University of Scranton
ACM Student Chapter / Computing Sciences Department
24th Annual High School Programming Contest (2014)

Problem 1: Four-Tuple Transformation

Suppose that you start with a four-tuple of nonnegative integers (a, b, c, d) and that you repeatedly apply this rule:

$$(a, b, c, d) \ \rightarrow \ (|a - b|, |b - c|, |c - d|, |d - a|)$$

That is, you replace a by $|a - b|$, b by $|b - c|$, c by $|c - d|$, and d by $|d - a|$.

It turns out that, eventually, you will obtain the four-tuple $(0, 0, 0, 0)$. For example, starting with $(7, 3, 6, 1)$ and applying the above rule repeatedly, we get

$$(7, 3, 6, 1) \rightarrow (4, 3, 5, 6) \rightarrow (1, 2, 1, 2) \rightarrow (1, 1, 1, 1) \rightarrow (0, 0, 0, 0)$$

In this case, it took four applications of the rule to transform $(7, 3, 6, 1)$ into $(0, 0, 0, 0)$, so we say that its *distance to zero* is four.

Develop a program that, given a four-tuple of nonnegative integers, reports its distance to zero.

Input: The first line contains a positive integer n indicating how many four-tuples are to be analyzed. Each of the next n lines contains four nonnegative integers describing a four-tuple.

Output: For each given four-tuple, display it, a colon, a space, and its distance to zero, as illustrated below.

```
Sample input:          Resultant output:
------------           ----------------
4
7 3 6 1                (7,3,6,1): 4
0 0 0 0                (0,0,0,0): 0
17 28 3 9              (17,28,3,9): 7
36 21 9 20             (36,21,9,20): 6
```

Problem 2: Date Format Conversion

Develop a program that, given a calendar date such as `January 28, 1974`, translates it into the `yyyy-mm-dd` format. For this example, that would be `1974-01-28`.

The names of the twelve months of the year, in order, are `January`, `February`, `March`, `April`, `May`, `June`, `July`, `August`, `September`, `October`, `November`, and `December`.

Input: The first line contains a positive integer n indicating how many calendar dates are to be processed. Each of the n lines thereafter contains one date, expressed using a month name, a space, a day number, a comma, a space, and a year number. (The numbers will be positive and have no leading zeros.)

Output: For each date given, the program should display it (in its given form), followed by a space, a colon, another space, and then the result of translating it to `yyyy-mm-dd` form. Note that the year component must be four digits in length and the month and day components must be two digits in length.

```
Sample input:          Resultant output:
------------           ----------------
3
January 28, 1974       January 28, 1974 : 1974-01-28
May 4, 2003            May 4, 2003 : 2003-05-04
November 17, 843       November 17, 843 : 0843-11-17
```

Problem 3: Bridge Crossing

Four hobbits must cross a narrow, rickety bridge over the Great River Anduin in order to escape from orc-ridden Ithilien to the safety of Minas Tirith, the capital of Gondor, just west of the river. The bridge can support at most two hobbits at one time. Moreover, it is dark, and any party crossing the bridge needs the light provided by a lantern to avoid falling into the river below.

Unfortunately, the hobbits have only one lantern among them. Therefore, for all four to reach Gondor requires a total of five bridge crossings: on the first, third, and fifth, a pair of hobbits will cross from east to west into Gondor; on the second and fourth, one hobbit must cross back to Ithilien, ferrying the lantern back with him.

The hobbits differ in physical agility, and therefore the time needed by one hobbit to cross the bridge may differ from that needed by another. Of course, when two hobbits cross the bridge together, they must cross at the speed of the slower of the two.

Develop a program that, given, for each hobbit, the time (s)he needs to cross the bridge, reports how much time is needed to get all four hobbits across the bridge into Gondor, using the best possible strategy (i.e., the one that minimizes the sum of the times spent by the five bridge crossings).

Input: The first line contains a positive integer n indicating how many instances of the problem are to be solved. Each of the next n lines describes one instance, which is given by four integers t_1, t_2, t_3, and t_4 (with $0 < t_1 \leq t_2 \leq t_3 \leq t_4$) that indicate how many units of time are needed by the four hobbits, respectively, to cross the bridge.

Output: For each instance of the problem, the program should report the time needed to get all four hobbits across the bridge into Gondor, using whatever strategy gets them there the most quickly.

