. In Chapter 16 we will revisit this example with some better tools that will allow us to apply additional criteria to selecting countries.

Chapter Summary

This chapter presented the fundamental operations that can be applied to problem solving:

- Using normal arithmetic operations with specific input and output values
- Inserting new elements in a collection
- Traversing a collection
- Building a collection by repetitive insertion
- Mapping a collection—changing the values of the data items in the collection, but not the number of them
- Filtering a collection—reducing the number of entries, but not changing the data contents of the collection
- Folding—summarizing the values in a collection into a single quantity
- Searching for a specific match in a collection
- Sorting a collection

Then we briefly discussed how to combine these fundamental tools to solve more complex data manipulation problems.

Self Test

Use the following questions to check your understanding of the material in this chapter:

True or False

1. Copying the elements of a structure array into a cell array is a combination of traversal and insertion.
2. If you map a collection, you must change at least one of its elements.
3. When you filter a collection, at least one data element is changed.
4. The function `max(...)` is not folding because it returns two values.
5. You can use a `for` loop to search a collection even if you need to stop the search when you find the answer.
6. Sorting must involve putting the items in a collection in numerical order (ascending or descending).

Fill in the Blanks

1. The problem-solving style recommended in this text is to identify the _____and the _____.
2. Building is usually the process of _____ and _____ _____.

3. Mapping may involve combining _____ of the same length.
4. We _____ vectors by applying built-in logical operations and then indexing with the results to produce new, shorter arrays.
5. Totaling, averaging, and finding the smallest element in a vector are typical examples of _____.
6. There are almost always two exit criteria for a search: _____ _____ or _____.
7. To save a collection to a text file, you _____ the collection _____ it to the file.

Programming Projects

1. The purpose of this problem is to write a set of functions that calculate the volume of a slant cylinder with an irregular pentagonal cross section shown in Figure 10.10.

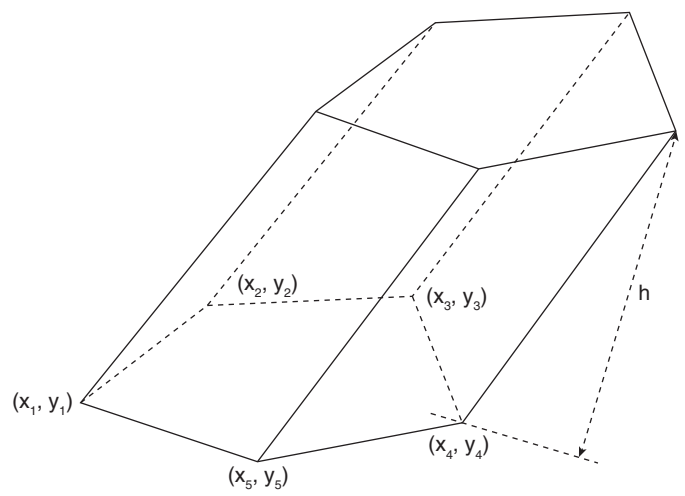

Figure 10.10 *The Slant Cylinder*

You will be given two vectors, `x` and `y`, containing the coordinates of the corners of the pentagon, and the value `h`, the vertical height of the cylinder. We will need to break this problem apart, writing functions to solve each part:

a. The volume of the cylinder is the area of the pentagon multiplied by the vertical height; write a function `polyvol(x, y, h)` to solve this.
b. The area of the pentagon is the sum of the areas of three triangles shown in Figure 10.11. So we need to write a function `pent_area(x, y)` that asks for the area of the three triangles and adds them together.

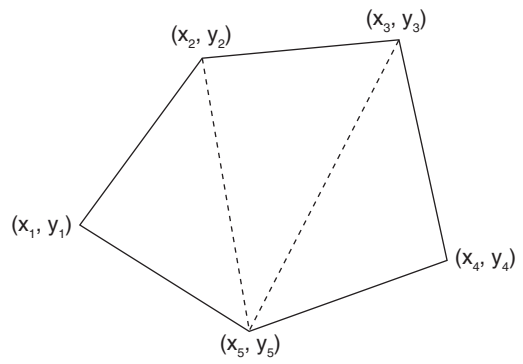

Figure 10.11 *Break down the pentagon*

c. Given the coordinates of the corners of a triangle, we need a function `tri_area(x, y)` to calculate the area of the triangle—see Figure 10.12. To compute the area of the triangle, we need the values of `a`, `b`, and `c`. So if we had the lengths of the lines, the area of the triangle is given by Heron's formula:

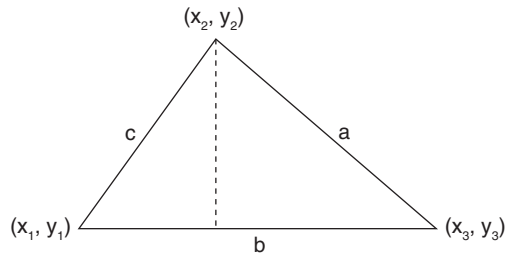

Figure 10.12 *Area of a triangle*

```
A = ( s(s-a)(s-b)(s-c) )
```

where `s` is half the sum of `a`, `b`, and `c`

d. So we need a function `tri_side(x, y)` that computes the length of a line when given its end points.
e. Then, we can put the pieces back together by calling the functions with the right parameters, and then build and test `polyvol` using the test cases provided.

2. This problem is about processing structure arrays. Write a function named `structSort` that sorts a structure array based on a given field that contains numerical values. Your function should take in a structure array and a string that should correspond to one of the fields of the structure array and return the original structure array sorted on the given field. It should check to be sure that the specified field name is in fact one of the fields of the structure array, and call the `error(...)` function if it is not.

   Test your function by using the `buildCDs` script from Chapter 7, using the input function to specify the sorting field.