Chapter 5

End of Chapter Exercises

1. *What values do the following assignment statements place in the variable result?*

```
a) result ← 3 + 4 ;
b) result  ← result + 7 ;
c) result  ← result ;
d) result ← result - 7 ;
```

a) 7 – the expression 3+4 gives 7.
b) 14 – `result` was already 7 from the last assignment so adding another 7 takes it to 14
c) 14 – we just assigned the value of `result` to itself.
d) 7 – now we have just subtracted 7 from `result`.

2. *A fairly common programming task is to swap the values of two variables. Because a variable can only hold one value at a time, a common solution is to introduce a temporary variable. For example, assume variable* `a` *has the value 7 and variable* `b` *the value 4. The following algorithm swaps their values via the intermediate variable temp:*

```
1. temp ← a ;
2. a ← b ;
3. b ← temp ;
```

All well and good, but suppose you didn't want to use `temp`*? Can you find a way to swap the values using only* `a` *and* `b` *in the solution?*

This algorithm will swap any two variables without the use of temporary storage. Try it out for yourself.

```
1. a ← a + b ;
2. b ← a - b ;
3. a ← a - b ;
```

3. *Professor Henry Higgins wants to attend a conference in Lincoln, Nebraska in the United States. The conference is being held in June and Professor Higgins wants to know what sort of clothing to take. Coming from northern Europe where temperatures are measured in degrees Celsius, Professor Higgins is unfamiliar with the Fahrenheit scale used in the United States. Therefore, he wants you to design an algorithm that will first convert a temperature in clothing based on the result. If the temperature in Lincoln in June is 24* *simple formula to convert from Fahrenheit to Celsius:* $C = (F - 32)\frac{5}{9}$

Thus, a temperature of 98 $= (98 - 32)\frac{5}{9} = 33.67$

Try your algorithm out with the following Fahrenheit temperatures: 57, 65, 72, 76, and 85

```
1.  celsius ← (fahrenheit – 32) × (5 ÷ 9) ;
2.  IF (celsius greater than or equal to 24)
       2.1. Display 'Take summer clothes' ;
    ENDIF
3.  IF (celsius less than 24) AND (celsius greater than 15)
       3.1. Display 'Take spring clothes' ;
    ENDIF
4.  IF (celsius less than or equal to 15)
       4.1. Display 'Take winter clothes' ;
    ENDIF
```

57F = 13.8 C = Winter clothes.

65 F = 18.3 C = Spring clothes.

72F = 22.2C = Spring clothes.

76F = 24.4C = Summer clothes.

85F = 29.4C = Summer clothes.

4. *Using the HTTLAP strategy design an algorithm that uses repeated subtraction to find out how many times one number divides another. For example, 20 divides 5 four times, 36 divides 7 five times (with a remainder of 1).*

```
1. Get firstNumber ;
2. Get secondNumber ;
3. counter ← 0 ;
4. remainder ← firstNumber ;
5. WHILE (remainder greater than or equal to secondNumber)
      5.1.  remainder ← remainder – secondNumber ;
      5.2.  counter ← counter + 1 ;
   ENDWHILE
6. Display firstNumber divides secondNumber by counter times ;
```

5. *We are used to seeing times in the form hh:mm:ss. If a marathon race begins at*

10:00:00 and the winner finishes at 14:07:41 we can see that the winning time was 4 hours, 7 minutes and 41 seconds. Design an algorithm that takes the start and finishing times of a runner and displays the time it took them to complete the race. If your solution worked using the start time of 10:00:00, does it still work for a start time of, say, 10:59:57?

What do we know? We know there are two times (the start and the finish times) and that the finish time must be greater than the start time. Sometimes the elapsed time is easy to spot. If the start time is 10:00:00 and the finish time is 14:07:41 then the race time is obviously 4:07:41. That suggests all we have to do is subtract the start hours from the finish hours, the start minutes from the finish minutes and the start seconds from the finish seconds. But that is not going to work in all cases. What about the start time of 10:59:57? How long did the race take if the finish time was still 14:07:41? If we just take the naive approach I just suggested we get 4 hours, -52 minutes, and -10 seconds! How to get round this? Take the start time of 1 minute 59 seconds, 1:59 and the finish time of 2:07. Now it's easy to see the elapsed time -- it's just 8 seconds. If we subtract the start seconds from the finish seconds we get -52. The difference between 52 seconds and 60 seconds (i.e. 1 minute) is 8 seconds. So, we cannot treat the seconds separately from the minutes or the minutes separately from the seconds. I think we need to put them all together into one number. Actually this is quite easy to do. What if we convert **everything** into seconds? 1 minute = 60 seconds. 1 hour = 60 minutes = 60 × 60 = 3600 seconds. So, the start time of 10:00:00 is 10 * 3600 seconds = 36,000 seconds. The finish time of 14:07:41 is 14 × 3600 + 7 × 60 + 41 seconds = 50400 + 420 + 41 = 50,861 seconds. Now, if we subtract the start time from the finish time we get the elapsed time of 50861−36000 = 14,861 seconds. But how to get this back to a time we can deal with? We just need to reverse the process. The number of hours in 14861 is given by the number of times 3600 divides 14861. 14861 ÷ 3600 = 4.13. So, the number of hours is 4. Now, if we take the remainder after dividing by 3600 we can repeat the process for the minutes. 14861 ÷ 3600 = 4, remainder 461. The number of minutes is, therefore, 461 ÷ 60 = 7, remainder 41. So, the elapsed time is 4:07:41 which is what we got before.

