8th Annual NB (English)

High School Programming Competition

hosted by UNB Saint John

Saturday, May 10, 2014

Competition Problems TEAM

Rules:

1. **For each problem, your program must read from the standard input and write to the standard output. Your programs cannot use files for input or output.**
2. **Your submission to each problem will consist of a single source-code file (i.e., you will create just one source-code file per problem).**
3. **Your source code will be recompiled by the judges prior to testing.**
4. **The output of your programs must correspond exactly with the examples provided, including spelling, spacing and capitalization. Note that in the examples provided, each line (including the final line) is terminated with an end-of-line character.**
5. **The sample inputs and corresponding outputs for each problem are in the contest folder, on the PC's desktop.**

There are 8 questions. Good luck!

Problem 1: making change

When going to a store and buying things cash, people rarely give the exact amount to the cashier. For example, if they have to pay a total of $3.50, they will give a $5 bill, and the cashier will give back $1.50. Of course, the client does not want to receive this change back in nickels and dimes only! We all want to get this change using the least number of coins.

Fortunately, our Canadian denomination system makes it relatively easy to return an amount with the least number of coins: we just have to start with the largest denomination first, and work our way from the largest to the smallest denomination. For example, if we need to give 65 cents in change, we have to use two quarters (25 cents), one dime (10 cents), and one nickel (5 cents). For simplicity here, we will assume that pennies (1 cent) still exist.

INPUT: The input contains 5 lines, each containing an amount to pay (in cents). This amount will always be a whole number between 1 and 99.

OUTPUT: For each of the amounts in the input, there should be one corresponding line, indicating the coins needed to return the proper change, assuming that the client gave a $1 (i.e., 100 cents). The sequence is important here: the first line of output should correspond to the first line of input, and etc. Each line is composed of the following 4 numbers (each separated by one space, and no space at the end): the first is the number of quarters required, the second is the number of dimes required, the third is the number of nickels required, and the last is the number of pennies required. For example, in the example below, the first line is for paying 35 cents. So here, 65 cents have to be returned. As indicated above, this is done using 2 quarters, 1 dime, 1 nickel, and no penny.

EXAMPLE INPUT:
35
99
46
68
50

EXAMPLE OUTPUT:
2 1 1 0
0 0 0 1
2 0 0 4
1 0 1 2
2 0 0 0

Problem 2: spy encoding

In the early days of spying, it was common to encode messages using a simple rule of substituting each letter with another one. Of course, both the sender and the receiver would have to know the substitution rule (as a table showing which letter is substituted for which one) in advance. There were some problems though: after a while of using the same substitution table, the enemies were starting to figure out how to decode those messages. So some new tricks for changing the rules on a daily basis were required. One example of such trick could be to use the day the message was created as the clue on how to decode the message. For example, if the message was written on January 3 (i.e., month=1 and day=3), each upper case letter would be shifted by 1 (i.e., A→B, B→C, …, Y→Z, and Z→A), and each lower case letter would be shifted by 3 (i.e., a→d, b→e, c→f, …, w→z, x→a, y→b, z→c), for encoding the message. You have to write a program for decoding such a message, assuming it was written on February 5th.

INPUT: The encoded message, all on a single line.

OUTPUT: The decoded message, again on a single line.

EXAMPLE INPUT:
Yj bfsy yt jfy Rneef!

EXAMPLE OUTPUT:
We want to eat Pizza!

Problem 3: dollar-bill collectors

If you have some dollar bills in your pocket, some of them might be worth a lot more than their face value. Some specialized collectors are ready to pay a high price for dollar bills having a special serial number. There are many types of special serial numbers, but for our problem here, we will be focusing on two of those types: “block” and “binary”. The “block” type is that all digits of the serial number are the same (e.g., 2222222). The “binary” type is that the serial number is composed of only two different digits, in any order (e.g., 6226662). Note: the serial number on Canadian dollar bills is composed of a few letters, followed by exactly seven digits. However, for our purpose, we will only look into the seven-digit part.

Since there are many types of special serial numbers (so they might be easy to forget), you are interested in writing a program where you could enter the serial number of the bills you have, and the program would automatically identify if you are lucky to have one of those special bills.

INPUT: Four seven-digit numbers, one per line.

OUTPUT: One of the following words (be careful to have the right spelling!), one per line, indicating the type of the corresponding serial number in the input: block, binary, or notSpecial. The first output line corresponds to the first input line, etc.

