Seventh Annual
University of Central Florida

acm • ΥΠΕ

High School Programming Tournament

Problems

Problem Name	Filename
How Many Zeroes?	ZEROES
Cross Words	CROSS
Mind Your PQs	PQUEUE
Interesting Intersections	SEGMENT
Dave's Socks	SOCKS
It Makes No Difference in the End	SUBTRACT
Orthogonal Latin Squares	LATIN
Found in the Shuffle	DECK
Rot13 Encryption	THIRTEEN
The 15-Puzzle	PUZZLE

Call your program file: *Filename*.PAS or *Filename*.C
Call your input file: *Filename*.IN

For example, if you are solving Found in the Shuffle:

Call your program file: DECK.PAS or DECK.C
Call your input file: DECK.IN

How Many Zeroes?

Filename: ZEROES

Factorial is an operation defined for integers greater than or equal to zero by the following formula:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{if } n > 0 \end{cases}$$

where n is a non-negative integer.

The Problem:

Given a non-negative integer n, determine the number of zeroes at the end of the expansion of $n!$. Here are some examples:

n	$n!$	Number of zeroes at end
4	24	0
5	120	1
7	5040	1

The Input:

There will be several data sets. Each set consists of one line of input, containing n.

The Output:

For each input data set, output one of the following messages, whichever is appropriate:

```
There is 1 zero at the end of n!.
There are x zeroes at the end of n!.
```

where x and n are the appropriate values.

Sample Input:

Sample Output:

```
There are 0 zeroes at the end of 4!.
There is 1 zero at the end of 5!.
There is 1 zero at the end of 7!.
```

Cross Words

Filename: CROSS

A *word cross* is formed by taking two words and intersecting them on a common letter, so that the first word appears horizontally while the second word appears vertically. In a *leading* word cross, the common letter of intersection always occurs as far to the left as possible in the horizontal word. The vertical word then intersects the horizontal word on this same letter, as near as possible to the top of the vertical word.

The Problem:

Given two pairs of words, display them as side-by-side leading word crosses. If this is not possible for the given words, print a message that says so.

The Input:

There will be an unknown number of word sets in the input. Each word set will consist of four words, with one word per line. A word will consist of one to ten uppercase letters.

The Output:

Your output will consist of two leading word crosses for each input word set, if both can be formed. The first cross will contain the first pair of words in the word set, and the second cross will contain the second pair. In each pair, the first word is to appear horizontally with the second crossing it vertically. The two horizontal words are to appear on the same line of output, separated by exactly two spaces. The first character of the first horizontal word must be in the first column of the output.

If one or both of the word crosses cannot be formed for the given word set, print the message:

```
Cannot cross the words.
```

There should be exactly one blank line between the output for different word sets.

Sample Input:

```
ALTER
POLARITY
NEUTRON
FLOW
CHEESECAKE
MATCHES
PICNIC
EXCUSES
PEANUT
BANANA
VACUUM
GREEDY
```

Sample Output:

```
P
O            F
L            L
ALTER  NEUTRON
R            W
I
T
Y

M
A              E
T              X
CHEESECAKE  PICNIC
H              U
E              S
S              E
               S

Cannot cross the words.
```

Mind Your PQs

Filename: PQUEUE

One way to organize data is with a PQ, or *priority queue.* A PQ is an organized collection of data that is accessed only through certain operations. A data element is added to a PQ by the INSERT operation. The REMOVE operation chooses the data element in the PQ with the *least value*, and removes it from the PQ. A REMOVE operation is valid only when the PQ is not empty.

The Problem:

Implement a PQ whose data elements are integers.

The Input:

There will be several data sets, each representing a series of operations on a PQ. Each data set will contain several lines, which will have one of the following forms:

```
INSERT num
REMOVE
```

where *num* is an integer to be added to the PQ. There is exactly one space between INSERT and *num*. All operations will be valid, and there will be no more than 100 elements in a PQ at a time. The end of a data set for a given PQ will be denoted by a line of the following form:

```
END
```

The Output:

Output each integer REMOVED from a PQ. Print one integer per line, left justified. Separate the output from different PQs by a blank line.