Lets apply this to a start time of 10:59:57 and a finish time of 14:07:41. Start time in seconds = (10 × 3600) + (59 × 60) + 57 = 39,597 seconds. The finish time is 50,400 seconds, so the elapsed time is 50861 − 39597 = 11,264 seconds.

Elapsed hours = 11264 ÷ 3600 = 3, remainder 464.
Elapsed minutes = 464 ÷ 60 = 7, remainder 44
Elapsed seconds = 44.
Elapsed time = 3:07:44. Check this out by adding 44 seconds to 10:59:57 = 11:00:41. Now add the 7 minutes = 11:07:41. Now add the 3 hours = 14:07:41. Now all we have to do is write these stages down as an algorithm.

```
1.   startTimeInSeconds ← startHours × 3600 + startMinutes × 60
```

```
+startSeconds ;
2.   finishTimeInSeconds ← finishHours × 3600 + finishMinutes × 60
+finishtSeconds ;
3.   elapsedTime ← finishTimeInSeconds – startTimeInSeconds ;
4.   elapsedHours ← elapsedTime ÷ 3600 ;
5.   remainder ← elapsedTime – (elapsedHours × 3600) ;
6.   elapsedMinutes ← remainder ÷ 60 ;
7.   elaspedSeconds ← remainder – (elapsedMinutes × 60) ;
8.   Display elapsedHours:elapsedMinutes:elapsedSeconds ;
```

Note, I am assuming that statements 4 and 6 will give me a whole number rather than a fractional number. We will see in Chapter 6 that whole and fractional numbers are treated differently in programming and in the StockSnackz project in chapter 6 we will find a special operator that gives us the remainder after division which will make statements 5 and 7 easier to write.

6. It is usual, when working with times, to convert real-world timings into a figure in seconds as this makes comparisons between two times much simpler. Given that there are 3600 seconds in an hour, write an algorithm that takes the start and finish times of a marathon runner in the form hh:mm:ss, converts them to a time in seconds, subtracts the start seconds from the finish seconds to give an elapsed time, and then converts this elapsed time in seconds back into a real world time of the form hh:mm:ss for display. For example, say I started a marathon at 09:05:45 and finished at 13:01:06, my start time would be 09 × 3600 + 05 × 60 + 45 = 32,400 + 300 + 45 = 32,745 seconds. My finish time would be 46,800 + 60 + 06 = 46,866 seconds. My race time would then be 46,866 - 32,745 = 14,121 seconds. This comes to 03:55:21. The real problem here is how did I get 03:55:21 from 14,121? The solution is related to the change-giving problem from Chapter 3.

Hmm, seems we've already solved this problem for exercise 5. Still, it's nice to see the solution I came up with being validated here.

7. In the solutions to the coffee-making problem we have been careful to give precise instructions for how to add the required number of sugars to each cup. If you look closely you may spot that we have not been as careful as we might have been with the rest of the solution. I am referring specifically to the instruction ‘measure coffee for number required’ (Solution 5.8). If you think about it you should realize that this action too needs more detailed instructions. How exactly do you measure the right amount of coffee? How many spoons (or scoops) of coffee do you need? Assume a standard measure of one dessert spoon for each cup of coffee required. Rewrite the task to make the instructions precise. Describe any variables that you need to add to the variable list.

