UCF "Practice" Local Contest — Aug 23, 2014

How Sweet It Is!

filename: `sweet`
(Difficulty Level: `Easy`)

Dr. Orooji's twins, Mack and Zack, love video games. We will assume that all games are $50. M/Z save all the money they get and, when they have $50 or more, they buy a game and say "Sweet!" out of happiness. If M/Z get a large amount of money at one time (e.g., on their birthday) and they can buy two or more games, they buy two or more games (as many as they can) and say "Totally Sweet!" since they are really in heaven! When M/Z buy game(s), they save the left-over money towards the next purchase.

The Problem:

Given the money (various amounts) M/Z receive, you are to write a program to tell Dr. O when Sweet or Totally Sweet is coming.

The Input:

Each input line contains a positive integer, indicating an amount M/Z are receiving. End-of-data is indicated with a zero.

The Output:

Print the input line numbers and the messages they generate. Follow the format illustrated in Sample Output.

Sample Input:

```
10
20
30
10
90
10
10
30
0
```

Sample Output:

```
Input #3: Sweet!
Input #5: Totally Sweet!
Input #8: Sweet!
```

Wheel of <u>U</u>niversally <u>C</u>opious <u>F</u>ortune

filename: `copious`
(Difficulty Level: `Easy`)

In the game "Wheel of Fortune", the number of letters in a word is given and the contestants guess the letters in the word and, as some letters appear, the contestants guess the word. But, you are a computer scientist and know that you can write a program to search a dictionary and provide candidate words (possible matches) for you.

The Problem:

Given the dictionary and a partially-defined word, you are to determine the candidate words. Note that there may be no candidate words for a given partially-defined word.

The Input:

The first input line contains an integer n *($1 \le n \le 100$)*, indicating the number of words in the dictionary. The dictionary words will be on the following n input lines, one word per line. Each word starts in column 1, contains only lowercase letters, and will be 1-20 letters (inclusive). Assume that the dictionary words are distinct, i.e., no duplicates. The next input line will contain a positive integer m, indicating the number of words to be checked against the dictionary. These words will be on the following m input lines, one word per line. Each word starts in column 1, contains only lowercase letters and hyphens, and will be 1-20 characters (inclusive). A letter in a position indicates that the word must have that letter in that position; a hyphen in a position indicates that any letter can be in that position.

The Output:

At the beginning of each word to be checked, output "`Word #`*`w`*`:`", where *`w`* is the word number (starting from 1). Then print the input word to be checked. Then, on the following output lines, print the candidate words from dictionary that could be a match (print these words in the order they appear in the dictionary). Also print the total number of candidate words (possible matches).

Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

(Sample Input/Output on the next page)

Sample Input:

```
8
at
cat
ali
sat
nerds
coach
couch
ninja
5
co-ch
-at
---
ali
a-c
```

Sample Output:

```
Word #1: co-ch
coach
couch
Total number of candidate words = 2

Word #2: -at
cat
sat
Total number of candidate words = 2

Word #3: ---
cat
ali
sat
Total number of candidate words = 3

Word #4: ali
ali
Total number of candidate words = 1

Word #5: a-c
Total number of candidate words = 0
```

