**11.40 (*Guess the capitals*) Write a program that repeatedly prompts the user to enter a capital for a state. Upon receiving the user input, the program reports whether the answer is correct. Assume that 50 states and their capitals are stored in a two-dimensional list, as shown in Figure 11.13. The program prompts the user to answer all the states' capitals and displays the total correct count. The user's answer is not case sensitive. Implement the program using a list to represent the data in the following table.

Alabama	Montgomery
Alaska	Juneau
Arizona	Phoenix
...	...
...	...

FIGURE 11.13 A two-dimensional list stores states and their capitals.

Here is a sample run:

```
What is the capital of Alabama? Montogomery Enter
The correct answer should be Montgomery
What is the capital of Alaska? Juneau Enter
Your answer is correct
What is the capital of Arizona? ...
...
The correct count is 35
```

14.8 (*Display nonduplicate words in ascending order*) Write a program that prompts the user to enter a text file, reads words from the file, and displays all the nonduplicate words in ascending order.

*14.10 (*Guess the capitals*) Rewrite Exercise 11.40 using a dictionary to store the pairs of states and capitals so that the questions are randomly displayed.

***6.14** (*Estimate* π) π can be computed using the following series:

$$m(i) = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \ \ldots \ + \frac{(-1)^{i+1}}{2i - 1}\right)$$

Write a function that returns `m(i)` for a given `i` and write a test program that displays the following table:

```
i                m(i)

1                4.0000
101              3.1515
201              3.1466
301              3.1449
401              3.1441
501              3.1436
601              3.1433
701              3.1430
801              3.1428
901              3.1427
```

****6.25** (*Emirp*) An *emirp* (*prime* spelled backward) is a nonpalindromic prime number whose reversal is also a prime. For example, both `17` and `71` are prime numbers, so `17` and `71` are emirps. Write a program that displays the first 100 emirps. Display 10 numbers per line and align the numbers properly, as follows:

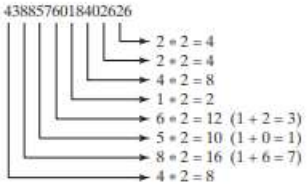
```
 13   17   31   37   71   73   79   97  107  113
149  157  167  179  199  311  337  347  359  389
...
```

**6.29 (*Financial: credit card number validation*) Credit card numbers follow certain patterns: It must have between 13 and 16 digits, and the number must start with:

- 4 for Visa cards
- 5 for MasterCard credit cards
- 37 for American Express cards
- 6 for Discover cards

In 1954, Hans Luhn of IBM proposed an algorithm for validating credit card numbers. The algorithm is useful to determine whether a card number is entered correctly or whether a credit card is scanned correctly by a scanner. Credit card numbers are generated following this validity check, commonly known as the *Luhn check* or the *Mod 10 check*, which can be described as follows (for illustration, consider the card number 4388576018402626):

1. Double every second digit from right to left. If doubling of a digit results in a two-digit number, add up the two digits to get a single-digit number.



2. Now add all single-digit numbers from Step 1.

$$4 + 4 + 8 + 2 + 3 + 1 + 7 + 8 = 37$$

3. Add all digits in the odd places from right to left in the card number.

$$6 + 6 + 0 + 8 + 0 + 7 + 8 + 3 = 38$$

4. Sum the results from Steps 2 and 3.

$$37 + 38 = 75$$

5. If the result from Step 4 is divisible by 10, the card number is valid; otherwise, it is invalid. For example, the number 4388576018402626 is invalid, but the number 4388576018410707 is valid.

Write a program that prompts the user to enter a credit card number as an integer. Display whether the number is valid or invalid. Design your program to use the following functions:

```
# Return true if the card number is valid
def isValid(number):

# Get the result from Step 2
def sumOfDoubleEvenPlace(number):

# Return this number if it is a single digit, otherwise, return
# the sum of the two digits
def getDigit(number):

# Return sum of odd place digits in number
def sumOfOddPlace(number):

# Return true if the digit d is a prefix for number
def prefixMatched(number, d):

# Return the number of digits in d
def getSize(d):

# Return the first k number of digits from number. If the
# number of digits in number is less than k, return number.
def getPrefix(number, k):
```