University of Scranton
Computing Sciences Department
29th Annual High School Programming Contest (2019)

Problem 1: Roman Numerals to Decimal Numerals

Develop a program that, given a Roman numeral, computes the equivalent decimal numeral.

Symbol	I	V	X	L	C	D	M
Value	1	5	10	50	100	500	1000

The figure above lists the seven symbols that can appear in a Roman numeral and their corresponding values (expressed as decimal numerals). With some exceptions, the symbols in a Roman numeral go from larger-valued to smaller-valued as we go from left-to-right, and the value represented by the numeral is the sum of the values of the individual symbols within it. The exceptions to this are that `IV` represents 4 (i.e., 5 - 1), `XL` represents 40 (i.e., 50 - 10), `XC` represents 90 (i.e., 100 - 10), `CD` represents 400 (i.e., 500 - 100), and `CM` represents 900 (i.e., 1000 - 100).

For example, parsing `MMCDXLVIII` as `(M)(M)(CD)(XL)(V)(I)(I)(I)`, we get the sum $1000 + 1000 + 400 + 40 + 5 + 1 + 1 + 1$, or 2448. As another example, we would parse `MCMLIX` as `(M)(CM)(L)(IX)`, yielding the sum $1000 + 900 + 50 + 9$, or 1959.

To ensure that each positive integer has a unique representation as a Roman numeral, the terms in the sum corresponding to such a numeral must be in descending order of value and must be minimal in number. Thus, for example, in a Roman numeral you would never see substrings such as `IXX` $(9 + 10)$ or `CMM` $(900 + 1000)$, as they would be written as `XIX` $(10 + 9)$ and `MCM` $(1000 + 900)$, respectively, so that the terms are in descending order.

Nor would you ever see substrings such as `LXL` $(50+40)$ or `IVI` $(4+1)$, as they would be written as `XC` (90) and `V` (5), respectively, to minimize the number of terms.

Moreover, the symbols `I`, `X`, and `C` never appear more than three times in a row, as `IIII` would appear instead as `IV`, `XXXX` would be written `XL`, and `CCCC` would be written `CD`.

As for `V`, `L`, and `D`, none of them can appear more than once in a numeral.

Finally, none of the substrings `CMD`, `XCL`, or `IXV` can appear, as they would instead be written `MCD`, `CXL`, and `XIV`, respectively.

Continued on next page ...

Input: The first line contains a positive integer n indicating how many Roman numerals appear on subsequent lines. Each of the next n lines contains a single Roman numeral.

Output: For each Roman numeral provided as input, on one line display it, followed by a space, followed by an equals sign, followed by a space, followed by its equivalent decimal numeral.

```
Sample input:                 Resultant output:
-------------                 -----------------
8
II                            II = 2
IV                            IV = 4
XXIV                          XXIV = 24
MMMMMIII                      MMMMMIII = 5003
MCDLIX                        MCDLIX = 1459
MMCMXLI                       MMCMXLI = 2941
CDXXXV                        CDXXXV = 435
MDXCVII                       MDXCVII = 1597
```

Problem 2: Decimal Numerals to Roman Numerals

Develop a program that, given a decimal numeral, computes the equivalent Roman numeral.

Symbol	I	V	X	L	C	D	M
Value	1	5	10	50	100	500	1000

The figure above lists the seven symbols that can appear in a Roman numeral and their corresponding values (expressed as decimal numerals). With some exceptions, the symbols in a Roman numeral go from larger-valued to smaller-valued as we go from left-to-right, and the value represented by the numeral is the sum of the values of the individual symbols within it. The exceptions to this are that `IV` represents 4 (i.e., 5 - 1), `XL` represents 40 (i.e., 50 - 10), `XC` represents 90 (i.e., 100 - 10), `CD` represents 400 (i.e., 500 - 100), and `CM` represents 900 (i.e., 1000 - 100).

For example, parsing `MMCDXLVIII` as `(M)(M)(CD)(XL)(V)(I)(I)(I)`, we get the sum $1000 + 1000 + 400 + 40 + 5 + 1 + 1 + 1$, or 2448. As another example, we would parse `MCMLIX` as `(M)(CM)(L)(IX)`, yielding the sum $1000 + 900 + 50 + 9$, or 1959.