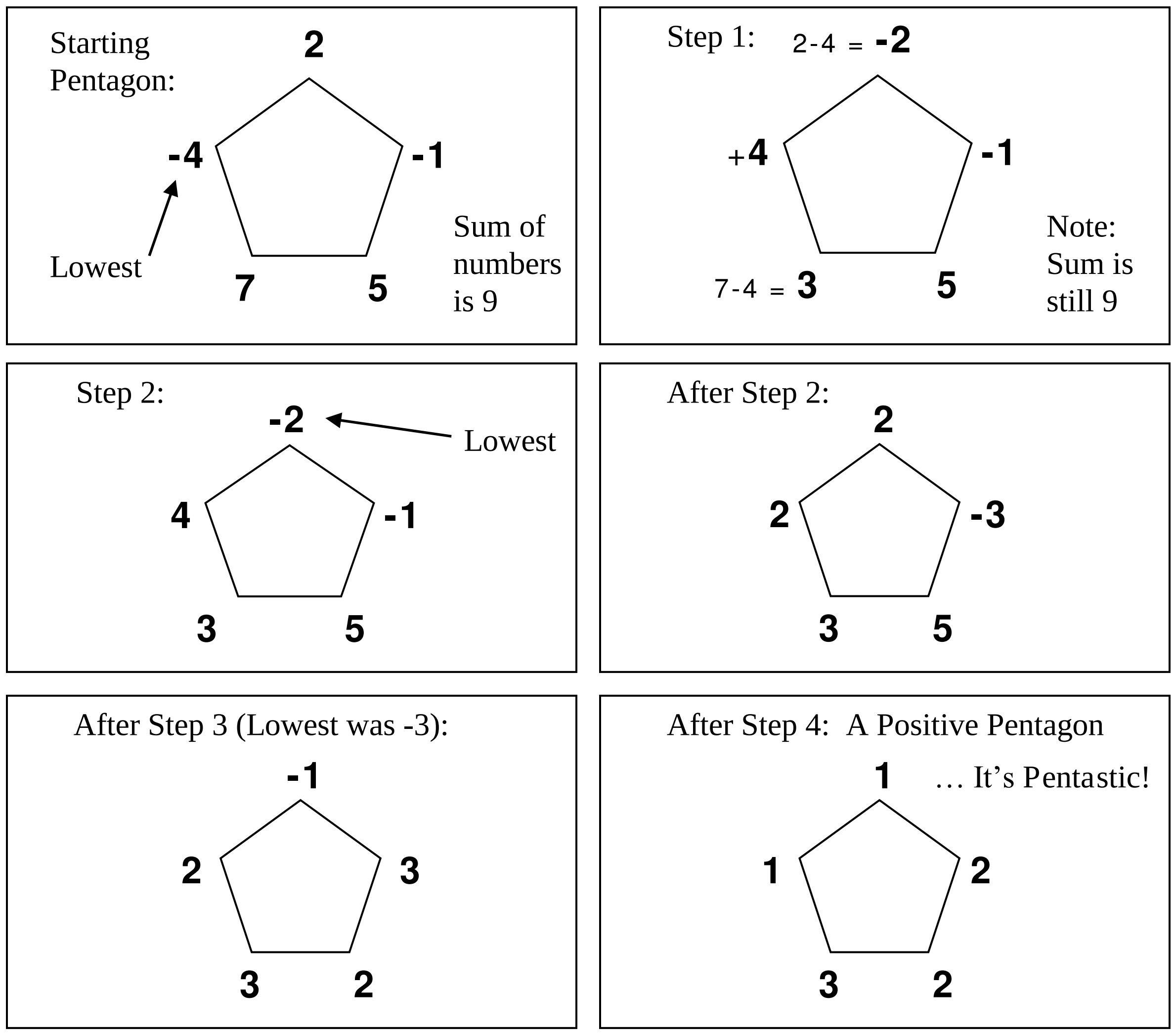
UCF “Practice” Local Contest — Aug 23, 2014

Positively Pentastic!

filename: `pentastic`
(Difficulty Level: `Easy`)

Five random integers are placed at the corners of a pentagon. Typically, some of these numbers will be negative, but their sum is guaranteed to be positive. The goal is to get rid of all the negative numbers through a balanced process of subtraction and negation.

Starting with the lowest of the negative numbers, we negate the number (thus making it positive), and then subtract that value from each of its two neighbors. The sum of the new numbers will remain the same as the original pentagon, so the pentagon is still “balanced.” This process (finding the lowest of the negative numbers, negating it, and subtracting it from its neighbors) is then repeated until all of the numbers are non-negative.



During any step, if the lowest negative number appears at more than one corner, use the one that would be found first, if you started at the top corner and traversed in clockwise direction.

The Problem:

Given the original five numbers at the corners of a pentagon, output the Positive Pentagon that can be created by following this process. You may assume this process will always make a pentagon “pentastic” in at most 1000 steps.

The Input:

The first input line will contain only a single positive integer N, which is the number of pentagons to process. The next N lines will contain pentagon descriptions, one per line. Each pentagon description will consist of exactly 5 integers, which are in the range -999 to 999 (inclusive), and which sum up to a positive number less than 1000.

