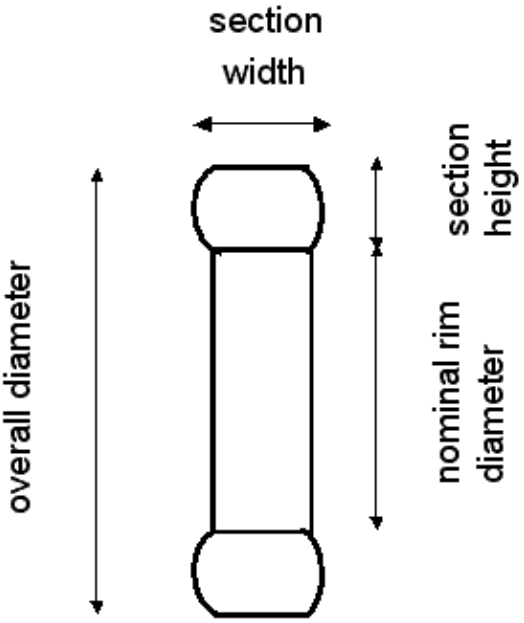
Problem A: Tire Dimensions

Input: `tire.in`
Output: `tire.out`

Given the tire descriptor typically found on the sidewall of a passenger or light truck tire, you will calculate the tire's overall circumference. Each line of the sample input contains an example of a tire descriptor. The following diagram illustrates some of the terminology:



The tire descriptor contains the following items of information:

1. One or two upper case letters to specify the type of tire. For our purposes, the tire types are "P" (passenger), "LT" (light truck), and "T" (temporary spare tire).
2. The *section width* (of an inflated tire) in millimeters. This number is followed by a slash.
3. The ratio of the section height to the section width, expressed as a percentage. For example, the ratio 75 means that the section height of an inflated tire is 75% of its section width.
4. Information about the *construction* of the tire (one upper−case letter), optionally preceded by the *speed symbol* (also one upper−case letter). In the first, second, and fourth lines of the sample output, the tire construction is specified by "R", which means "radial". In the third line, it is "D", which means "diagonal". In the second line, "R" is preceded by the optional speed symbol "H".
5. The nominal rim diameter in inches. It is called "nominal" because it does not include the rim's flanges.

The *overall circumference* (the goal of your calculations) is based on the *overall diameter*, which is the diameter of an inflated tire at the outermost surface of the tread.

Input

The input will consist of one or more lines, terminated by end−of−file. Each line of the input will contain one tire descriptor, as discussed in the preceding paragraphs. All numerical quantities will be positive integers. Exactly one blank space will separate consecutive items (including the slash) on the input line.

Output

For each line of input, the program will produce exactly one line of output, consisting of: the input line, followed by a colon, one blank space, and the overall circumference, expressed in centimeters, rounded to the nearest integer. Note that 1 centimeter equals 10 millimeters, and 1 inch equals 2.54 centimeters.

Sample input

```
P 195 / 75 R 14
P 205 / 60 H R 15
T 115 / 70 D 15
LT 245 / 75 R 16
```

Output for sample input

```
P 195 / 75 R 14: 204
P 205 / 60 H R 15: 197
T 115 / 70 D 15: 170
LT 245 / 75 R 16: 243
```

Problem B: Random Walk

Input: `randomwalk.in`
Output: `randomwalk.out`

Random walks are used to model a wide range of phenomena, from Brownian motion to gambling. For example, a gambler who bets on heads or tails on a coin toss wins or loses his bet each turn. The amount of money the gambler has throughout the game is a random walk. Although the bets in each turn may be different, it is easy to see that the gambler wins the maximum amount of money if he wins every turn. Similarly, he loses the maximum amount if he loses every turn.

We consider the following two−dimensional variation of the random walk. We are given n two−dimensional nonzero vectors $v_i = (x_i, y_i)$, no two of which are parallel. In step i, a coin is flipped. If it is heads, we move x_i meters in the x direction and y_i meters in the y direction. If it is tails, we move $-x_i$ and $-y_i$ meters in the x and y directions.

We are interested in computing the maximum distance we can be away from our starting point. This is easy in one−dimension, but it is not so easy in the two−dimensional version.

Input

The input consists of a number of test cases. Each test case starts with a line containing the integer n, which is at most 100. Each of the next n lines gives a pair of integers x_i and y_i specifying v_i. The coordinates are less than 1000 in magnitude. The end of input is specified by $n = 0$.