To ensure that each positive integer has a unique representation as a Roman numeral, the terms in the sum corresponding to such a numeral must be in descending order of value and must be minimal in number. Thus, for example, in a Roman numeral you would never see substrings such as `IXX` $(9 + 10)$ or `CMM` $(900 + 1000)$, as they would be written as `XIX` $(10 + 9)$ and `MCM` $(1000 + 900)$, respectively, so that the terms are in descending order.

Nor would you ever see substrings such as `LXL` $(50+40)$ or `IVI` $(4+1)$, as they would be written as `XC` (90) and `V` (5), respectively, to minimize the number of terms.

Moreover, the symbols `I`, `X`, and `C` never appear more than three times in a row, as `IIII` would appear instead as `IV`, `XXXX` would be written `XL`, and `CCCC` would be written `CD`.

As for `V`, `L`, and `D`, none of them can appear more than once in a numeral.

Finally, none of the substrings `CMD`, `XCL`, or `IXV` can appear, as they would instead be written `MCD`, `CXL`, and `XIV`, respectively.

Continued on next page ...

Input: The first line contains a positive integer n indicating how many decimal numerals appear on subsequent lines. Each of the next n lines contains a single unsigned decimal numeral that represents a positive (non-zero) integer. (By unsigned is meant that there is no leading '+' or '-' sign.) You may assume that no input value exceeds 20 thousand.

Output: For each decimal numeral provided as input, on one line display it, followed by a space, followed by an equals sign, followed by a space, followed by its equivalent Roman numeral.

```
Sample input:                 Resultant output:
-------------                 -----------------
8
2                             2 = II
4                             4 = IV
24                            24 = XXIV
5003                          5003 = MMMMMIII
1459                          1459 = MCDLIX
2941                          2941 = MMCMXLI
435                           435 = CDXXXV
1597                          1597 = MDXCVII
```

Problem 3: Groups of Anagrams

Two strings are said to be anagrams of one another if, by permuting the characters in one, you can obtain the other. For example, each of `eat`, `tea`, and `ate` is an anagram of the others. For the purposes of this problem, underscore characters are ignored in determining whether two strings are anagrams of each other. Thus, we consider `george_bush` and `he_bugs_gore` to be anagrams, even though they contain differing numbers of underscores.

Develop a program that, given a collection of strings, places them into groups of anagrams.

Input: The first line contains a positive integer n indicating how many strings are in the collection. Each of the next n lines contains a single string composed of only lower case letters (i.e., in the range `a..z`) and underscore characters.

Output: Each line of output should be a list of strings, separated by spaces, where each string is an anagram of all the others. Each list must be maximal, meaning that no pair of strings appearing in different lists can be anagrams of each other. Every string provided as input must appear in exactly one list. Moreover, the lists must be ordered as follows: Within each list, the strings must appear in the same relative order as they appeared in the input. The first string in the first list must be the first string provided as input. The first string in each subsequent list must have appeared in the input data later than the first string in the previous list.

```
Sample input:                 Resultant output:
-------------                 -----------------
20
bored                         bored
ate                           ate eat tea
eat                           listen silent tinsel
listen                        baloney
baloney                       study dusty
tea                           tacit attic
study                         trace cater crate react
tacit                         brag grab
trace                         george_bush he_bugs_gore
brag
cater
crate
silent
george_bush
dusty
tinsel
he_bugs_gore
attic
react
grab
```

Problem 4: Pattern Matching

Develop a program that, given a pattern and a string, determines whether or not the string matches the pattern. The given string will be composed entirely of lower case letters (i.e., any among `a`, `b`, ..., `z`). The pattern will be composed of lower case letters along with the full stop (`.`) and asterisk (`*`) symbols. The symbols in a pattern act as follows:

1. A letter not followed by an asterisk is matched by one occurrence of that letter.
2. A full stop not followed by an asterisk is matched by one occurrence of any letter.
3. A letter followed by an asterisk is matched by zero or more occurrences of that letter.
4. A full stop followed by an asterisk is matched by any string of zero or more letters.

