



# Problem A

## Find a Minor

Input: minor.in

In a graph  $G$ , *contraction* of an edge  $e$  with endpoints  $u, v$  is the replacement of  $u$  and  $v$  with a single vertex such that edges incident to the new vertex are the edges other than  $e$  that were incident with  $u$  or  $v$ . The resulting graph has one less edge than  $G$ . A graph  $H$  is a *minor* of a graph  $G$  if a copy of  $H$  can be obtained from  $G$  via repeated edge deletion, edge contraction and isolated node deletion.

Minors play an important role in graph theory. For example, every non-planar graph contains either the graph  $K_{3,3}$  (i.e., the complete bipartite graph on two sets of three vertices) or the complete graph  $K_5$  as a graph minor.

Write a program to find a graph minor  $K_{n,m}$  or  $K_n$  in an *undirected connected simple graph*.

### Input

The input consists of several test cases. The first line of each case contains an integers  $V$  ( $3 \leq V \leq 12$ ), the number of vertices in the graph, followed by a string in format " $K_n$ " or " $K_{n,m}$ " ( $1 \leq n, m \leq V$ ), the graph minor you're finding. The following  $V$  lines contain the adjacency matrix of the graph (1 means directly connected, 0 means not directly connected). The diagonal elements of the matrix will always be 0, and the element in row  $i$  column  $j$  is always equal to the element in row  $j$  column  $i$ . The last test case is followed by a single zero, which should not be processed.

### Output

For each test case, print the case number and the string "Found" or "Not found".

### Sample Input

### Output for the Sample Input

```
5 K2,2
0 1 1 1 1
1 0 0 0 0
1 0 0 0 0
1 0 0 0 0
1 0 0 0 0
4 K3
0 1 0 1
1 0 1 0
0 1 0 1
1 0 1 0
4 K2,2
0 1 0 1
1 0 1 1
0 1 0 1
1 1 1 0
5 K2,2
0 1 0 0 1
1 0 0 0 1
0 0 0 1 1
0 0 1 0 1
1 1 1 1 0
5 K4
0 1 0 1 1
1 0 1 1 0
0 1 0 1 1
1 1 1 0 1
1 0 1 1 0
0
```

```
Case 1: Not found
Case 2: Found
Case 3: Found
Case 4: Not found
Case 5: Found
```

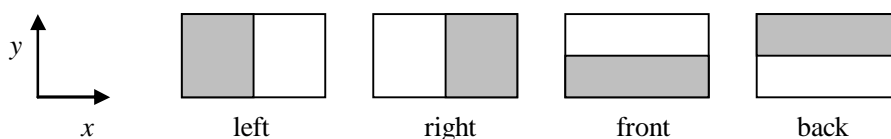


## Problem B

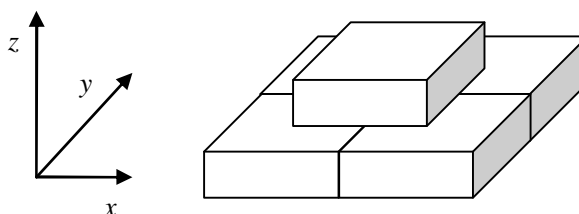
### Bricks

Input: bricks.in

You have some equal-height bricks and want to pile them into a two-layered object. To ensure the stability of the object, no *full* half-blocks can be over empty space. There are totally four half-blocks for each brick, shown below.



The following figure shows an object consisting of four  $2 \times 2 \times h$  bricks in the lower layer, one  $2 \times 2 \times h$  brick in the upper layer. Each half-block does not include its boundary, so you're not allowed to leave only one brick, since the upper brick would have two half-blocks hanging over empty space (though you may argue that the upper brick may stand still if you don't touch it). In this case, at most 2 bricks could be removed.



Each time, you can remove exactly one brick from the lower layer by dragging it in one of four possible directions: decreasing  $x$  (left), increasing  $x$  (right), decreasing  $y$  (front), increasing  $y$  (back). You can only drag a brick *along one direction*. E.g. you cannot drag it to the left, and then to the front to remove it. The bricks are smooth enough and don't suffer from friction, so you can remove any brick you want as long as dragging it out does not hit any other brick (touching other bricks is allowed, though), and after the brick is removed, no full half-blocks of any brick in upper layer are over empty space.