Sample input:	Resultant output:
5	
1 2 5 10	17
3 4 4 8	22
3 12 18 33	69
5 6 7 10	33
4 9 19 39	70

Problem 4: Dice Probabilities

The **outcome** of rolling a standard six-sided **die** is one among 1, 2, ..., 6, each with probability 1/6. The outcome of rolling a pair of such dice is one among (1, 1), (1, 2), ..., (1,6), (2,1), (2,2), ..., (6, 6), each with probability 1/36.

In many games that use dice, you roll a pair of dice and take the sum, which will be in the range 2..12. Each possible sum, referred to as an **event**, arises from any among a set of outcomes. Take the event 4 as an example. Because there are three outcomes (out of 36 equally likely outcomes) that yield a sum of 4 (namely (1, 3), (2, 2), and (3, 1)), the probability of the event of rolling 4 is 3/36 (or 1/12 in simplest terms). There are six outcomes that give rise to the most likely event, which is 7.

For this problem, we will consider non-standard dice described by two numbers, m and k, where m is the number of sides and the possible outcomes are k, $k + 1$, ..., $k + m - 1$, each having probability $1/m$. (Of course, $m \geq 1$.)

Develop a program that, given as input the description of two dice, reports the probability of each event, where the events correspond to the sums that are possible when rolling the two dice.

Input: The first line contains a positive integer n indicating how many pairs of dice are to be analyzed. Each of the next n lines describes a pair of dice. Each such description consists of four integers m_1, k_1, m_2, k_2 (with $m_1, m_2 \geq 1$). The ith die ($i = 1, 2$) has m_i sides and rolling it yields any of the outcomes k_i, $k_i + 1$, ..., $k_i + m_i - 1$.

Output: For each pair of dice described in the input, report the probability of each event in ascending order from smallest possible sum ($k_1 + k_2$) to largest ($k_1 + m_1 - 1 + k_2 + m_2 - 1$). Express each probability as a fraction with denominator $m_1 \cdot m_2$ rather than in simplest form. (See sample output below for proper formatting.) Display a blank line after the last probability.

Sample input and output appear on the next page.

```
Sample input:      Remarks:
-------------      --------
2
6 1 6 1            two standard 6-sided dice
3 4 5 0            3-sided die (outcomes 4..6) and 5-sided die (outcomes 0..4)


Resultant output:
-----------------
2 : 1/36
3 : 2/36
4 : 3/36
5 : 4/36
6 : 5/36
7 : 6/36
8 : 5/36
9 : 4/36
10 : 3/36
11 : 2/36
12 : 1/36

4 : 1/15
5 : 2/15
6 : 3/15
7 : 3/15
8 : 3/15
9 : 2/15
10 : 1/15
```

Problem 5: Numbrix Verify

A **Numbrix** puzzle is an $n \times n$ matrix in which each cell is either empty or contains an integer in the range $1..n^2$. The player's goal is to fill in the empty cells so that each of the integers in the range $1..n^2$ appears in the matrix and so that, for each k, $1 \leq k < n^2$, the cells in which k and $k+1$ appear are adjacent to each other. Two cells are defined to be adjacent if they have a side in common. (Having only a corner in common is not sufficient.)

Here is an example of a (correctly) completed 6×6 Numbrix puzzle:

```
34 33 32 19 18 17
35 30 31 20 21 16
36 29 28 23 22 15
 1  2 27 24 13 14
 4  3 26 25 12 11
 5  6  7  8  9 10
```

Develop a program that, given a completed Numbrix puzzle, reports whether or not it is correct. In the case that it is not, the program reports how many distinct values k, where $1 \leq k < n^2$, are such that k and $k+1$ do not appear in adjacent cells.

Note that it is not necessary to use a two-dimensional array to solve this problem.

Input: The first line contains a positive integer n indicating how many completed Numbrix puzzles are to be evaluated. Each Numbrix puzzle is described by a line containing a single positive integer m followed by m lines each containing m positive integers.

Output: For each of the given Numbrix puzzles, a single line of output is to be generated. If the puzzle is correct, that fact is to be reported using the word CORRECT. If the puzzle is not correct, the word INCORRECT should appear, followed by a colon, a space, and a count of how many cells contain a value $k < n^2$ such that no adjacent cell holds $k+1$.

In the second puzzle shown in the sample input on the next page, the three values that do not appear in cells adjacent to their successors are 5, 9, and 21.