For example, the pattern `.*baa*b` is matched by any string that ends with one or more `a`'s sandwiched between a pair of `b`'s.

Your solution **may not** make use of pattern-matching or regular expression libraries.

Input: The first line contains a positive integer n indicating how many (pattern, string) pairs will appear on subsequent lines. Each of the n lines thereafter contains a pattern and a string, separated by a space. Each pattern and each string is enclosed between a pair of dollar signs (`$`).

Output: For each (pattern, string) pair given as input, the output echoes the pair and reports whether or not (`YES` or `NO`) the string matches the pattern. See the sample output below for the expected format.

```
Sample input:                     Resultant output:
-------------                     -----------------
9
$.*baa*b$ $bab$                   $.*baa*b$ $bab$ : YES
$.*baa*b$ $bbacdbabaaaaaab$       $.*baa*b$ $bbacdbabaaaaaab$ : YES
$.*baa*b$ $cadbbabb$              $.*baa*b$ $cadbbabb$ : NO
$...cat.*dog.*$ $xyzcattledoggie$ $...cat.*dog.*$ $xyzcattledoggie$ : YES
$...cat.*dog.*$ $mycatateadoggie$ $...cat.*dog.*$ $mycatateadoggie$ : NO
$...cat.*dog.*$ $hiscatdog$       $...cat.*dog.*$ $hiscatdog$ : YES
$a*$ $$                           $a*$ $$ : YES
$a*b.$ $ba$                       $a*b.$ $ba$ : YES
$a*bc*$ $aaabaa$                  $a*bc*$ $aaabaa$ : NO
```

Problem 5: Number Split Sums

Given a d-digit positive integer, one can "split" it into r separate pieces, where $r \leq d$, by inserting $r - 1$ boundary markers. For example, using vertical bars as boundary markers, 4752 can be split in any of eight ways:

$$4|7|5|2 \quad 4|7|52 \quad 4|75|2 \quad 47|5|2 \quad 4|752 \quad 47|52 \quad 475|2 \quad 4752$$

In the last split, we inserted no boundary markers, leaving the number intact! In general, a d-digit number can be split in any of 2^{d-1} ways, in total, and it can be split into exactly r pieces (where r satisfies $0 < r \leq d$, of course) in any of

$$\binom{d-1}{r-1} = C(d-1, r-1) = \frac{(d-1)!}{(r-1)!(d-r)!}$$

ways.

Now, for each way of splitting a number, we can sum the resulting pieces to obtain the "split-sum". For example, the split-sum of 47|5|2 is $47 + 5 + 2$, or 54, and the split-sum of 4|752 is $4 + 752$, or 756.

Develop a program that, given two positive integers k and m, finds the largest split-sum that is not greater than m from among all ways of splitting k.

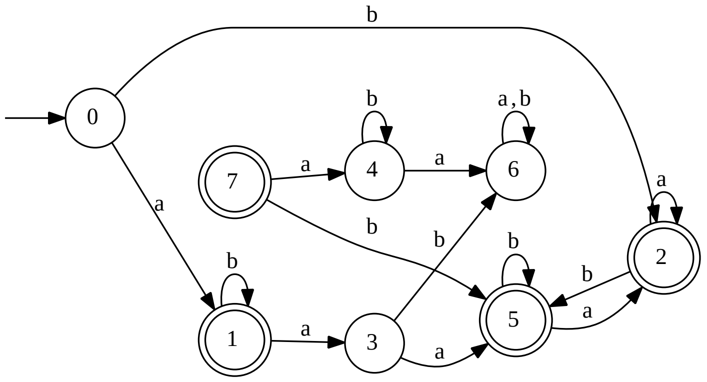
Input: The first line contains a positive integer n indicating how many instances of the problem are described on the succeeding n lines. Each instance of the problem is described on a single line containing two positive integers, k and m, as described in the preceding paragraph.