Sample Input:

```
INSERT 4
INSERT 10
INSERT 3
REMOVE
REMOVE
REMOVE
END
INSERT 100
REMOVE
INSERT 50
REMOVE
END
```

Sample Output:

```
3
4
10

100
50
```

Interesting Intersections

Filename: SEGMENT

The Problem:

Given a circle and a line segment, determine whether the line segment intersects the circle.

The Input:

There will be several data sets. Each data set will consist of exactly two lines of input. The first line will contain three real numbers x, y, and r, where the point (x, y) is the center of the circle and r is its radius (r will be positive). The second line will contain four real numbers x_1, y_1, x_2, and y_2, where (x_1, y_1) and (x_2, y_2) are the endpoints of the line segment. (x_1, y_1) and (x_2, y_2) will not be the same point.

The Output:

For each data set, print one of the following messages, whichever is appropriate:

```
The line segment intersects the circle.
The line segment does not intersect the circle.
```

Sample Input:

```
0.0 0.0 1.0
10.0 10.0 20.0 20.0
5.0 0.0 4.0
0.0 10.0 10.0 -10.0
```

Sample Output:

```
The line segment does not intersect the circle.
The line segment intersects the circle.
```

Dave's Socks

Filename: SOCKS

Dave likes to wear mismatched socks. In fact, he refuses to wear socks that match. This sometimes means that he must plan his wardrobe days in advance. For example, suppose Dave has 1 Red sock, 1 Green sock, and 2 Blue socks left in his drawer. If he wears the Red and Green socks today, then tomorrow he is left with matching Blue socks, which makes him unhappy. Thus, he must wear one Blue sock today in order to have a mismatched set tomorrow. Dave always wears a clean pair of socks, and doesn't do laundry until all of his socks have been worn.

The Problem:

Given the contents of Dave's sock drawer, produce a day-by-day plan of how he could wear the socks, always mismatched, until they have all been worn. If this isn't possible, produce a message to this effect.

The Input:

There will be several input sets. Each set will begin with a line containing a single integer, *n*, from 1 to 15, indicating the number of different colors of socks currently in Dave's sock drawer. The next *n* lines will each contain a positive integer followed by a single space and then a string. The integer indicates the number of socks of that color, and the string indicates the color. The string will contain 1 to 20 letters (no numbers, spaces or other characters), and no two colors in any data set will be the same.

The Output:

For each data set, if there is a solution, print the daily plan on consecutive lines. Each line should contain two colors (as they were input) separated by a space, indicating a mismatched pair of socks to wear on that day. If no solution is possible, then print the single-line message

```
Dave can't do it.
```

Separate the output from consecutive data sets with a blank line.

Sample Input:

```
3
1 Red
1 Green
2 Blue
1
2 Lavender
```

Sample Output:

```
Blue Green
Red Blue

Dave can't do it.
```

It Makes No Difference in the End

Filename: SUBTRACT

Consider a sequence of four non-negative integers. Each integer is a *neighbor* of the integers immediately before it and after it in the sequence. Also, the first and last numbers are neighbors. The absolute-value differences of neighboring integers can be used to construct a new sequence. For example, in the sequence 28 1 25 37, the differences are 28-1=27, 25-1=24, 37-25=12, and 37-28=9, giving a new sequence of 27 24 12 9. This process can be repeated until all integers in the new sequence are the same. Once this occurs, all successive sequences will be all zeroes.

The Problem:

Given a sequence of four non-negative integers, determine all successive sequences (obtained by differences of neighbors, as described above) until all of the integers in the sequence are the same.

The Input:

There will be several sequences in the input. Each sequence will consist of four non-negative integers, all on one line, separated by spaces.

The Output:

For each input sequence, output the original sequence followed by all successive sequences, one sequence per line. The first integer in each sequence should be the difference of the first two numbers of the previous sequence. Print each integer right-justified in a five-character field. Separate the output for different input sequences by a blank line.