Here's solution 5.8:

```
1.  Find out how many cups are required ;
```

```
2.  IF (more than zero cups wanted)
       2.1.  IF (more than six cups wanted)
                2.1.1  limit cups required to six ;
             ENDIF
       2.2.  Put water for cups required in coffee machine ;
       2.3.  Open coffee holder ;
       2.4.  Put filter paper in machine ;
       2.5.  Measure coffee for number required ;
       2.6.  Put coffee into filter paper ;
       2.7.  Shut the coffee holder ;
       2.8.  Turn on machine ;
       2.9.  Wait for coffee to filter through ;
             // Process the cups of coffee
       2.10. WHILE (cups poured not equal to cups required)
             // Add  sugar and milk as necessary
                2.10.1.  Find out how many sugars required ;
                2.10.2.  Find out whether milk required ;
                2.10.3.  WHILE (sugars added not equal to required)
                            2.10.3.1.  Add spoonful of sugar ;
                            2.10.3.2.  Add 1 to number of sugars added ;
                         ENDWHILE
                2.10.4.  IF (white coffee required)
                            2.10.4.1.  Add milk/cream ;
                         ENDIF
                2.10.5.  Pour coffee into mug ;
                2.10.6.  Stir coffee ;
                2.10.7.  Add 1 to number of cups poured
             ENDWHILE
       2.11. Turn off machine ;
    ENDIF
```

Statement 2.5 is the one we are focusing on. What do we need? We need a variable to hold the number of cups required, but that's taken care of in statement 1. We then need to repeatedly add a spoon of coffee until we have added as many spoonfuls as there are cups required. This is very similar to the sugar-adding problem (see statement 2.10.3), so I guess we can copy that.

```
2.5.  WHILE (coffee spoons added not equal to cups required)
     2.5.1. Add spoon of coffee ;
     2.5.2. Add 1 to number of coffee spoons added ;
  ENDWHILE
```

8. *I said it would be hard enough to try and solve the problem on the assumption that the coffee pot is large enough for the required coffee and that I would come back to the more general problem of making coffee for any number of people. You are to tackle that problem now. The capacity of the coffee pot is eight cups. Try to extend the coffee making solution you produced for exercise 7 to deal with the requirement of being able to make more than eight cups of coffee.*

The problem has changed now, so that we do not have a coffee machine of infinite size. We know the machine can make 8 cups. If we wanted 9 cups then we would have to make 2 pots (we will not worry whether that should be 1 pot of 8 cups followed by 1 pot of 1 cup or 1 pot of 5 cups followed by 1 pot of 4 cups). So, a new thing to find out is how many pots are required. For each pot required we will have to make a number of cups, so we can see there will be two nested loops: the outer one counts the pots and the inner one counts the cups. Statements 2.2 through 2.10 (and its sub statements) deal with making a single pot, so all we need to do is wrap that lot up inside an outer loop to deal with multiple pots. Also, we should remove the 6 cup limit imposed in statement 2.1: that was only there to get round the infinitely-sized coffee pot problem. Here's the above solution with the new statement for measuring the coffee incorporated and the 6-cup limit removed:

```
1.  Find out how many cups are required ;
2.  IF (more than zero cups wanted)
      2.1.  Put water for cups required in coffee machine ;
      2.2.  Open coffee holder ;
      2.3.  Put filter paper in machine ;
     2.4.     WHILE (coffee spoons added not equal to cups required)
           2.4.1. Add spoon of coffee ;
           2.4.2.     Add 1 to number of coffee spoons added ;
        ENDWHILE
      2.5.  Put coffee into filter paper ;
      2.6.  Shut the coffee holder ;
      2.7.  Turn on machine ;
      2.8.  Wait for coffee to filter through ;
            // Process the cups of coffee
      2.9. WHILE (cups poured not equal to cups required)
            // Add  sugar and milk as necessary
               2.9.1.  Find out how many sugars required ;
               2.9.2.  Find out whether milk required ;
               2.9.3.  WHILE (sugars added not equal to required)
                           2.9.3.1.  Add spoonful of sugar ;
                           2.9.3.2.  Add 1 to number of sugars added ;
                        ENDWHILE
               2.9.4.  IF (white coffee required)
                           2.9.4.1.  Add milk/cream ;
                        ENDIF
               2.9.5.  Pour coffee into mug ;
               2.9.6.  Stir coffee ;
               2.9.7.  Add 1 to number of cups poured
            ENDWHILE
      2.10. Turn off machine ;
    ENDIF
```