Output: For each given instance (k, m) of the problem, report the largest split-sum, from among all ways of splitting k, that is not greater than m. If there is no such split-sum, report that. For the proper output format, see the sample output below.

```
Sample input:         Resultant output:
------------          ----------------
4
4752 95               Largest split-sum of 4752 not exceeding 95 is 81
4752 12               Largest split-sum of 4752 not exceeding 12 does not exist
50872 987             Largest split-sum of 50872 not exceeding 987 is 922
50872 213             Largest split-sum of 50872 not exceeding 213 is 139
```

Problem 6: DFA State Reachability

Depicted in the figure below is a *deterministic finite automaton* (DFA) 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 *accepting states*. By convention, we number the states in a DFA from 0 to $m-1$, where m is the number of states. The set of symbols appearing as labels on transitions is called the *alphabet* of the DFA. Notice that M's alphabet is $\{a,b\}$.



A DFA processes an input string by starting in its initial state and then hopping from state to state, following the sequence of transitions whose labels spell out that string. For example, presented with the input string *ababa*, M follows the path

$$0 \xrightarrow{a} 1 \xrightarrow{b} 1 \xrightarrow{a} 3 \xrightarrow{b} 6 \xrightarrow{a} 6$$

and thereby finishes in state 6. In a DFA, each state has, for each symbol of the alphabet, exactly one outgoing transition labeled by that symbol. (This is what makes it "deterministic", as opposed to "nondeterministic".) Therefore, for every state p and every string x, there is exactly one path beginning at p whose labels spell out x.

If processing a string causes a DFA to finish in one of its accepting states, we say that the string is *accepted*. Otherwise, we say that the string is *rejected*. As shown above, processing *ababa* causes M to finish in the non-accepting state 6; hence M rejects that string. On the other hand, when M processes *abaaa* it finishes in the accepting state 2, and so accepts it.

In some cases, it is possible to simplify a DFA (without changing which strings it accepts) by removing and/or by merging some of its states. Specifically, any state that is unreachable from the initial state can be removed, for obvious reasons. As an example of where multiple states can be merged into one, consider those states from which no accepting states are reachable. Any computation arriving at such a state is doomed to end in a rejection of the input string. Thus, all such states can be merged into a single non-accepting “dead” state all of whose outgoing transitions go to itself. Similarly, all states from which *only* accepting states are reachable can be merged into a single “immortal” state!

Develop a program that, given a DFA with alphabet $\{a, b\}$, identifies three subsets of its states:

1. those that are unreachable from the initial state,
2. those from which no accepting states are reachable, and
3. those from which only accepting states are reachable.

Input: The first line contains three nonnegative integers, m, i, and k, which identify, respectively, the number of states in the DFA to be analyzed (which are numbered $0..m-1$), the ID of the initial state, and the number of accepting states.

The second line contains the IDs of the k accepting states. The next m lines describe the outgoing transitions from states 0, 1, ..., $m-1$, in that order, one state per line. On each such line will be two integers identifying the states to which the transitions labeled a and b go, respectively, from the state in question. (The sample input below describes our example DFA M.)

Output: Three lines of output are to be produced, each beginning with an appropriate label (as illustrated in the sample output below) and followed by a list of state IDs in increasing order. Respectively, the three lines identify the states unreachable from the initial state, the (dead) states from which no accepting states are reachable, and the (immortal) states from which only accepting states are reachable. Note that any (or all) of the lists could be of length zero or one.

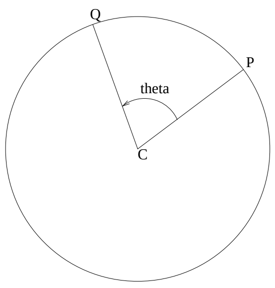
```
Sample Input:           Explanation:
-------------           ------------
8 0 4                   8 states; 0 is initial state; 4 accepting states
1 2 5 7                 IDs of the four accepting states
1 2                     state 0's transitions go to states 1 and 2
3 1                     state 1's transitions go to states 3 and 1
2 5                     etc., etc.
5 6
6 4
2 5
6 6
4 5

Resultant Output:
-----------------
Unreachable: 4 7
Dead:  4 6
Immortal: 2 5
```

Problem 7: Rotating a Point about another Point

Develop a program that, given distinct points C and P on the plane and an angle measure θ, determines the point Q such that line segments $\overline{PC}$ and $\overline{QC}$ are of equal lengths and the angle $\angle PCQ$ has measure θ.