Sample Input:

```
28 1 25 37
8 7 4 11
```

Sample Output:

```
   28    1   25   37
   27   24   12    9
    3   12    3   18
    9    9   15   15
    0    6    0    6
    6    6    6    6

    8    7    4   11
    1    3    7    3
    2    4    4    2
    2    0    2    0
    2    2    2    2
```

Orthogonal Latin Squares

Filename: `LATIN`

A **Latin Square** of size n is an n by n matrix in which each row contains the numbers 1 through n, and each column also contains the numbers 1 through n. Here are two Latin Squares of size 3:

3	2	1
2	1	3
1	3	2

3	2	1
1	3	2
2	1	3

Two Latin Squares of the same size can be combined to form a matrix of *ordered pairs*. Each ordered pair contains a number from a given position in the first Latin Square and the number from the same position in the second Latin Square. For example, the two Latin Squares shown above would combine into this matrix:

(3, 3)	(2, 2)	(1, 1)
(2, 1)	(1, 3)	(3, 2)
(1, 2)	(3, 1)	(2, 3)

Because these are *ordered* pairs, the numbers from the first Latin Square will always appear first, and the pair (2, 1) is not considered to be the same as (1, 2). Since all of the ordered pairs in the above matrix are unique, the Latin Squares are said to be **orthogonal**. Here are two Latin Squares of size 3 which are **not** orthogonal:

1	2	3
2	3	1
3	1	2

3	2	1
2	1	3
1	3	2

All that is needed to prove the Latin Squares are **not** orthogonal is a single counter-example. For these Latin Squares, the ordered pair obtained from the upper right corners is (3, 1). However, the pair obtained from the lower left corners is also (3, 1). Therefore they are not orthogonal.

The Problem:

Given two Latin Squares, determine whether or not they are orthogonal.

The Input:

There will be several sets of input data. The first line of each data set will contain a single integer, n, from 1 to 15, representing the size of the two Latin Squares to be considered. The first Latin Square will occupy the next n lines of the data set. Each of those lines will contain n integers. The second Latin Square will follow on the next n lines of the data set, in the same format.

The Output:

For each input set of Latin Squares, output one of the following messages, whichever is appropriate:

```
The Latin Squares are orthogonal.
The Latin Squares are NOT orthogonal.
```

Print the messages on consecutive lines.

Sample Input:

```
3
3 2 1
2 1 3
1 3 2
3 2 1
1 3 2
2 1 3
3
1 2 3
2 3 1
3 1 2
3 2 1
2 1 3
1 3 2
```

Sample Output:

```
The Latin Squares are orthogonal.
The Latin Squares are NOT orthogonal.
```

Found in the Shuffle

Filename: DECK

An ordinary deck of playing cards contains 52 cards, each of which has a suit and a value. Most card games require that a deck be shuffled at the beginning of a play. However, sometimes the deck isn't shuffled well enough, and there are arrangements of cards in the deck which may cause a bias in the play. Two such arrangements are a *same-suit sequence* and an *ascending sequence.* A same-suit sequence is simply a sequence of consecutive cards in the deck with the same suit. An ascending sequence is a sequence of consecutive cards in the deck that follow one another in increasing value, with Ace following King and preceding two. Thus, 2S 5S KS 3S AS is a same-suit sequence of length five, 9C 10D JC QS KH AC 2D is an ascending sequence of length seven, and 2H 3H 4H 5H 6H is both a same-suit and ascending sequence of length five.

The Problem:

Given a deck of cards, determine the longest ascending sequence and the longest same-suit sequence present in the deck.

The Input:

The input will consist of a series of decks of cards, each deck occupying two lines with 26 cards per line. Each card will be represented by a two-character string: the value followed by the suit. Values will be the characters `A` for ace, `2-9` for two through nine, `T` for ten, `J` for jack, `Q` for queen, `K` for king. Suits will be `S` for spades, `D` for diamonds, `H` for hearts, and `C` for clubs.