Write a program to find the maximum number of bricks you can remove from the lower layer.

### Input

The input contains several test cases. The first line of input contains two integer  $m, n$  ( $1 \leq m, n \leq 10$ ), the number of bricks in the lower and upper layer. The next line describes the lower layer with  $m$  brick descriptions in format  $(x_1, y_1) - (x_2, y_2)$  where integers  $x_1, y_1, x_2, y_2$  do not exceed 1000 by their absolute values and they satisfy  $x_1 < x_2, y_1 < y_2$ . Each description does not contain any space inside; neighboring descriptions are separated by exactly one single space. The next line describes the upper layer in the same format. The initial object is always valid. The last test case is followed by a single zero, which should not be processed.

### Output

For each test case, print the case number and the number of bricks can be removed from the lower layer.

### Sample Input

```
1 1
(0,0)-(1,1)
(0,0)-(1,1)
4 1
(0,0)-(2,2) (2,0)-(4,2) (0,2)-(2,4) (2,2)-(4,4)
(1,1)-(3,3)
0
```

### Output for the Sample Input

```
Case 1: 0
Case 2: 2
```

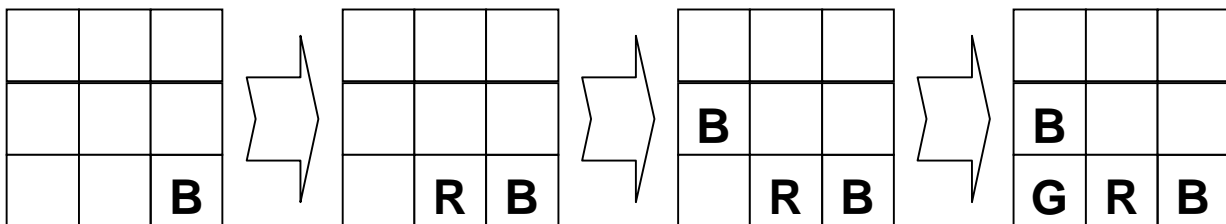


## Problem C

### Color Squares

Input: square.in

You have a 3\*3 board of color squares. Each square is either empty or has a block in it. Initially, all the squares are empty. There are four kinds of blocks: blue (B), red (R), green (G) and yellow (Y). Each of these block scores  $w_b$ ,  $w_r$ ,  $w_g$  and  $w_y$ , respectively (blocks of the same color always have the same score). We assume that  $w_b \leq w_r \leq w_g \leq w_y$ .



In each step, you can place a new block in a square. If that square already has a block in it, take it out first (taking it out does not count as a step). You can do this as many times as you like (you're given enough blocks for each color), as long as you follow these rules:

**Rule 1:** You can always place a blue block.

**Rule 2:** You can place a red block if and only if it's surrounded by at least one blue block.

**Rule 3:** You can place a green block if and only if it's surrounded by at least one blue and one red block.

**Rule 4:** You can place a yellow block if and only if it's surrounded by at least one blue, one red and one green block

Every square is surrounded by squares that share one edge with it, so each of four corner squares is surrounded by exactly two squares, each of four squares on the edge (but not at corners) is surrounded by exactly three squares, and the center square is surrounded by exactly four squares.

Write a program to find the minimal number of steps needed to get a score of at least  $w$ . The total score is the sum of individual scores of each block on the current board, regardless of what blocks you've thrown away.

### Input

The input contains several test cases. Each case contains five positive integer,  $w_b$ ,  $w_r$ ,  $w_g$ ,  $w_y$ ,  $w$  ( $1 \leq w_b \leq w_r \leq w_g \leq w_y \leq 100$ ,  $0 \leq w \leq 1000$ ) in a single line. The last test case is followed by a single zero, which should not be processed.

### Output

For each test case, print the case number and the minimum number of steps. If it is impossible, output "Impossible".