Output

For each test case, print on a line the maximum distance we can be away from the starting point, in the format shown below. Output the answer to 3 decimal places.

Sample input

```
3
1 1
0 1
−1 1
2
4 0
1 1
7
1 3
−2 −7
7 8
−2 9
−7 −3
4 −3
−2 −2
0
```

Output for sample input

```
Maximum distance = 3.000 meters.
Maximum distance = 5.099 meters.
Maximum distance = 37.336 meters.
```

Problem C: Paint Mix

Input: `paint.in`
Output: `paint.out`

You are given two large pails. One of them (known as the black pail) contains *B* gallons of black paint. The other one (known as the white pail) contains *W* gallons of white paint. You will go through a number of ***iterations*** of pouring paint first from the black pail into the white pail, then from the white pail into the black pail. More specifically, in each iteration you first pour *C* cups of paint from the black pail into the white pail (and thoroughly mix the paint in the white pail), then pour *C* cups of paint from the white pail back into the black pail (and thoroughly mix the paint in the black pail). *B*, *W*, and *C* are positive integers; each of *B* and *W* is less than or equal to 50, and $C < 16 * B$ (recall that 1 gallon equals 16 cups). The white pail's capacity is at least *B*+*W*.

As you perform many successive iterations, the ratio of black paint to white paint in each pail will approach *B/W*. Although these ratios will never actually be equal to *B/W* one can ask: how many iterations are needed to make sure that the black−to−white paint ratio in *each* of the two pails differs from *B/W* by less than a certain tolerance. We define the tolerance to be 0.00001.

Input

The input consists of a number of lines. Each line contains input for one instance of the problem: three positive integers representing the values for *B*, *W*, and *C*, as described above. The input is terminated with a line where $B = W = C = 0$.

Output

Print one line of output for each instance. Each line of output will contain one positive integer: the smallest number of iterations required such that the black−to−white paint ratio in *each* of the two pails differs from *B/W* by less than the tolerance value.

Sample input

```
2 1 1
2 1 4
3 20 7
0 0 0
```

Output for sample input

```
145
38
66
```

Problem D: Open and Close

Input: `openclose.in`
Output: `openclose.out`

Morphological operations are tools that are used for extracting image components to represent and describe region shapes. Two common morphological operations are open and close. Before we define these operations, we first have to define how images are represented.

Given a binary image *A* with *M* rows and *N* columns, we can represent *A* as a set of the coordinates *(r, c)* (1 <= *r* <= *M*, 1 <= *c* <= *N*) such that the pixel at the specified coordinates is 1. The coordinates of the top−left corner are (1, 1). We are also given a binary image *B* (called the *structuring element*) with *2S+1* rows and columns. The structuring element can be represented as a set as before, except that *(−S,−S)* are the coordinates of the pixel at the top−left corner.

Two operations important in morphological image processing are dilation and erosion. Dilation of an image *A* by the structuring element *B* is defined by:

```
A ^ B = { a + b | a in A, b in B } intersect Z
```

where the addition of coordinates is defined componentwise, and *Z* is the set of coordinates *(i,j)* with 1 <= *i* <= *M* and 1 <= *j* <= *N*. Similarly, erosion of *A* by *B* is defined by:

```
A v B = { w | w + b in A for every b in B }
```

With these two operations defined, the opening of *A* by *B* is defined by

```
A o B = (A v B) ^ B
```

and the closing of *A* by *B* is defined by

```
A . B = (A ^ B) v B
```

Roughly speaking, the opening operation is used to remove small details while preserving the overall shape. The closing operation is used to fill in gaps while preserving the overall shape.

Input

The input consists of a number of cases. Each case starts with a line containing the integers *M*, *N*, and *S* separated by spaces (10 <= *M*, *N* <= 256, 1 <= *S* <= 4). The next *M* lines contain the rows of the image *A* specified by *N* characters that are '.' (0) or '*' (1). The next *2S+1* lines specify the structuring element *B* in a similar manner. The input is terminated by *M = N = S = 0*.

Output

For each case, print the case number followed by a blank line. Then display the result of `A o B` followed by a blank line, followed by the result of `A . B`. The format of the resulting images is the same as those of the input images. Separate the output for different cases by a line consisting of 75 equal signs (=).