There will be exactly one space between numbers, and no leading or trailing spaces on the input lines. Positive numbers in the input will not have a leading ‘+’ sign. The numbers are given in a clockwise order around the pentagon, starting from the top. This means that the 1st and 3rd numbers are neighbors of the 2nd number, the 5th and 2nd numbers are neighbors of the 1st number, and so on.

The Output:

For each pentagon in the input, output the message “`Pentagon #`*`p`*`:`”, where *`p`* is the pentagon number (starting from 1). Then, for each pentagon, output the Positive Pentagon that results from applying the process described above. Output the numbers for each corner using the same ordering and method used in the input, with number for the top corner first, and the others following a clockwise order. Output one space between output numbers.

Leave a blank line after the output for each test case. Follow the format illustrated in the Sample Output.

Sample Input:

```
2
2 -1 5 7 -4
99 -1 -1 4 0
```

Sample Output:

```
Pentagon #1:
1 2 2 3 1

Pentagon #2:
97 1 1 2 0
```

UCF "Practice" Local Contest — Aug 23, 2014

Lottery Coprimes

filename: `coprime`
(Difficulty Level: `Easy`)

Lou lost the lottery last week, but he still plans to buy a ticket for this week's draw. He's also buying tickets for all his relatives. They are all mathematicians (who understand probability) and would never buy tickets for themselves. Lou insisted that they each choose their own numbers. When he looked at the numbers, it appeared as though all of his relatives had played a joke on him. They seemed to choose numbers by picking a pair of coprime integers, concatenating them, then splitting the digits up into the number spots on the lottery ticket.

Two integers are called coprimes, or relative primes, if they do not share any positive factors greater than 1. (That's the joke—they are "relative" primes from his relatives.)

These are the lottery numbers from one of Lou's math-loving relative's tickets. The numbers 169 and 7203 are coprime.

The Problem:

Given a list of concatenated digits from a lottery ticket, determine whether this list can be split into two numbers which are coprimes. Note that the digits can not be reordered.

The Input:

The first line of input will contain only a single positive integer N, which is the number of lottery tickets to evaluate. Each of the next N input lines will contain 3 to 8 digits, representing a single ticket. Neither the first digit nor the last digit will ever be zero, and there will never be two consecutive zeroes. There will be no spaces or other characters on these lines, other than digits.

The Output:

For each ticket in the input, output "`Ticket #`*`T`*`:`", where *`T`* is the ticket number (starting at 1). On the next line, output the two coprimes found by splitting the digits for that ticket. Separate the numbers by at least one space. If no coprimes are found, output the message "`Not relative`" instead, since the numbers were probably not picked by any of Lou's relatives. If there are multiple possible ways to split the digits into coprimes, use the one in which the first number is the lowest. If the split occurs before a zero digit, you may omit this leading zero when outputting the second number, i.e., you can output a number with leading zeroes with or without those zeroes.

Leave a blank line after the output for each ticket. Follow the format shown in Sample Output.

Sample Input:

```
4
47108
222
1697203
7203217
```

Sample Output:

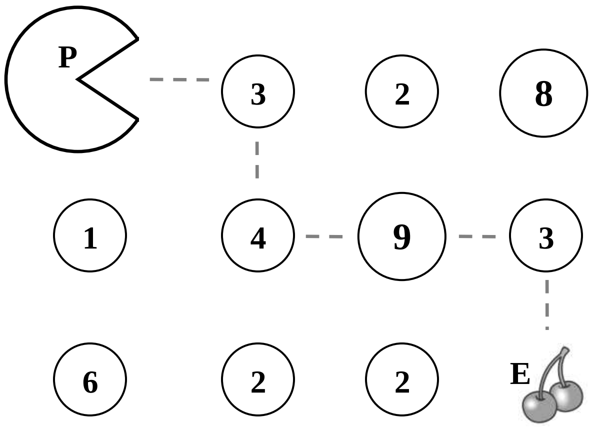
```
Ticket #1:
47 108

Ticket #2:
Not relative

Ticket #3:
1 697203

Ticket #4:
72 3217
```

Pac Man for your New Phone

filename: `pacman`
(Difficulty Level: `Medium`)

You are writing an app for your friend's new Phone, the newPhone. Since you grew up on Pac Man, you want to write a simplified version of the game. In this game, the board is a rectangular grid and Pac Man starts at the upper left-hand corner. His goal is to get to the lower right-hand corner. He always moves one square to the right or one square down. Each square he goes to has a "goody" that's worth a particular amount of points. Your score is simply the sum of the scores of the goodies in each square you have visited.