Sample input and output appear on the next page.

```
Sample input:
-------------
2
6
34 33 32 19 18 17
35 30 31 20 21 16
36 29 28 23 22 15
 1  2 27 24 13 14
 4  3 26 25 12 11
 5  6  7  8  9 10
5
 1  2  5  7  8
10  3  4  6  9
11 12 13 14 15
25 22 20 19 16
24 23 21 18 17


Resultant output:
-----------------
CORRECT
INCORRECT: 3
```

Problem 6: Batting Averages

In baseball, a batting average is computed by dividing number of **hits** by number of **at bats** and rounding to the nearest thousandth. For example, two hits in seven at bats gives rise to a batting average of .286, as $2/7 = .\overline{285714}$. Moreover, in casual conversation, we usually multiply the average by one thousand, as though it were an integer between zero and one thousand (rather than a real number in the interval $[0, 1]$). In the example just cited, we would say that the batting average is "two [hundred] eighty six".

Develop a program that, given a batting average as a whole number in the range 0..1000, computes the smallest possible number of hits and at-bats that are consistent with that average. That is, given an integer k satisfying $0 \leq k \leq 1000$, the program reports the smallest nonnegative integers h and b such that $\frac{k}{1000}$ is obtained by rounding $\frac{h}{b}$ to the nearest thousandth (or, equivalently, such that k is obtained by rounding $\frac{1000 \cdot h}{b}$ to the nearest integer).

Input The first line contains a positive integer n indicating how many batting averages are to be processed. Each of the next n lines contains a batting average, expressed as an integer between 0 and 1000, inclusive. For example, the input `286` represents the batting average properly expressed as .286.

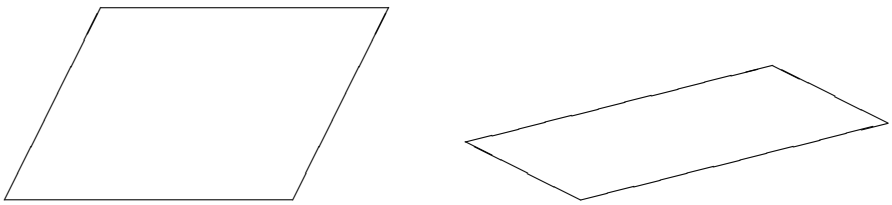
Output For each batting average provided as input, the program is to echo the batting average and indicate the minimum number of hits and at-bats that gives rise to that average. See below for proper formatting.

```
Sample input:              Resultant output:
-------------              -----------------
6
0                          0: 0 hits in 1 at bats
1000                       1000: 1 hits in 1 at bats
286                        286: 2 hits in 7 at bats
9                          9: 1 hits in 106 at bats
262                        262: 11 hits in 42 at bats
331                        331: 39 hits in 118 at bats
```

Problem 7: Parallelogram Completion

Develop a program that, given as input the cartesian coordinates of three vertices of a parallelogram, computes the coordinates of the fourth vertex, thereby "completing" the parallelogram.

Recall that a parallelogram is a polygon with four sides and four vertices such that each pair of opposite sides has the same length (and slope). Here are two examples:



Input: The first line contains a positive integer n indicating how many parallelograms are to be completed. For each one, three of its vertices A, B, and C are described on a single line via six real numbers, x_1, y_1, x_2, y_2, x_3, and y_3, separated by spaces. The intended interpretation is that $A = (x_1, y_1)$, $B = (x_2, y_2)$, and $C = (x_3, y_3)$. Also, it is to be understood that AB and BC are two adjacent (i.e., non-opposite) sides of the parallelogram.

Output: For each parallelogram to be completed, generate three lines of output. On the first line, display the three given vertices. On the second, display the fourth. (See the sample below for formatting details.) The third line should be blank.

Sample input and output appear on next page.

```
Sample input:
-------------
4
0.0 0.0 1.0 0.0 1.0 1.0
-2.0 1.0 2.0 5.0 -1.0 6.0
2.0 1.0 -3.0 -2.0 -7.0 0.0
2.5 5.2 2.5 -2.4 7.0 6.5


Resultant output:
-----------------
Three given vertices: (0.0,0.0) (1.0,0.0) (1.0,1.0)
Fourth vertex: (0.0,1.0)

Three given vertices: (-2.0,1.0) (2.0,5.0) (-1.0,6.0)
Fourth vertex: (-5.0,2.0)

Three given vertices: (2.0,1.0) (-3.0,-2.0) (-7.0,0.0)
Fourth vertex: (-2.0,3.0)

Three given vertices: (2.5,5.2) (2.5,-2.4) (7.0,6.5)
Fourth vertex: (7.0,14.1)
```

Problem 8: NFA String Acceptance

Depicted in the figure below is a *nondeterministic finite automaton* (NFA) that we will refer to as M. Each circle represents a *state* and each arrow labeled by a symbol represents a *transition* from one state to another (or itself) associated with that symbol. The unlabeled arrow points to the *initial state*. Double circles correspond to *final states*. By convention, we name the states in an NFA q_0, q_1, ..., q_{m-1}, where m is the number of states and q_0 is the initial state. The set of symbols appearing as labels on transitions is called the *alphabet* of the NFA.

