Section Solution: Python

Solution 1: Jane Austen's Favorite Word

Project Gutenberg is an open-source effort intended to legally distribute electronic copies of thousands of books no longer under copyright. So, if you're tired of coding in spite of the strong Internet connection, you might download yourself a little Franz Kafka by visiting:

`http://www.gutenberg.org/files/5200/5200.txt`

and e-flip through Metamorphosis to pass the time.

A quick scan of **`www.gutenberg.org`** reveals that virtually all of Jane Austen's novels—Emma, Lady Susan, Love and Friendship, Northanger Abbey, Persuasion, Price and Prejudice, and Sense and Sensibility—are available in plain text format. Naturally, this prompts you to wonder what everyone else is wondering: What's Jane Austen's favorite word?

Given an array of URLs, write a Python function **`getFavoriteWord`** that reads in all of the plain text documents addressed by the URLs, compiles a huge frequency map of all words she uses anywhere in any of the texts, and returns the most commonly occurring word. Only count a word if it's all lowercase letters, and assume the favorite word is the one with the highest score, where the score of a word is equal to $\mathbf{freq}^{\mathbf{len}}$, where **freq** is the word's frequency and **len** is the word's length. For simplicity, assume all punctuation marks are separated from real words by whitespace.

See the last page of the section handout for a list of all dictionary methods. (You'll want to focus on the **`get`** and **`operator[]`** methods.)

Sample Solution

```
#!/usr/bin/env python
#

from urllib2 import urlopen

# The urlopen function (in the urllib2 module--they tried
# real hard to come up with a good name from the second iteration
# of the urllib module, eh?) takes a url string (like "http://www.google.com")
# and returns a file object with some address attributes (like the 'code'
# attribute, which is the response code of the HTTP request).
#
# Note the use of the map built-in, and the use of a lambda function to
# generates a new sequence which is identical to the old one, save that
# all of the "\n" have been chomped off (That's what the rstrip--short
# for reverse-strip--method takes care of)
def pullLines(url):
   instream = urlopen(url)
```

```
   if instream.code != 200: return []
   lines = instream.readlines()
   lines = map(lambda line: line.rstrip("\n"), lines)
   instream.close()
   return lines

# Updates the map to include all of the individual words
# in the line.  The split methods takes a line and generates
# a list of all those words in the line that are separated
# from one another by whitespace.  We're *pretending* that
# punctuation marks hugging actual words isn't something we want to
# worry about, although in practice we would really have to worry about it. :)
# Note the use of the dictionary's get method, which allows you to return
# a default value whenever the key of interest isn't present.
def digestSingleLine(line, freqMap):
   words = line.split()
   for i in xrange(0, len(words)):
      if words[i].islower():
         freq = freqMap.get(words[i], 0) + 1
         freqMap[words[i]] = freq

# Trivial wrapper function
def digestLines(lines, freqMap):
   for i in xrange(0, len(lines)):
      digestSingleLine(lines[i], freqMap)

# Two subtasks: break down into lines, and then digest each line one by one
def digestText(url, freqMap):
   lines = pullLines(url)
   digestLines(lines, freqMap)

# Pulls all of the contents from each of the URLs in the
# 'urls' list, and returns what apppears to be the favorite
# word, where the metric for what favorite means is outlined above
def getFavoriteWord(urls):
   freqMap = {}
   for i in xrange(0, len(urls)):
      digestText(urls[i], freqMap)
   favorite = ""
   favoriteScore = 0
   for key in freqMap.keys():
      score = freqMap[key] ** len(key)
      if score > favoriteScore:
         favorite = key
         favoriteScore = score
   return favorite

# Top level expressions that get executed in sequence
favoriteWord = \
   getFavoriteWord(["http://www.gutenberg.org/dirs/etext98/pandp12.txt",
                    "http://www.gutenberg.org/dirs/etext98/lvfnd10.txt",
                    "http://www.gutenberg.org/dirs/etext97/lsusn11.txt",
                    "http://www.gutenberg.org/dirs/etext94/persu11.txt",
                    "http://www.gutenberg.org/dirs/etext94/nabby11.txt",
                    "http://www.gutenberg.org/dirs/etext94/mansf10.txt"])
print "It appears Jane Austen really likes the word \"%s\"." % favoriteWord
```

Solution 2: Maximizing Points

You're given an unlimited number of pebbles to distribute across an N x N game board (N drawn from [3, 15]), where each square on the board contains some positive point value between 10 and 99, inclusive. A 6 x 6 board might look like this:

33	**74**	**26**	**55**	**79**	**54**
67	**56**	**91**	**72**	**44**	**32**
44	**64**	**22**	**91**	**29**	**61**
12	**32**	**76**	**50**	**50**	**32**
81	**65**	**56**	**38**	**96**	**36**
38	**78**	**50**	**92**	**90**	**75**

The player distributes pebbles across the board so that:

- At most one pebble resides in any given square.
- No two pebbles are placed on adjacent squares. Two squares are considered adjacent if they are horizontal, vertical, or even diagonal neighbors. There's no board wrap, so 44 and 61 aren't neighbors. Neither are 33 and 75.