For example, if the game board looks like this (P indicates Pac Man's starting location, and E indicates his ending location):



then Pac Man's optimal strategy would be to move right, down, right, right again, then down to yield a score of 3 + 4 + 9 + 3 = 19.

The Problem:

Given a game board, determine the maximum possible score for Pac Man.

The Input:

There will be multiple game boards in the input file. The first input line contains a positive integer n, indicating the number of game boards to be processed. The first line of each game board will contain two positive integers, r ($0 < r < 100$) and c ($1 < c < 100$), representing the number of rows and columns for this game board. (The example above has three rows and four columns.) Each of the following r input lines will contain c tokens, representing the contents of that row. The first item on the first of these lines will be the character 'P', representing Pac Man's original location and the last item on the last line will be the character 'E', representing

Pac Man's goal location. The rest of the items will be positive integers less than 1000. Items will be separated by a single space on each line.

The Output:

At the beginning of each test case, output "`Game Board #`*`g`*`:`", where *`g`* is the input board number (starting from 1). For each game board, simply print out the maximum possible score for the game.

Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

Sample Input:

```
2
3 4
P 3 2 8
1 4 9 3
6 2 2 E
2 2
P 5
401 E
```

Sample Output:

```
Game Board #1: 19

Game Board #2: 401
```

UCF “Practice” Local Contest — Aug 23, 2014

Sierpiński Triangle

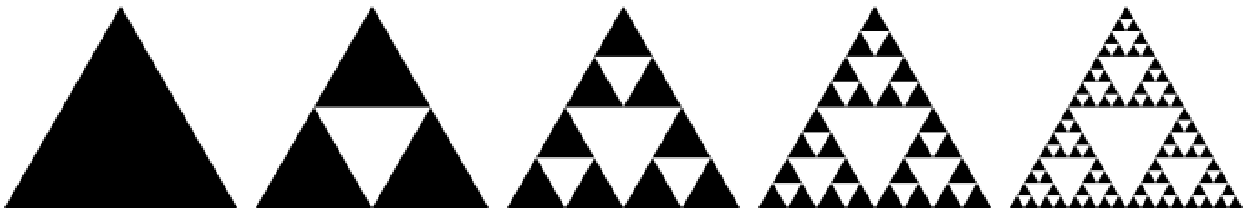
filename: `triangle`
(Difficulty Level: `Medium`)

The Sierpiński triangle is a beautiful fractal found in mathematics. As with many fractal patterns, it is constructed by starting with a given shape, applying a function to that shape, then applying the same function to the resulting shapes, and so on. In theory, this function is applied infinitely many times, but in practice it usually stops after a given number of applications since the resulting shapes get too small to be noticeable.

The Sierpiński triangle is created with the following steps:

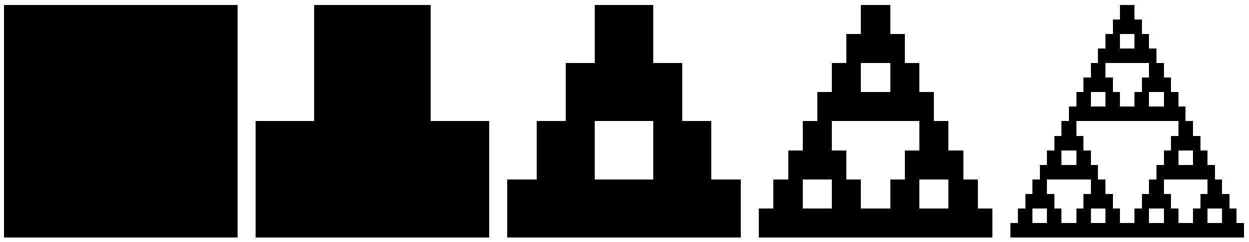
1) Start with an equilateral triangle with the base parallel to the horizontal axis.
2) Create an upside down triangle with half the height and width of the original triangle and cut this pattern out of the center of the original triangle, leaving 3 equilateral triangles.
3) Repeat step 2 on each of the newly formed triangles.