Now we just need to add the new outer loop. We need to be careful though as statements 2.1, 2.4, and 2.9 are all stated in terms of a single pot. If we are making 9 cups then we need to execute this block of statements twice: the first time for 8 cups and the second time for 1 cup. However, if we just stick in an outer loop, these inner statements 2.1, 2.4, and 2.9 will try to deal with 9 cups **each time** which is clearly not what we want. What we need to do is first to calculate how many pots are needed and then for each pot we make, calculate the number of cups required in **that pot**. Expressing how to deal with the various numbers required can be done now but will be much easier to do once we have learned how to deal with remainders cleanly (chapter 6) and how to deal with multiple conditions (also chapter 6). For now, let's deal with this problem at the structural level only and leave the abstraction level quite high:

```
1.  Find out how many cups are required ;
2.  IF (more than zero cups wanted)
          2.1 WHILE (pots to make)
                2.1.1.  Put water for cups required for this pot in coffee machine ;
         2.1.2.  Open coffee holder ;
         2.1.3.  Put filter paper in machine ;
         2.1.4.     WHILE (coffee spoons added not equal to cups required for this pot)
                          2.1.4.1. Add spoon of coffee ;
                          2.1.4.2. Add 1 to number of coffee spoons added ;
                        ENDWHILE
         2.1.5.  Put coffee into filter paper ;
         2.1.6.  Shut the coffee holder ;
         2.1.7.  Turn on machine ;
         2.1.8.  Wait for coffee to filter through ;
                 // Process the cups of coffee
         2.1.9.  WHILE (cups poured not equal to cups required for this pot)
                   // Add  sugar and milk as necessary
                    2.1.9.1.  Find out how many sugars required ;
                    2.1.9.2.  Find out whether milk required ;
                    2.1.9.3.  WHILE (sugars added not equal to required)
                                 2.1.9.3.1.  Add spoonful of sugar ;
                                 2.1.9.3.2.  Add 1 to number of sugars added ;
                              ENDWHILE
                    2.1.9.4.  IF (white coffee required)
                                 2.1.9.4.1.  Add milk/cream ;
                              ENDIF
                    2.1.9.5.  Pour coffee into mug ;
                    2.1.9.6.  Stir coffee ;
                    2.1.9.7.  Add 1 to number of cups poured
                 ENDWHILE
         2.1.10. Turn off machine ;
       ENDWHILE
    ENDIF
```

If you are interested in seeing how to deal with the 'for this pot' bit, then read on.

We have variables to keep track of how many cups are required, and how many have been poured. Lets add some more: `potsRequired` and `potsMade` to deal with the pots of coffee. Also I will add `cupsToMake` to hold the number of cups needed for each pot and finally `cupsMade` to store how many cups we have made so far. Here's the new solution incorporating these variables. The new sections are explained after the pseudo-code listing.

```
1.  Find out how many cups are required ;
2.  IF (more than zero cups wanted)
       2.1 cupsMade ← 0 ;
       2.2. potsRequired ← cupsRequired ÷ 8 ;

           2.3   IF (cupsRequired - (potsRequired × 8) not equal to 0
             2.3.1. potsRequired ← potsRequired + 1 ;
       //Alterntive statement 2.3. using the MOD operator -- see StockSnackz project on p.160
       2.3  IF (cupsRequired MOD 8 not equal to 0)
             2.3.1. potsRequired ← potsRequired + 1 ;

              2.4. WHILE (potsRequired greater than 0)
               // Calculate cups needed for this pot
               2.4.1.   cupsPoured ← 0 ;
               2.4.2.   IF (potsRequired greater than 1)
                           2.4.2.1. cupsToMake ← 8 ;
                        ENDIF
               2.4.3.   IF (potsRequired  equals 1)
                           2.4.3.1. cupsToMake ← cupsRequired - cupsMade ;
                        ENDIF

                         // Make the coffee
               2.4.4.   Put water for cupsToMake in coffee machine ;
               2.4.5.   Open coffee holder ;
               2.4.6.   Put filter paper in machine ;
                    2.4.7.   spoonsAdded ← 0 ;
               2.4.8.   WHILE (coffee spoons added not equal to cupsToMake)
                                  2.4.6.1.    Add spoon of coffee ;
                                  2.4.6.2.    spoonsAdded ← spoonsAdded + 1 ;
                                ENDWHILE
               2.4.9.   Put coffee into filter paper ;
               2.4.10.  Shut the coffee holder ;
               2.4.11.  Turn on machine ;
               2.4.12.  Wait for coffee to filter through ;
                        // Process the cups of coffee
               2.4.13.  WHILE (cupsPoured less than cupsToMake)
                              // Add  sugar and milk as necessary
                           2.4.13.1.  Find out how many sugars required ;
                           2.4.13.2.  Find out whether milk required ;
                           2.4.13.3.  WHILE (sugars added not equal to required)
                                         2.4.13.3.1.  Add spoonful of sugar ;
                                         2.4.13.3.2.  Add 1 to number of sugars added ;
                                      ENDWHILE
                           2.4.13.4.  IF (white coffee required)
                                         2.4.13.4.1.  Add milk/cream ;
                                      ENDIF
                           2.4.13.5.  Pour coffee into mug ;
                           2.4.13.6.  Stir coffee ;
                           2.4.13.7.  cupsPoured ← cupsPoured + 1 ;
                           2.4.13.8.  cupsMade ← cupsMade + 1 ;
                        ENDWHILE
               2.4.14. Turn off machine ;
               2.4.15. potsRequired ← potsRequired - 1 ;
          ENDWHILE
    ENDIF
```