The Output:

For each input deck, output the messages (each on a separate line):

```
Longest same-suit sequence: n
Longest ascending sequence: m
```

where *n* and *m* are the appropriate values. Separate the output from different decks by a blank line.

Sample Input:

```
2S5SKS3SAS9CTDJCQSKHAC2D2H3H4H5H6HQD9SJD8HAH4D7CJS8C
KD5C2CQHTS9H5DJHQC4C8D7STHAD7H6D6C6S9D4S7DKC3D8S3CTC
QCTD4C8D7STHAD7H2D3S6D6C6S9D4SAS7D2HKC5H3DTC8S9C3H3C
QD9SQSJD8HAH2SKS4D4H5S7CJS8CKD5C2CACQHJCTS6HKH9H5DJH
```

Sample Output:

```
Longest same-suit sequence: 5
Longest ascending sequence: 7

Longest same-suit sequence: 3
Longest ascending sequence: 2
```

Rot13 Encryption

Filename: THIRTEEN

Rot13 encryption is a simple encoding technique. Each letter in a message is replaced by another letter. The first 13 letters of the alphabet are replaced by the last 13 letters and the last 13 letters are replaced by the first 13. For example, ABC becomes NOP and XYZ becomes KLM. The case of each encoded letter is maintained, i.e., ABCxyz becomes NOPklm. In this encoding technique, only letters are encoded. Punctuation and spacing remain the same in the original and encoded messages.

The Problem:

Given a message containing several lines of text, produce a properly encoded version of the message, using the method described above.

The Input:

The input will consist of several lines of text. Each line will contain at least one letter.

The Output:

Output the properly encoded version of the message. All spacing, punctuation, and capitalization must be the same as in the original message.

Sample Input:

```
ABC
XYZ
ABCxyz
"Rot13 Encryption" has thirteen letters.
There are thirteen words in this sentence, if you count all of them.
```

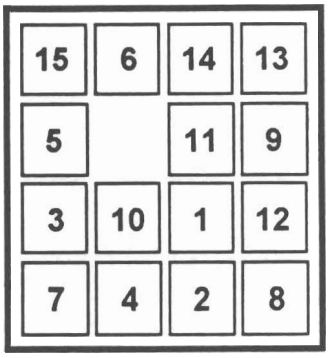
Sample Output:

```
NOP
KLM
NOPklm
"Ebg13 Rapelcgvba" unf guvegrra yrggref.
Gurer ner guvegrra jbeqf va guvf fragrapr, vs lbh pbhag nyy bs gurz.
```

The 15-Puzzle

Filename: `PUZZLE`

The 15-Puzzle involves a 4-by-4-square box containing 15 square tiles, numbered from 1 through 15, as shown below. Note that one square is left empty.



With the tile arrangement shown, either the 6, 5, 11, or 10 tile may slide into the empty square. Moving one of those tiles has the effect of "moving" the empty square to a new position.

The Problem:

Given an initial tile arrangement and a list of directions in which the empty square is moved, produce the final tile arrangement.

The Input:

There will be several data sets, each composed of two parts. The first part specifies an initial tile arrangement for a 15-Puzzle. The integers 0 through 15 will be given on four lines with four integers per line. The zero indicates the initial position of the empty square. The second part of each data set begins with a line containing a single integer, *n*. The next *n* input lines will contain a direction indicating where the empty square is moved. All moves will be valid. Directions will be specified as one of the following uppercase characters: `U` (for up), `D` (for down), `L` (for left), and `R` (for right).

The Output:

For each data set, output the final tile arrangement. Justify each number in a field of two characters and leave one space between columns, as shown in the sample. Use two spaces (not a zero) to represent the empty square. Leave a blank line between the output for different data sets.

Sample Input:

```
15 6 14 13
5 0 11 9
3 10 1 12
7 4 2 8
2
D
R
1 2 3 4
5 6 7 8
10 0 12 11
9 13 14 15
3
R
U
L
```

Sample Output:

```
15  6 14 13
 5 10 11  9
 3  1    12
 7  4  2  8

 1  2  3  4
 5     6  8
10 12  7 11
 9 13 14 15
```