The following image (from Wikipedia) shows triangles of level 1 through 5, respectively:



Sierpiński Triangle

Interestingly enough, there are other shapes to which you can apply a similar pattern that result in close approximations of a Sierpiński triangle. For example, starting with a square, you can remove rectangles of half the height and a quarter of the width from the upper left and right corners, resulting in three new squares. The following image (from Wikipedia) shows approximate triangles of level 1 through 5, respectively:



Approximate Sierpiński Triangle

The Problem:

Given a level, draw an ASCII version of the approximate Sierpiński triangle.

The Input:

The input will begin with a positive integer T representing the number of triangles to draw. This will be followed by T lines each with a positive integer K ≤ 10 representing the level of the triangle you are to draw.

The Output:

For each triangle, begin with the line "`Triangle #`*`S`*`:`" with S starting at 1. Then output the ASCII version of the approximate level K triangle, using the character 'X' to represent the drawn pattern and spaces elsewhere. You should scale the image so that the smallest square regions are 2x2. The bottom left point of the image should be in column 1. See the sample output for clarification. Follow each triangle with a blank line.

Sample Input:

```
2
1
4
```

Sample Output:

```
Triangle #1:
XX
XX

Triangle #2:
       XX
       XX
      XXXX
      XXXX
     XX  XX
     XX  XX
    XXXXXXXX
    XXXXXXXX
   XX      XX
   XX      XX
  XXXX    XXXX
  XXXX    XXXX
 XX  XX  XX  XX
 XX  XX  XX  XX
XXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
```

```
Triangle #1:
XX
XX

Triangle #2:
.......XX.......
.......XX.......
......XXXX......
......XXXX......
.....XX..XX.....
.....XX..XX.....
....XXXXXXXX....
....XXXXXXXX....
...XX......XX...
...XX......XX...
..XXXX....XXXX..
..XXXX....XXXX..
.XX..XX..XX..XX.
.XX..XX..XX..XX.
XXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
```

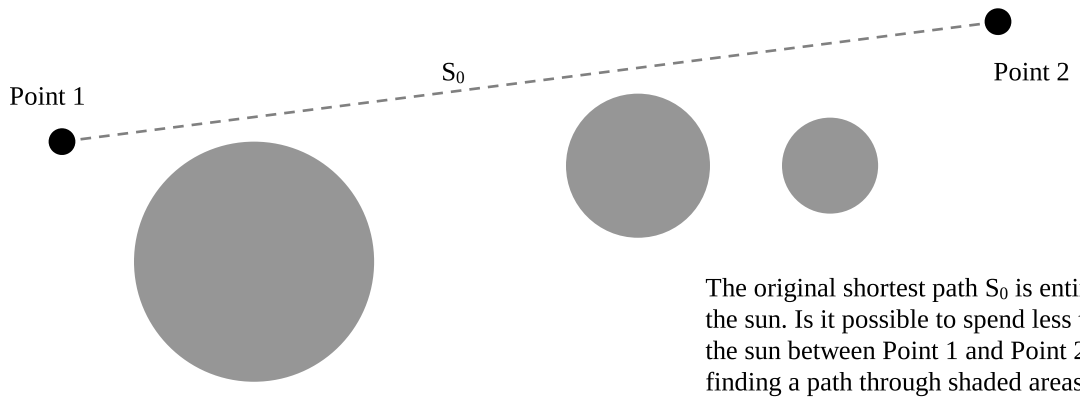
Note: The first column shows what you should actually output. The second column shows a '.' where all spaces occur for clarification. Do not submit a solution with periods for spaces – you will receive wrong answer for this.