Notes

Statement 2.3 finds out how many pots to make. If there are 1 to 8 cups required then we only need 1 pot. If we have 9 or more coffees required then we need more than 1 pot. We know the number of pots is related to multiples of 8: 8 cups needs 1 pot, 16 cups needs 2 pots and so on. If we divide `cupsRequired` by 8 we get part way there. If `cupsRequired` equals 8 then dividing it by 8 gives us 1 pot; if it's 16 then it gives us 2 pots: so far so good. However, what if `cupsRequired` is 7? Dividing it by 8 gives 0. But clearly we need 1 pot. So, statement 2.3 deals with these situations. First, begin by dividing `cupsRequired` by 8 and assigning the result to `potsRequired`. Then, if the remainder after dividing `cupsRequired` by 8 is not zero, then add 1 to `potsRequired`. In the case of `cupsRequired` being exactly 8, then there is no left over, so the number of `potsRequired` is 1. However, when `cupsRequired` is, say, 7, then the first assignment would give `potsRequired` the value 0: 8 goes into 7 zero times. But the remainder after dividing by 8 is 7, so we know we do need to make coffee. The same works for, say, 17 cups. 17 ÷ 8 = 2, so we know we need at least 2 pots. But there is a left over of 1 (8 goes into 17 twice with a remainder of 1) , so we need a third pot to deal with the seventeenth cup.

Statement 2.4.1 sets the number of `cupsPoured` for **this pot** to zero. Statements 2.4.2 and 2.4.3 find out how many cups to make for this pot. If the number of pots still to make is greater than 1 then we know we can make 8 cups -- the capacity of the pot. But, if this is the last pot, then we need to make however many cups are left to make which is given by subtracting the number of cups we have made so far from the number originally required. If you read chapter 6 you will see we can tidy this statement up as follows:

```
IF (potsRequired > 1)
  cupsToMake ← 8 ;
ELSE
  cupsToMake ← cupsRequired - cupsMade ;
ENDIF
```

9. Paul's Premier Parcels *was so pleased with your work that they want you to extend your van loading solution to incorporate some extra requirements. In addition to the existing reports the manager wants to know the weight of the lightest payload that was sent out. He also wants to know the average payload of all the despatched vans. Using PftB draw up solutions to these additional problems and try to incorporate them into Solution 5.18. Think about what extra variables may be needed and how they should be initialized and their values calculated. An average of a set of values is calculated by dividing the sum of all the values by the number of values in the set. Thus, the average of 120, 130, and 140 is (120 + 130 + 140) ÷ 3 = 130.*

```
// ******************************************
// Instructions for loading vans
// Written by Paul Vickers, June 2007
// ******************************************
// Initialize variables
1.  capacity ← 750 ;
2.  numberOfVans ← zero ;
3.  heaviestVan ← zero ;
4.  lightestPayload ← 751 ;
5.  totalPayload ← zero ;
6.  averagePayload ;
7.  Get first parcelWeight ;
8.  WHILE (conveyor not empty)
    // Process vans
      8.1.  payload  ←  zero ;
      8.2.  WHILE (payload + parcelWeight less than or equal to capacity)
AND (conveyor NOT empty)
      // Load a single van
               8.2.1.  Load parcel on van ;
               8.2.2.  payload ← payload + parcelWeight ;
               8.2.3.  Get next parcelWeight ;
            ENDWHILE
      8.3.  Despatch van ;
      8.4.  numberOfVans ← numberOfVans + 1 ;
      // Check whether this is the heaviest van
      8.5.  IF (payload more than heaviestVan)
               8.5.1.  heaviestVan ← payload ;
            ENDIF
      // Check if this is the lightest van
      8.6.  IF (payload less than lighestPaylod)
               8.6.1.  lightestPayload ← payload ;
            ENDIF
      //Add payload to total payload
      8.7.  totalPayload ← totalPayload + payload ;
    ENDWHILE
    // Calculate average payload ;
9.  IF (totalPayload NOT equal to zero) // Can't divide by zero
       9.1.  averagePayload ← totalPayload ÷ numberOfVans ;
    ENDIF
10. Report numberOfVans used ;
11. Report heaviestVan sent ;
12. Report lighestPayload sent out ;
13. Report averagePayload sent out ;
```

10. *In 2006 Fifa (the world governing body for international football) revised the way teams are allocated points for international football matches as shown in Table 5.5. Thus in a normal match the winning team gets 3 points with the losers getting zero. If a match is won through a penalty shootout the winning team only gets 2 points and the losers get 1 point. Both teams get 1 point each for a draw.*

Result	***Points***
Win	*3*
Draw	*1*
Lose	*0*
Win (with penalty shootout)	*2*
Lose (with penalty shootout)	*1*

Design an algorithm that analyses the result of a match and awards points to the two teams accordingly. For example, given the following score line:
`England 2 : Brazil 0, Penalties: No`
England would be awarded 3 points and Brazil 0 points. For this match:
`Netherlands 5: Portugal 4, Penalties: Yes`
The Netherlands would get 2 points and Portugal 1 point.