As the figure below illustrates, you can think of C as the center of a circle having radius equal to the length of $\overline{PC}$. Of course, P lies on that circle. If the line segment $\overline{PC}$ were to rotate —like the second hand of a clock, except in the counterclockwise direction— a little less than $\pi/2$ radians, point P would sweep along the arc until it met point Q. (Recall that $\pi \approx 3.14159265$ radians corresponds to 180 degrees.)



Hint: Points on the plane are commonly identified using the cartesian coordinate system, where (x, y) refers to the point that is $|x|$ units in a horizontal direction and $|y|$ units in a vertical direction from the origin. (The signs of x and y distinguish between right and left and between up and down, respectively.)

An alternative way of identifying points on the plane is to use **polar coordinates**, where (r, θ) refers to the point that is r units from the origin and such that θ is the measure of the (counterclockwise) angle formed by the positive x-axis and the line connecting the origin to the point.

Translating between the cartesian and polar coordinate systems is not difficult. The point identified in the polar coordinate system by (r, θ) is identified in the cartesian coordinate system by $(r \cdot \cos\theta, r \cdot \sin\theta)$.

Going in the other direction, (x, y) in the cartesian system translates into $(\sqrt{x^2 + y^2}, \tan^{-1} \frac{y}{x})$ in the polar system, assuming that $x > 0$. If $x < 0$, add π to the angle component. If $x = 0$, then the angle should be either $\frac{\pi}{2}$ or $\frac{3\pi}{2}$, according to whether $y > 0$ or $y < 0$, respectively. If both x and y are zero, the angle value is irrelevant.

Continued on the next page ...

Input: The first line contains a positive integer n indicating how many instances of the problem are described thereafter. Each of the subsequent n lines of input includes five real numbers, the first two of which are the cartesian coordinates of point C, the next two of which are the cartesian coordinates of point P, and the last of which is the angle θ, in radians. You can assume that $0 \leq \theta < 2\pi$ and that P and C are not the same point.

Output: For each instance of the problem provided as input (described by C, P, and θ), the program is to display a message that identifies each of C, P, and *theta* as well as the point Q satisfying the condition that $\overline{PC}$ and $\overline{QC}$ have the same length and the angle $\angle PCQ = \theta$. See the sample output below for the expected output format.

Because calculations upon real numbers on digital computers yield only approximate results, the cartesian coordinates of Q displayed by your program need not correspond exactly to those anticipated by the judges.

```
Sample Input
------------
7
0.0 0.0 2.0 0.0 3.14159265
3.0 3.0 4.0 4.0 1.5708
-4.0 -6.0 0.0 0.0 3.14159265
-2.0 4.0 5.2 3.0 6.0
0.0 0.0 5.0 3.0 1.5
4.5 -1.8 -2.4 5.0 2.5
2.4 6.0 -8.0 -3.5 0.3
```

```
Resultant Output
----------------
Rotating (2.0,0.0) 3.142 radians about (0.0,0.0) yields (-2.0,0.0)
Rotating (4.0,4.0) 1.571 radians about (3.0,3.0) yields (2.0,4.0)
Rotating (0.0,0.0) 3.142 radians about (-4.0,-6.0) yields (-8.0,-12.0)
Rotating (5.2,3.0) 6.000 radians about (-2.0,4.0) yields (4.634,1.028)
Rotating (5.0,3.0) 1.500 radians about (0.0,0.0) yields (-2.639,5.2)
Rotating (-2.4,5.0) 2.500 radians about (4.5,-1.8) yields (5.958,-11.377)
Rotating (-8.0,-3.5) 0.300 radians about (2.4,6.0) yields (-4.728,-6.149)
```

Problem 8: The Game of Dots

A once popular child's game is **Dots**, which is played on a rectangular grid of dots. Two dots are said to be *adjacent* if they lie on the same horizontal or vertical line and no other dot lies between them. A *box* on the grid is a square region bounded by four line segments –two vertical and two horizontal– involving four dots, each of which is adjacent to two of the others.