UCF “Practice” Local Contest — Aug 23, 2014

Walking in the Sun

filename: `sunwalk`
(Difficulty Level: `Medium`)

You have calculated all the shortest distances between points on campus, but have found that this strategy is not optimal in the summer. Some of these shortest paths are in the sun, and it’s far more annoying to be sweating than to walk a few extra steps in the shade. Your goal is to recalculate shortest distances in the sun on campus, given information about where shade is located.


Point 1
S_0
Point 2
The original shortest path S_0 is entirely in the sun. Is it possible to spend less time in the sun between Point 1 and Point 2 by finding a path through shaded areas?

The Problem:

We simplify the problem by specifying all locations on campus by 2-D Cartesian coordinates, and by specifying all areas of shade as circular areas with a given center and radius. We also assume that any straight line between two Cartesian points can be walked, i.e., there are no objects blocking any possible straight line paths. Instead of calculating the shortest distance between locations on campus, your goal will be to calculate the least amount of walking that needs to be done in the sun to get between those two points.

The Input:

There will be multiple test cases in the input file. The first input line contains a positive integer, indicating the number of test cases.

The first line of each test case, s *($0 \leq s < 50$)*, contains the number of locations of shade on campus for that test case. Each of the next s lines contains the x-coordinate, y-coordinate, and radius, respectively, of a shade location, separated by spaces.

The next line of each test case, q ($0 < q < 100$), contains the number of distance queries for the test case. Each of the next q lines will contain four numbers representing two points on campus, in the form x_1 y_1 x_2 y_2, where (x_1,y_1) are the coordinates of the first point, and (x_2,y_2) are the coordinates of the second point.

All x-coordinates, y-coordinates, and radii will be real numbers between -1000 and 1000, inclusive.

The Output:

At the beginning of each test case, output “`Campus #c:`”, where *c* is the test case number (starting from 1) on the first line.

For the subsequent q lines, begin the output with the header “ `Path #p:`”, where p is the distance query number (starting from 1) for that case. Follow each of these headers with the statement of the form: “`Shortest sun distance is D.`”, where D is the desired shortest sun distance rounded to one decimal place. To clarify “rounded to one decimal place”: the output for 1.74 should be 1.7, for 1.75 should be 1.8, and for 1.76 should be 1.8.

Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

Sample Input:

```
2
3
5.2 3.3 4.7
-8.8 -6.1 3.1
18.5 6.1 2.2
6
1.1 20.2 6.1 18.1
1.1 20.2 3.3 -2.5
0.4 -2.7 3.3 -2.5
6.1 18.1 -5.5 -9.2
3.3 -2.5 0.4 -2.7
1.1 20.2 0.4 -2.7
1
0.0 0.0 20.0
1
3.1 2.2 7.7 8.1
```

Sample Output:

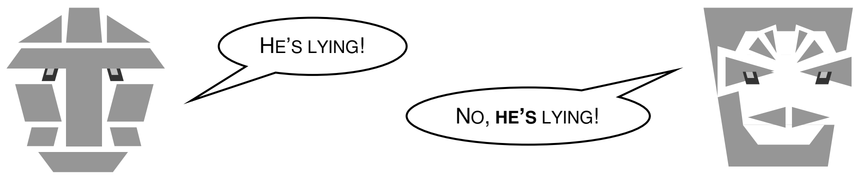
```
Campus #1:
  Path #1: Shortest sun distance is 5.4.
  Path #2: Shortest sun distance is 14.1.
  Path #3: Shortest sun distance is 2.9.
  Path #4: Shortest sun distance is 20.6.
  Path #5: Shortest sun distance is 2.9.
  Path #6: Shortest sun distance is 15.7.

Campus #2:
  Path #1: Shortest sun distance is 0.0.
```

UCF “Practice” Local Contest — Aug 23, 2014

Tautobots and Contradicticons

filename: `logotron`
(Difficulty Level: `Medium`)

The planet Logotron is inhabited by two types of robots – the Tautobots and the Contradicticons. The Tautobots are programmed to always tell the truth, while the Contradicticons must always lie. Unfortunately, there is no simple way for outsiders to tell them apart, which often causes problems.