Now extend your solution to deal with an unspecified number of matches. The algorithm should stop calculating points when the following score line appears that has a negative goal score for either team. Teams may play more than one match, so each team's points tally will need to be updated as necessary. At the end, the team with the highest points tally should be declared top of the world rankings.

What do we know? There are two teams each of which will have scored a number of goals (possibly zero). The team that wins gets 3 points, the loser getting 0 points. If they score the same number of goals then the match is a draw and each team gets 1 point. However, if the game goes into a penalty shootout then the winner only gets 2 points with the loser getting 1 point. So, we can see there are two basic choices: somebody won the match, or it was a draw. In the case of a win there are two choices: it was won on penalties or it was won without penalties. I think this gives us enough to go on.

```
IF (team1Goals equals team2Goals) // A draw
   team1Points ← 1 ;
   team2Points ← 1 ;
ENDIF
IF (team1Goals not equal to team2Goals) // A win
   IF (penalties)
      IF (team1Goals greater than team2Goals)
         team1Points ← 2 ;
     teams2Points ← 1 ;
  ENDIF
  IF (team1Goals less than team2Goals)
     team1Points ← 1 ;
         team2Points ← 2 ;
      ENDIF
   ENDIF
   IF (not penalties)
      IF (team1Goals greater than team2Goals)
         team1Points ← 3 ;
     teams2Points ← 0 ;
  ENDIF
  IF (team1Goals less than team2Goals)
     team1Points ← 0 ;
         team2Points ← 3 ;
      ENDIF
   ENDIF
ENDIF
```

Using the extended notation from Chapter 6 I can simplify the above algorithm thus:

```
IF (team1Goals equals team2Goals) // A draw
   team1Points ← 1 ;
   team2Points ← 1 ;
ELSE // A win
   IF (penalties)
      IF (team1Goals greater than team2Goals)
         team1Points ← 2 ;
     teams2Points ← 1 ;
  ELSE
     team1Points ← 1 ;
         team2Points ← 2 ;
      ENDIF
   ELSE
      IF (team1Goals greater than team2Goals)
         team1Points ← 3 ;
     teams2Points ← 0 ;
  ELSE
     team1Points ← 0 ;
         team2Points ← 3 ;
      ENDIF
   ENDIF
ENDIF
```

The extended problem requires some knowledge of data structures to do completely and that is beyond the scope of this chapter. However, we can deal with the problem in an abstract way thus:

```
Get result ;
```

```
WHILE (neither team's score is negative)
   IF (team1Goals equals team2Goals) // A draw
      team1Points ← 1 ;
      team2Points ← 1 ;
   ENDIF
   IF (team1Goals not equal to team2Goals) // A win
      IF (penalties)
         IF (team1Goals greater than team2Goals)
            team1Points ← 2 ;
        teams2Points ← 1 ;
     ENDIF
     IF (team1Goals less than team2Goals)
        team1Points ← 1 ;
            team2Points ← 2 ;
         ENDIF
      ENDIF
      IF (not penalties)
         IF (team1Goals greater than team2Goals)
            team1Points ← 3 ;
        teams2Points ← 0 ;
     ENDIF
     IF (team1Goals less than team2Goals)
        team1Points ← 0 ;
            team2Points ← 3 ;
         ENDIF
      ENDIF
   ENDIF
ENDWHILE
Find team with highest points score ;
Display 'Team ' highest scoring team ' are world champions ;
```

11. *Vance Legstrong, a world-class Stocksfield-based cyclist has asked the Department of Mechanical Engineering at the University of Stocksfield to help him out with the selection of gear wheels on his new bicycle. Vance wants to know, for each of 10 gears on his bicycle how many times he must make one full turn of the pedals in order to travel 1 km. A bicycle's gear ratio is given as the ratio between the number of teeth on the chain wheel (the one the pedals are connected to) and the number of teeth on the gear wheel (the one attached to the rear wheel). The rear wheel has ten gear wheels all with a different number of teeth. The chain wheel has 36 teeth and the gear wheels have teeth as given in Table 5.6:*

Gear wheel	*Number of teeth*
1	*36*

2	*34*
3	*32*
4	*30*
5	*28*
6	*26*
7	*24*
8	*22*
9	*20*
10	*18*

Thus, the ratio of gear 1 is 1 as the chain wheel ÷ gear wheel = 36/36. The ratio of gear 5 is 1.2857, and so on. The diameter of the wheels on Vance's bicycle is 27 inches (bicycle parts are commonly measured in imperial rather than metric units).

a) Design an algorithm to calculate the gear ratios of all ten gears.
b) Extend your solution to tell Vance how many pedal turns are needed to travel 1 km in each gear. You may assume that no freewheeling is allowed. Think about how far the bicycle will travel for each full turn of the pedals.