On each turn, a player chooses two adjacent, but as yet unconnected, dots and draws the line segment that connects them. If this line segment "completes" a box (i.e., the other three line segments bounding the box had already been drawn), the player gets a point for that box and must make the next turn. Otherwise, no points are awarded and the opposite player makes the next turn. (Note that it is possible to win two points in a single turn, because drawing a line segment could complete two boxes simultaneously.)

A game ends when every pair of adjacent dots has been connected (or, equivalently, when every box has been completed). The winner is the player who has accumulated the greater number of points (i.e., completed the greater number of boxes).

Two games are depicted below. On the left is a completed game on a 3×3 grid, while on the right is an unfinished game on a 4×4 grid. Coordinates of the dots on the exterior of the grids are shown. The number labeling each line segment indicates the turn on which it was drawn.

```
   (0,0)  (0,1)  (0,2)
     *------*------*
     |   9  |   3  |
     |      |      |
     |4    8|     7|
     |      |      |
     |   1  |   5  |
(1,0)*------*------*(1,2)
     |      |      |
     |      |      |
     |10  11|     2|
     |      |      |
     |   6  |  12  |
     *------*------*
  (2,0)   (2,1)  (2,2)
```

```
   (0,0)  (0,1)  (0,2)  (0,3)
     *      *------*      *
            |   5  |
            |      |
           3|     7|
            |      |
        1   |   9  |   6
(1,0)*------*------*------*(1,3)
     |      |      |      |
     |      |      |      |
     |15   8|      |2   11|
     |      |      |      |
     |  16  |   4  |  10  |
(2,0)*------*------*------*(2,3)
     |      |
     |      |
     |17  12|
     |      |
     |  14  |         13
     *------*      *------*
   (3,0)  (3,1)  (3,2)  (3,3)
```

Develop a program that, given a description of the sequence of moves that were made during a (possibly unfinished) game pitting **Ann** against **Bill**, reports the current score. Assume that Ann took the first turn. The details of input and output formats are found below.

Input: The first line contains a positive integer p indicating how many games of Dots are described on the succeeding lines. The description of a game begins with a line containing two positive integers, n and k, where n indicates the size of the grid (n dots $\times$ n dots) and k indicates the number of turns made during the (possibly unfinished) game.

The following k lines identify the line segments that were drawn during the game, in order from the first turn through the k-th. A line segment is described by a pair of integers (r, c) (each in the range $0..n-1$) providing the row-and-column coordinates of one of the line segment's endpoints, followed by either `H` or `V` indicating whether the line segment is horizontal or vertical. The endpoint identified in a horizontal segment is the one having the smaller column number and the one identified in a vertical segment is the one with the smaller row number. Thus, for example `2 3 H` (respectively, `2 3 V`) describes the line segment connecting the dots at locations $(2, 3)$ and $(2, 4)$ (respectively, $(2, 3)$ and $(3, 3)$).

The two games described in the sample input below correspond to the games shown in the figure above.

Output: For each game described by the input data, report the score of that game by displaying `Ann` followed by her number of points and then `Bill` followed by his number of points, on one line (with spaces used as separators).

```
Sample input:          Resultant output:
------------           ----------------
2                      Ann 2 Bill 2
3 12                   Ann 2 Bill 3
1 0 H
1 2 V
0 1 H
0 0 V
1 1 H
2 0 H
0 2 V
0 1 V
0 0 H
1 0 V
1 1 V
2 1 H
4 17
1 0 H
1 2 V
0 1 V
2 1 H
0 1 H
```

```
1 2 H
0 2 V
1 1 V
1 1 H
2 2 H
1 3 V
2 1 V
3 2 H
3 0 H
1 0 V
2 0 H
2 0 V
```