The Problem:

You are given a set of statements made by a group of robots. Every robot knows the type of every other robot, as well as itself. Each statement consists of an author (the robot that made the statement), a subject (the robot the statement is about), and the type of the subject (Tautobot or Contradicticon). For example, “Robot 5 says that Robot 2 is a Tautobot” is a valid statement. Note that if Robot 5 is a Contradicticon, then Robot 2 must also be a Contradicticon, since Robot 5 lied.

Given M statements made by N robots, you must find the number of distinct ways to assign a type to each robot, consistent with the statements. Two assignments are considered to be different if at least one robot is a Tautobot in one and a Contradicticon in the other.

The Input:

The first input line contains a positive integer T, indicating the number of test cases to be processed. This will be followed by T test cases.

Each test case is formatted as follows. The first line consists of the numbers N and M ($1 \leq N \leq 15$, $0 \leq M \leq 100$). This is followed by M lines, each of which represents a statement by one of the robots. A statement is formatted as “A S X”. Here A and S are integers between 1 and N (inclusive) representing the author of the statement and its subject respectively (assume that A and S will be different robots). X will be one of the characters 'T' (for Tautobot) or 'C' (for Contradicticon).

Assume that a robot will not contradict itself (making a statement and then making the opposite of that statement) but different robots may contradict each other (in that case, there is no possible answer, i.e., zero assignments). Also assume that we will not have the same statement repeated by a robot several times, i.e., no two input statements will be completely identical.

The Output:

For each test case, output a single line, formatted as: "Case `#`*`t`*`:`", where *`t`* is the test case number (starting from 1), a single space, and then the number of distinct assignments that can be made for that case. Follow the format illustrated in Sample Output.

Sample Input:

```
3
3 2
1 2 T
2 3 C
4 2
1 2 T
2 3 C
2 0
```

Sample Output:

```
Case #1: 2
Case #2: 4
Case #3: 4
```

UCF “Practice” Local Contest — Aug 23, 2014

A Constant Struggle

filename: `constant`
(Difficulty Level: `Hard`)

Your math teacher Xavier Guha has given your class an extra credit assignment to work on. The problem he gives is as follows: Given a linear equation of the form

$$c_1x_1 + c_2x_2 + c_3x_3 + c_4x_4 + c_5x_5 + c_6x_6 + c_7x_7 + c_8x_8 = N$$

with $c_1, \ldots, c_8, N$ given, he asks you to give the unique solution x (a vector of length 8) to the equation, where each x_i is a non-negative integer. Of course, being a clever student, you realize that depending on the values of c and N, there may be no solution or there may be several solutions. After class, you approach him and inform him of the mistake, but he is stubborn and will not have any of your nonsense.

Having taken several programming classes from his brother, you decide to prove your teacher wrong[1] by writing a program to determine how many solutions the equation has.

The Problem:

Given $c_1, \ldots, c_8$ and N, determine how many unique solutions the equation has. Two solutions p and q are considered unique (different) if there exists some i for which $p_i \neq q_i$. Since you may get this assignment again in the future, your program should be able to solve several instances of the equation.

The Input:

Input will begin with a positive integer T denoting the number of equations to solve. This will be followed by T lines, each containing an instance of the equation to solve. Each instance will be described by 9 space separated positive integers, all $\leq$ 100. The first 8 numbers represent $c_1, \ldots, c_8$, and the 9th number represents N.

The Output:

For each equation, output a single line “`Equation #`*`E`*`: `*`S`*” where E is the equation number beginning with 1 and S is the number of unique solutions to the equation. It is guaranteed that the value of S will fit in a 64-bit signed integer.

(Sample Input/Output on the next page)

[1] This (proving your teacher is wrong) is generally a bad idea. The author of this problem absolves himself of any (liability for) ill will created between you and your teacher, as well as any detrimental effect this may have on your final course grade. Solve at your own risk.