A sketch of the principal components may be useful. You may like to tackle some of sub-problems separately.

a) Gear ratios. What do we know? We know the ratio of a gear wheel = 36 ÷ number of teeth on the wheel. We can also see a relationship between the number of teeth and the wheel number: the number of teeth goes down by 2 with every increase in gear number. So, if we start with an initial value of 1 for a variable gearNumber and a value of 36 for gearTeeth all we need is a simple loop to process each gear in turn:

```
gearNumber ← 1 ;
gearTeeth ← 36 ;
WHILE (gearNumber less than or equal to 10)
   ratio ← 36 ÷ gearTeeth ;
   Display (gearNumber ' has ratio ' ratio) ;
   gearNumber ← gearNumber + 1 ;
   gearTeeth ← gearTeeth – 2 ;
ENDWHILE
```

b) Distance travelled. We were given the following hint: One of the sub-problems is finding out how far one turn of the pedals will move the bicycle. In gear 1, turning the pedals one full turn will rotate the rear wheel one full turn as the gear ratio is 1:1, thus the bicycle will travel 27 inches; In gear 5, the rear wheel will make 1.2857 full turns. Another sub-problem is finding out how many times the pedal has to turn to move 1 km: how many inches are there in 1 km? You can start off by knowing that 1 inch = 2.54 cm and there are 100 cm in 1 m and 1000 m in 1 km.***.

The distance travelled per pedal turn is related to the ratio of the gear. The algorithm for part (a) calculates the ratio, so all we need to do is extend this to calculate the number of pedal turns. In gear 1, each pedal turn takes Vance 27 inches as the ratio is 1.0. 27 inches in cm is 27 × 2.54 = 68.58 cm. 1 km = 100,000 cm, so in gear 1 we need to turn the pedal 100,000 ÷ 68.58 = 1,458.15 times! So, the procedure is thus: Calculate the ratio, calculate how far 1 pedal turn will take us with this ratio, convert this distance in inches to cm and then divide this into 100,000.

```
gearNumber ← 1 ;
gearTeeth ← 36 ;
WHILE (gearNumber less than or equal to 10)
   ratio ← 36 ÷ gearTeeth ;
   distanceInInches ← ratio × 27 ;
   distanceInCM ← distanceInInches × 2.54 ;
   numberPedalTurns ← 100000 ÷ distanceInCM ;
   Display (gearNumber ' has ratio ' ratio) ;
   Display ('You need ' numberPedalTurns ' to travel 1 km) ;
   gearNumber ← gearNumber + 1 ;
   gearTeeth ← gearTeeth – 2 ;
ENDWHILE
```

Projects

StockSnackz Vending Machine

Up till now we assumed the vending machine had unlimited supplies. It is time to put away such childish notions. Therefore, extend your solution so that the machine now shows a "sold out" message if it has run out has run out of a selected item. For testing purposes set the initial stock levels to 5 of each item (otherwise it will take your friend a long time to work through the algorithm!).

Previously if the buttons 0, 7, 8, 9 were pressed the machine simply did nothing. Now it is time to design a more typical response. The machine should behave as before when buttons 1–6 are pressed but should now show an "Invalid choice" message if the buttons 0, 7, 8, 9 are pressed. For both problems remember to write down all the variables (together with their ranges of values) that are needed.

Identifier	**Description**	**Range of values**
`chocolateStock`	Stock level for chocolate bars	{1..20}
`muesliStock`	Stock level for muesli bars	{1..20}
`cheesePuffStock`	Stock level for chees puffs	{1..20}