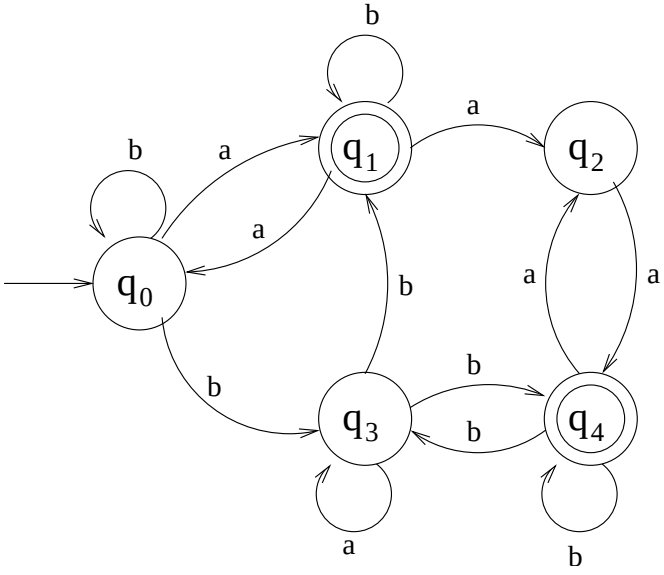


Figure 1: Nondeterministic Finite State Machine M

Consider the string *baab*. In M there are three *paths* beginning at the initial state that "spell out" that string:

$$q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_1 \xrightarrow{a} q_0 \xrightarrow{b} q_0$$
$$q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_1 \xrightarrow{a} q_0 \xrightarrow{b} q_3$$
$$q_0 \xrightarrow{b} q_3 \xrightarrow{a} q_3 \xrightarrow{a} q_3 \xrightarrow{b} q_4$$

Due to the fact that at least one of these paths (namely, the last one) ends in a final state, M is said to *accept baab*.

If, on the other hand, all paths spelling out a particular string end in non-final states, that string is said to be *rejected* by M. An example is the string *baa*, as the three paths that spell it out end in the non-final states q_0, q_2, and q_3, respectively.

Notice that it is possible for there to be *no* paths that spell out a particular string, in which case it is certainly rejected. An example of this is aab (or any string of which aab is a prefix). (The only path spelling out aa ends in q_2, but there is no outgoing transition from there labeled b; hence, even if q_2 were a final state, any string having aab as a prefix is rejected.)

Develop a program that, given an NFA with alphabet $\{a, b\}$ and some input strings composed of a's and b's, reports, for each input string, whether or not it is accepted by the NFA.

Input: The first line contains the number $m > 0$ of states in the NFA to be tested, followed by the number $k > 0$ of final states, followed by the number $t > 0$ of transitions. The second line contains k integers in the range $0..m-1$ identifying the NFA's final states. (The initial state is not explicitly identified because it is to be understood that it is state 0.) Each of the following t lines describes a single transition, which is given by two state identifiers (integers in the range $0..m-1$) p and q separated by the label (either a or b) on a transition that goes from state p to state q. The transitions need not be listed in any particular order. (The sample input below describes M.)

On the following line is a positive integer r indicating how many input strings are to be processed. Each of the following r lines contains one such string.

Output: For each string given as input, generate one line of output that classifies that string as being either accepted or rejected by the NFA.

```
Sample Input:
-------------
5 2 13
1 4
0 b 0
0 b 3
4 b 4
3 a 3
0 a 1
3 b 1
1 a 0
1 b 1
3 b 4
1 a 2
2 a 4
4 a 2
4 b 3
6
baab
aabab
bbbb
bbbababaa
aaaaaa
abbaaabaa
```

```
Resultant Output:
-----------------
baab is accepted
aabab is accepted
bbbb is accepted
bbbababaa is accepted
aaaaaa is rejected
abbaaabaa is rejected
```