Sample Input:

```
5
1 1 1 1 1 1 1 1 1
1 2 3 4 5 6 7 8 9
2 4 6 8 10 12 14 16 29
2 4 6 8 10 12 14 17 17
1 1 1 1 1 1 1 1 100
```

Sample Output:

```
Equation #1: 8
Equation #2: 29
Equation #3: 0
Equation #4: 1
Equation #5: 26075972546
```

UCF “Practice” Local Contest — Aug 23, 2014

Polygon Restoration

filename: `polygon`
(Difficulty Level: `Hard`)

A rectangular polygon is a closed figure with all vertices at points with integer coordinates in the XY-plane, and whose edges are all either horizontal or vertical. The vertices are all distinct, and no two edges intersect, except for neighboring edges intersecting at their common vertex. For the purposes of this problem, every horizontal edge will be adjacent to a vertical edge, and vice versa, so all angles are either 90 or 270 degrees.

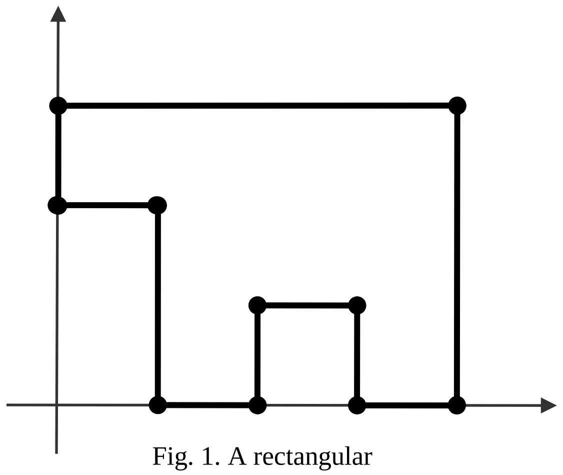


Fig. 1. A rectangular polygon in the XY-plane

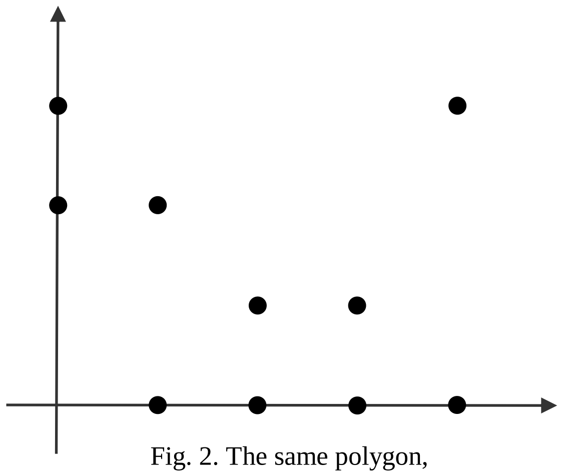


Fig. 2. The same polygon, with all of its edges erased

The Problem:

Dr. O is an avid collector of polygons. He takes particular pride in his collection of pencil-rendered rectangular polygons from the early 19th century. Unfortunately, some sneaky vandal has broken into his collection and erased all the edges, leaving only the vertices of each polygon. You have been called in to try to restore Dr. O's collection to its former glory by redrawing the edges of the rectangular polygons.

The Input:

There will be multiple polygons in the input file. The first line of every polygon description will be an integer N ($4 \leq N \leq 50$), the number of vertices of the polygon. Each of the next N input lines will contain two integers, giving the coordinates of a vertex of a polygon in the form “X Y” ($-10000 \leq X, Y \leq 10000$). Note that these vertices are given in arbitrary order. All vertices will be distinct.

End of input will be indicated by a case with N = 0. This case should not be processed.

The Output:

For each test case, output a single line, formatted as: "Polygon `#t:`", where `t` is the test case number (starting from 1), a single space, followed by the vertices of the polygon in counterclockwise order (with a single space separating vertices). Points should be referred to by their number in the order they were given in the input, the first input being vertex 1 (see Sample Output for clarification). The list must start from the vertex with minimum y-coordinate. If there are multiple points with minimum y-coordinate, use the one with minimum x-coordinate. It is guaranteed that a closed rectangular polygon can always be constructed from the given data.

Sample Input:

```
4
0 0
1 1
0 1
1 0
10
0 3
4 0
1 2
2 1
3 0
4 3
0 2
3 1
2 0
1 0
0
```

Sample Output:

```
Polygon #1: 1 4 2 3
Polygon #2: 10 9 4 8 5 2 6 1 7 3
```