appleStock	Stock level for apples	{1..20}
popcornStock	Stock level for popcorn	{1..20}

```
1. Install machine ;
2. Turn on power ;
3. Fill machine ;
4. chocolateStock  5 ;
5. muesliStock  5 ;
6. cheesePuffStock  5 ;
7. appleStock  5 ;
8. popcornStock  5 ;
9. WHILE (not the end of the day)
   9.1 IF (button 1 pressed)
          IF (chocolateStock > 0)
             Dispense milk chocolate ;
             chocolateStock  chocolateStock - 1 ;
          ENDIF
          IF (chocolateStock = 0)
             Display 'Sold out message' ;
          ENDIF
       ENDIF
   4.2 IF (button 2 pressed)
          IF (muesliStock > 0)
             Dispense muesli bar ;
             muesliStock  muesliStock - 1 ;
          ENDIF
          IF (muesliStock = 0)
             Display 'Sold out message' ;
          ENDIF
       ENDIF
   4.3 IF (button 3 pressed)
          IF (cheesePuffStock > 0)
             Dispense cheese puffs ;
             cheesePuffStock  cheesePuffStock - 1 ;
          ENDIF
          IF (cheesePuffStock = 0)
             Display 'Sold out message' ;
          ENDIF
       ENDIF
   4.4 IF (button 4 pressed)
          IF (appleStock > 0)
             Dispense apple ;
             appleStock  appleStock - 1 ;
          ENDIF
          IF (appleStock = 0)
             Display 'Sold out message' ;
          ENDIF
       ENDIF
   4.5 IF (button 5 pressed)
          IF (popcornStock >0)
             Dispense popcorn ;
             popcornStock  popcornStock - 1 ;
          ENDIF
          IF (popcornStock = 0)
             Display 'Sold out message' ;
          ENDIF
       ENDIF
   4.6 IF (button 6 pressed)
          Print sales summary ;
       ENDIF
   4.7 IF (button 0, 7, 8, 9 pressed)
          Display 'Invalid choice message' ;
       ENDIF
   ENDWHILE
```

Stocksfield Fire Service: Hazchem Signs

Look at the algorithm you created in Chapter 4 for the hazchem problem. Identify and write down all the variables you think you may need for this solution. Remember to also indicate the typical ranges of values that each variable can take.

Identifier	**Description**	**Range of values**
`fireFightingCode`	Fire fighting code	{1,2,3,4}
`precautionsCode`	Fire fighters' precautions	{P,R,S,T,W,X,Y,Z}
`publicHazardCode`	Public hazard	{V,blank}

Puzzle World: Roman Numerals and Chronograms

Look at the algorithm you created in Chapter 4 for the Roman numerals validation and translation problems. Identify and write down all the variables you think you may need for your solutions. Remember to also indicate the typical ranges of values that each variable can take.

Identifier	**Description**	**Range of values**
`romanDigit`	A single roman numeral	{I, V, X, L, C, D, M}
`subSequence`	a compound roman number	{IV, IX, XL, XC, CM}
`romanNumber`	a complete roman number	Many, e.g. I, LXI, MCMLXXIX, etc.

`digitValue`	value of a roman numeral	{1, 5, 10, 50, 100, 500,1000}
`compoundValue`	value of compound number	{4, 9, 50, 90, 900}

There are probably more, but I won't really know until we try a full worked up algorithm.

Pangrams: Holoalphabetic Sentences

Look at the algorithm you created in Chapter 4 for the pangram problem. Identify and write down all the variables you think you may need for this solution. Remember to also indicate the typical ranges of values that each variable can take.

Identifier	**Description**	**Range of values**
`currentLetter`	The letter we're currently working with	{'A' . .'Z'}
`numberCrossedOut`	The number of letters not appearing in the candidate sentence	{1 .. 26}

Online Bookstore: ISBNs

Identify likely variables and their ranges of values for:

a) the ISBN hyphenation problem
b) the ISBN validation problem

Write your solution to the ISBN validation problem using an iteration to work through the nine main digits of the ISBN.

a) Hyphenation problem

Identifier	**Description**	**Range of values**
`groupCode`	The publishing area code	{0, 1}
`publisherCode`	The code for a publisher	{0 .. 9999999}
`titleCode`	The book title code	{0 .. 999999}
`checkDigit`	The check digit	{0..9, X}

b) Validation problem

Identifier	**Description**	**Range of values**
`currentDigit`	The current digit to process	{0..9, X}
`counter`	loop counter	{0 .. 9}
`checkDigit`	The check digit	{0..9, X}
`calcCheck`	our computed check digit	{0..9, X}
`total`	Needed to compute checksum	{0.. approx 500}
`remainder`	Needed to compute checksum	{0..10}

ISBN	The ISBN itself	{9 characters 0..9 followed by one character 0..9,X}

```
1.   currentDigit ← first digit of ISBN ;
2.   counter ← 1 ;
3.   WHILE (counter less than or equal to 9)
        3.1. total ← currentDigit × (11 – counter)  ;
        3.2. currentDigit ← next digit of ISBN ;
    ENDWHILE
4. remainder ← total – (total ÷ 11) ; // or remainder ← total MOD 11 ; --
see p.167
5. IF (11 - remainder equals checkDigit)
        5.1. Display 'ISBN valid' ;
   ENDIF
```