Sample input

```
10 12 1
............
..*.........
.***...***..
.***...***..
.*********..
.**********.
.*********..
.***...*.*..
............
............
***
***
***
10 12 1
............
..*.........
.***...***..
.***...***..
.*********..
.**********.
.*********..
.***...*.*..
............
............
.*.
***
.*.
0 0 0
```

Output for sample input

```
Case 1

............
............
.***...***..
.***...***..
.*********..
.*********..
.*********..
.***........
............
............

............
..*.........
.***...***..
.***...***..
.*********..
.**********.
.*********..
.***...***..
............
............
================================================================================
Case 2
```

```
............
..*.........
.***....*...
.***...***..
.*********..
.**********.
.*********..
..**...*....
............
............

............
..*.........
.***...***..
.****.****..
.*********..
.**********.
.*********..
.***...*.*..
............
............
```

Problem F: Smallest Difference

Input: `mindiff.in`
Output: `mindiff.out`

Given a number of distinct decimal digits, you can form one integer by choosing a non−empty subset of these digits and writing them in some order. The remaining digits can be written down in some order to form a second integer. Unless the resulting integer is 0, the integer may not start with the digit 0.

For example, if you are given the digits 0, 1, 2, 4, 6 and 7, you can write the pair of integers 10 and 2467. Of course, there are many ways to form such pairs of integers: 210 and 764, 204 and 176, etc. The absolute value of the difference between the integers in the last pair is 28, and it turns out that no other pair formed by the rules above can achieve a smaller difference.

Input

The first line of input contains the number of cases to follow. For each case, there is one line of input containing at least two but no more than 10 decimal digits. (The decimal digits are 0, 1, ..., 9.) No digit appears more than once in one line of the input. The digits will appear in increasing order, separated by exactly one blank space.

Output

For each test case, write on a single line the smallest absolute difference of two integers that can be written from the given digits as described by the rules above.

Sample input

```
1
0 1 2 4 6 7
```

Output for sample input

```
28
```

Problem G: Faulty Odometer

Input: `odometer.in`
Output: `odometer.out`

You are given a car odometer which displays the miles traveled as an integer. The odometer has a defect, however: it proceeds from the digit 3 to the digit 5, always skipping over the digit 4. This defect shows up in all positions (the one's, the ten's, the hundred's, etc.). For example, if the odometer displays 15339 and the car travels one mile, odometer reading changes to 15350 (instead of 15340).

Input

Each line of input contains a positive integer in the range 1..999999999 which represents an odometer reading. (Leading zeros will not appear in the input.) The end of input is indicated by a line containing a single 0. You may assume that no odometer reading will contain the digit 4.

Output

Each line of input will produce exactly one line of output, which will contain: the odometer reading from the input, a colon, one blank space, and the actual number of miles traveled by the car.

Sample input

```
13
15
2003
2005
239
250
1399
1500
999999
0
```

Output for sample input

```
13: 12
15: 13
2003: 1461
2005: 1462
239: 197
250: 198
1399: 1052
1500: 1053
999999: 531440
```

Problem I: Suit Distribution

Input: `suit.in`
Output: `suit.out`

Bridge is a 4−player (two teams of two) card game with many complicated conventions that even experienced players have difficulty keeping track of. Fortunately, we are not interested in these conventions for this problem. In fact, it is not even important if you understand how to play the game.

What is important to know is that the way the cards are distributed among your two opponents often determine whether you will be successful in your game. For example, suppose you and your partner hold 8 spades. The remaining 5 spades are held by your opponents (since there are 13 cards in each suit) and can be distributed in the following ways: 0−5, 1−4, 2−3. Notice that a 0−5 "split" can be realized in two ways−−−opponent 1 has no spade and opponent 2 has 5 spades, or vice versa.

Good bridge players know that the best line of play often depends on the distribution. Sometimes good players can "read their opponents' cards" and determine the distribution, but sometimes even good players have to guess. In those cases, knowing the probability of the different distributions would be useful in making an educated guess.

You can assume that the 52 cards in a deck are dealt out randomly to 4 players, so that every player has 13 cards, and that you know which 26 cards your team holds.

Input

The input consists of a number of cases. Each case consists of two integers *a*, *b* (0 <= *a*, *b* <= 13, *a* + *b* <= 13). The input is terminated by *a* = *b* = *−1*.

Output

For each case, print the probability of a split of *a*+*b* cards so that one opponent has *a* cards and the other has *b* cards in the format as shown in the sample output. You may assume that the remaining cards in the suit are held by you and your partner. Output the probabilities to 8 decimal places.

Sample input

```
2 2
3 3
4 2
−1 −1
```

Output for sample input

```
2−2 split: 0.40695652
3−3 split: 0.35527950
4−2 split: 0.48447205
```