EXAMPLE INPUT:
5555523
7717117
4653888
9999999

EXAMPLE OUTPUT:
notSpecial
binary
notSpecial
block

Problem 4: communication towers

In a communication network, a number of towers are placed strategically for transmission purposes. Each of these towers has an effective range (or radius covered) of, say, 5 km. In order to have good reception of the signal, and to ensure one would not be disconnected in case of failure of one of the towers, it is recommended to be in a location covered by at least two towers. Your job is to check that each of the locations provided are indeed covered by at least two towers. Note: in order to be covered by a tower, the direct distance between the location and the tower must be smaller than 5. As a reminder, here is the formula to calculate the distance between two points (one at coordinates <x1,y1> and one at coordinates <x2, y2>):

$$\sqrt{(x1 - x2)^2 + (y1 - y2)^2}$$

INPUT: The input begins with a single line containing N, a positive whole number that indicates the number of towers. Then there are N lines. Each consists of the coordinates of each of those towers, as 2 positive whole numbers separated by a space: the X coordinate, then the Y coordinate. This is followed by an empty line, and then a positive whole number M (on a single line) indicating the number of locations to be checked. There are then M lines. Each consists of the coordinates of each of these locations (again, as 2 positive whole numbers separated by a space: X coordinate, then Y coordinate).

OUTPUT: The index of the locations that are not properly covered (the first location in the input file having an index of 1, the second one having an index of 2, and so on). You can assume that there will always be at least one location not properly covered. The indexes should be displayed in ascending order.

For the example below, the first location is properly covered, as it is clearly within a distance of 5 of both the towers 1 and 3. However, the second location is not properly covered because it is within the range of tower 3 only. The third location is way out of range of any of the towers.

EXAMPLE INPUT:

```
3
5  3
2  8
6  9

3
5  6
8  11
30  9
```

EXAMPLE OUTPUT:

```
2
3
```

Problem 5: neighbor fencing disputes

A number of neighbors, who are in a dispute over the exact location of the land they own, have decided to bring the problem to the city's offices. Each of them is requesting the permission to put a fence around what they believe to be their properties. However, before the city can approve the fencing request, they have to make sure that the areas to be fenced are not overlapping (which would indicate the source of the dispute among the neighbors). To speed up the process, you are asked to write a program that will take these fencing requests (as location where the fence is going to be), and return how many pairs of neighbors have overlapping properties. For simplicity, we will assume that all properties are in the shape of a rectangle, aligned perfectly on the north-south axis or the west-east axis. The fencing requests are made by specifying the coordinate of the north-west corner on the map (i.e., the top-left corner), and the dimensions of the land: west-east axis first (or horizontal size), and then north-south axis (or vertical size). The origin (or the point at coordinate X=0 and Y=0) is located at the bottom-left corner of the map.

INPUT: The input begins with a single line containing N, a positive whole number that indicates the number of neighbors. Then there are N lines, corresponding to the property locations for each of the N neighbors. The property location is composed of 4 numbers (only positive whole numbers), separated by a space: the first one is the horizontal coordinate of the top-left corner, the second one is the vertical coordinate of the top-left corner, the third one is the horizontal dimension, and the last one is the vertical dimension (when viewing the properties on a map). For example, if the line is "1 3 3 1", then the rectangle representing the property would have the following corners: top-left corner at <1,3> and bottom-right corner at <4,2>.

OUTPUT: The number of pairs of properties that are overlapping.

EXAMPLE INPUT:
4
1 3 3 1
1 7 1 2
3 5 1 3
2 6 3 5

EXAMPLE OUTPUT:
3

Problem 6: event pattern matching

In the world of advertisement, identifying patterns in what potential clients are doing can help to have better targeted ads. For example, if a lot of people are going to a coffee shop after going to see a movie, then it would be wise to put an advertisement for a coffee shop near the exit of a movie theater. Identifying such pattern is not easy though, as the pattern could be interwoven with other events (e.g., some of those people going to a coffee shop after a movie might make a short stop in between to buy gas). What is important here is to identify a subset of events, which are happening in the same sequence, for a relatively large group of people.

Of course, the identification of such pattern is too complex for a programming competition. What you are asked to do here is to write a program that will check how many people are satisfying a given pattern. For simplicity, those events are coded. For example, we might have the following events:
A – leave work
B – leave home
C – go to restaurant
D – go see a movie
E – go to coffee shop
F – buy gas
G – go get an ice cream cone

Then, a string containing the letters above would indicate a sequence of events for a particular person. For example, the string "ACDE" would mean that someone goes to restaurant after work, then go see a movie, and finally go to the coffee shop. We say that this string contains the pattern "DE" (i.e., going to coffee shop after seeing a movie), but many other strings could contain it, such as: BDFE, ADGHE, and GDECG. Note: your program should work with events that could be coded with any letter from A to Z (always upper case). Also, the pattern and the strings of events will always contain at least one letter, and those letters can be repeated more than once in the same string. The strings can be of any length (max 100 characters).

INPUT: The first line contains the pattern to locate in the strings of events. The second line contains N, a positive whole number indicating the number of strings of events that should be checked. Then there are N lines, each of which containing a string of events to be checked.

OUTPUT: For each of the strings of events, a "yes" or a "no" should appear (one per line), indicating whether the corresponding string does contain the pattern or not. Note: it is important that the first line of output correspond to the first string of event in input (i.e., third line), etc. The last line should indicate a count of the number of "yes".