The goal is to maximize the number of points claimed by your placement of pebbles.

Write a program that reads in a sequence of boards from standard input and posts the maximum number of points attainable by an optimal pebble placement for each. Each board is expressed as a series of lines, where each line is a space-delimited series of numbers. A blank line marks the end of each board (including the last one.)

The better solution to this problem uses memoization to reduce the running time of the solution from something very exponential in running time to something noticeably less exponential (though still exponential.) Don't worry too much about the file reading portion of the solution—focus instead on the ideal distribution of pebbles and the Python code that helps to discover it. (See the last page of this handout for a list of all dictionary and mutable sequence methods.)

So, if the following were fed to standard input:

```
71 24 95 56 54
85 50 74 94 28
92 96 23 71 10
23 61 31 30 46
64 33 32 95 89

78 78 11 55 20 11
98 54 81 43 39 97
12 15 79 99 58 10
13 79 83 65 34 17
85 59 61 12 58 97
40 63 97 85 66 90

33 49 78 79 30 16 34 88 54 39 26
80 21 32 71 89 63 39 52 90 14 89
49 66 33 19 45 61 31 29 84 98 58
36 53 35 33 88 90 19 23 76 23 76
77 27 25 42 70 36 35 91 17 79 43
33 85 33 59 47 46 63 75 98 96 55
75 88 10 57 85 71 34 10 59 84 45
29 34 43 46 75 28 47 63 48 16 19
62 57 91 85 89 70 80 30 19 38 14
61 35 36 20 38 18 89 64 63 88 83
45 46 89 53 83 59 48 45 87 98 21

15 95 24 35 79 35 55 66 91 95 86 87
94 15 84 42 88 83 64 50 22 99 13 32
85 12 43 39 41 23 35 97 54 98 18 85
84 61 77 96 49 38 75 95 16 71 22 14
18 72 97 94 43 18 59 78 33 80 68 59
26 94 78 87 78 92 59 83 26 88 91 91
34 84 53 98 83 49 60 11 55 17 51 75
29 80 14 79 15 18 94 39 69 24 93 41
66 64 88 82 21 56 16 41 57 74 51 79
49 15 59 21 37 27 78 41 38 82 19 62
54 91 47 29 38 67 52 92 81 99 11 27
31 62 32 97 42 93 43 79 88 44 54 48
```

then your program would print the maximum number of points one can get by optimally distributing pebbles while respecting the two rules, which would be this:

```
572
683
2096
2755
```

Sample Solution

```
#!/usr/bin/env python

import sys
import copy

def computeOptimalScoreForSubBoard(gameBoard, currRow,
                                   currCol, permissions, cache):
   if currCol >= len(gameBoard[0]): # wrap here so recursive calls are easier
      currRow = currRow + 1
      currCol = 0
   if currRow == len(gameBoard): return 0

   if cache.has_key(key(currRow, currCol, permissions)):
      return cache[key(currRow, currCol, permissions)]

   permissionsWithout = copy.copy(permissions)
   permissionsWithout.pop(0)
   permissionsWithout.append(True)
   optimalScore = optimalScoreWithout = \
      computeOptimalScoreForSubBoard(gameBoard, currRow, currCol + 1, \
                                     permissionsWithout, cache)
   if (permissions[0]):
      permissionsWith = copy.copy(permissions)
      permissionsWith.pop(0)
      permissionsWith[-1] = False
      if currCol > 0: permissionsWith[-2] = False
      if currCol < len(gameBoard[0]) - 1: permissionsWith[0] = False
      permissionsWith.append(currCol == len(gameBoard[0]) - 1)
      optimalScoreWith = gameBoard[currRow][currCol] + \
         computeOptimalScoreForSubBoard(gameBoard, currRow, currCol + 1, \
                                        permissionsWith, cache)
      if (optimalScoreWith > optimalScore): optimalScore = optimalScoreWith
   cache[key(currRow, currCol, permissions)] = optimalScore
   return optimalScore

def computeOptimalScore(gameBoard):
   permissions = map(lambda x: True, range(0, len(gameBoard[0]) + 1))
   cache = {}
   return computeOptimalScoreForSubBoard(gameBoard, 0, 0, permissions, cache)

def key(currRow, currCol, permissions):
   return str(currRow) + " - " + str(currCol) + " - " + str(permissions)

def readBoard():
   gameBoard = []
   for line in sys.stdin:
      line = line.rstrip("\n")
      if line == "": return gameBoard
      row = map(int, line.split())
      gameBoard.append(row)
   return None

while True:
   gameBoard = readBoard()
   if gameBoard == None: break
   print computeOptimalScore(gameBoard)
```