(see the example input and output on the next page)

EXAMPLE INPUT:
ADE
4
ADE
ACEGD
BDFE
AEDFE

EXAMPLE OUTPUT:
yes
no
no
yes
2

Problem 7: peaks identification

With the new technologies and sensors available, there are more and more opportunities to monitor certain variables over time, all automatically. For example, at Environment Canada, they store the maximum temperature reached every day, in many locations. Such sequence of data over a given period of time (for a single location) is called a "time series". There are many further analyzes that might be interesting to do on those time series. The type of analysis that we are interested in here is the identification of "local peaks": data points that are significantly higher than their neighbors in the time series. For our example above, that would mean identifying the days that were significantly warmer than the few days before and after.

Let's see how this concept of "local peak" works with a real example. The first row of the table below indicates the daily maximum temperature in Saint John, for the first 15 days of April 2014 (rounded). The numbers above the table indicate the day. The second row indicates the average temperature for the 5 days around: the first number (3.2) is the average over days 1 to 5 inclusively (i.e., (1+1+4+6+4)/5), the second number (4.4) is the average over days 2 to 6 (i.e., (1+4+6+4+7)/5), the third number (6.4) is the average over days 3 to 7 (i.e., (4+6+4+7+11)/5), etc. The local peaks (shaded areas) are the days where the temperature was more than 0.999 degree above the corresponding average over the 5 days around it (which we will define as being "significantly above average" here).

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	4	6	4	7	11	8	10	6	10	13	6	10	15
		3.2	4.4	6.4	7.2	8.0	8.4	9.0	9.4	9.0	9.0	10.8		

INPUT: The input begins with a single line containing N, a positive whole number that indicates the number of values in our time series. Then there are N lines, consisting of these N values in sequence (always whole numbers).

OUTPUT: The days (assuming that the first one is day 1) where we have a local peak, as defined above (i.e., where the temperature was more than 0.999 degree higher than the average of the 5 days around it). When the local peak covers more than one day, a range should be provided instead, as the first day and the last day of that peak, separated by a dash character ("-"). Note: the first 2 days and the last 2 days are never considered, as we cannot calculate a local average for these. You can assume that there will always be at least one peak.

(see the example input and output, matching the table above, on the next page)

EXAMPLE INPUT:
15
1
1
4
6
4
7
11
8
10
6
10
13
6
10
15

EXAMPLE OUTPUT:
4
7
9
11-12

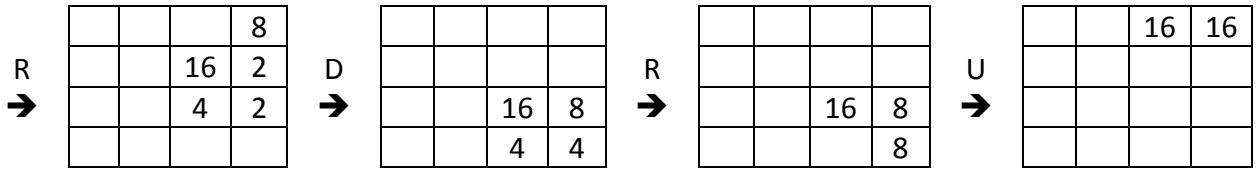
Problem 8: bouncing numbers game

Recently, a new interesting game has been launched, named "2048". You may or may not have heard about it, but that is not important. It is played on a 4-by-4 grid of numbers. Players try to "build" numbers as high as possible, by "bouncing" the numbers up, down, left or right. For example, if the player "moves" (or "bounces") the numbers to the right, then they slide one after the other (taking numbers from right to left, separately for each row) to the rightmost empty slot. When 2 numbers that are the same "hit" each other, they get merged into a single number that is the sum of the two numbers merged. In the real game, new numbers are appearing in the grid as we play, but we will not use this feature here.

For the problem here, we will assume that you get such a 4-by-4 grid containing some numbers and some empty spaces. For example, let's assume that you get the following grid:

		8	
	16		2
4		2	

You also get a number of moves: "U" for up, "D" for down, "L" for left, and "R" for right. You have to apply those moves in sequence, and see what the grid would look like after those moves. For example, with the sequence of moves "RDRU", the grid above would be modified as follows:



Note: if you get 3 numbers that are the same, in sequence, the way to merge them depends on the direction of the move. For example, the sequence "2 – 2 – 2" becomes "2 – 4" if moved right, but it becomes "4 – 2" if moved left.

INPUT: The original 4-by-4 grid, with the 16 numbers (or "." for an empty space) appearing one per line (read in the table left to right, then up to down, as you would read in a book). Those numbers are all positive whole numbers. The last line of the input contains the string representing the sequence of moves (note: there could be any number of such moves, but there will always be at least one).

OUTPUT: The final grid after performing all moves, using the same notation as in the input (i.e., the 16 numbers – or "." for an empty space – one per line).

The following example input and example output are for the example provided above:

EXAMPLE INPUT:

.
.
8
.
.
16
.
2
4
.
2
.
.
.
.
.
RDRU

EXAMPLE OUTPUT:

.
.
16
16
.
.
.
.
.
.
.
.
.
.
.
.