### Sample Input

```
1 1 1 1 3
1 2 4 8 21
1 1 1 100 500
7 20 53 94 395
0
```

### Output for the Sample Input

```
Case 1: 3
Case 2: 7
Case 3: Impossible
Case 4: 11
```



## Problem D

### Difficult Melody

Input: melody.in

You're addicted to a little game called '*remember the melody*': you hear some notes, and then you repeat it. In most cases, the longer the melody, the harder to repeat, but it isn't always true. Also, melodies of the same length are usually not equally easy to remember. To find a way to define the *remember difficulty* of a melody, you invented a statistics-based model:



Suppose you're investigating melodies of a particular length. If a melody appeared in  $p$  games, among which you successfully repeated  $q$  games, the smaller  $q/p$ , the more difficult the melody. If there is more than one melody having the minimal ratio, the one with larger  $p$  is considered more difficult. But there is an exception: if  $p$  is smaller than a threshold  $m$ , you simply ignore it (you can't call it difficult if you haven't tried it a lot of times, can you?). The melody appears in a game if its string representation is a consecutive substring occurring at least once in that game.

Write a program to find the most difficult melody of length  $k$ , given  $n$  games you've played.

### Input

The input contains several test cases. Each case consists of three integers  $n, m, k$  ( $1 \leq m \leq n \leq 100, 1 \leq k \leq 20$ ), the next  $n$  lines each contain two strings separated by exactly one space: the game, and whether you successfully repeated it. The first string will contain at least one at most 100 upper case letters 'C', 'D', 'E', 'F', 'G', 'A', 'B'. The second string will be either 'Yes' or 'No' (case sensitive). The last test case is followed by a single zero, which should not be processed.

### Output

For each test case, print the case number and the most difficult melody. If there is more than one solution, output the lexicographically smallest one. If there is no solution, output the string 'No solution'.

### Sample Input

### Output for the Sample Input

```
3 2 3
EEECEG Yes
BFCEG No
DEBFCEGEEC No
3 2 2
AAA No
BBB No
CCC Yes
0
```

```
Case 1: BFC
Case 2: No solution
```



## Problem E

### Expensive Drink

Input: drink.in

There are some water, milk and wine in your kitchen. Your naughty little sister made some strange drinks by mixing them together, and then adds some sugar! She wants to know whether they taste good, but she doesn't want to try them herself. She needs your help.

Your sister knows that you don't want to drink them either (anyone wants to?), so she gives you a chance to escape: if you can guess the price of a special drink, she gives you freedom. Warning: she loves her special drink so much that you should never under-estimate its cost! That is, you're to find the most expensive possible price of it.

The price of each drink equals to its cost. If the amounts of water, milk, wine and sugar used in the drink are  $a_1$ ,  $a_2$ ,  $a_3$  and  $a_4$  respectively, and the unit costs of water, milk, wine and sugar are  $c_1$ ,  $c_2$ ,  $c_3$  and  $c_4$  respectively, then the drink costs  $a_1c_1+a_2c_2+a_3c_3+a_4c_4$ . To give you some hope to win, she told you the costs of exactly  $n$  ordinary drinks. Furthermore, she promised that the total cost of sugar  $a_4c_4$  is always a *real number* in the interval  $[L, R]$ , in any drink.

Sadly, you don't know the exact price of anything (you're a programmer, not a housewife!), but you know that water is the cheapest; wine is the most expensive, i.e.,  $0 \leq c_1 \leq c_2 \leq c_3$ . Then the best thing you can do is to assume *units costs can be any real numbers satisfying this inequality*.

Write a program to find the highest possible price of the special drink.

#### Input

The input contains several test cases. The first line of each test case contains three positive integers  $n$ ,  $L$ ,  $R$  ( $1 \leq n \leq 100$ ,  $0 \leq L \leq R \leq 100$ ). The next  $n$  lines each contain four non-negative integer  $a_1$ ,  $a_2$ ,  $a_3$ ,  $p$  ( $0 \leq a_1, a_2, a_3 \leq 100$ ,  $0 \leq p \leq 10000$ ), the amount of water, milk and wine, and the price. The last line of the case contains three integers  $a_1$ ,  $a_2$ ,  $a_3$  ( $0 \leq a_1, a_2, a_3 \leq 100$ ), the drink to be estimated. The last test case is followed by a single zero, which should not be processed.

#### Output

For each test case, print the case number and the highest possible price to four decimal places. If the input is self-contradictory, output "Inconsistent data". If the price can be arbitrarily large, output "Too expensive!".

#### Sample Input

#### Output for the Sample Input

```
1 3 5
1 2 3 10
2 4 6
1 2 4
1 1 1 1
1 1 1
1 3 8
0 1 0 17
0 0 1
3 1 2
2 1 3 14
1 5 1 15
7 3 2 21
4 1 6
2 0 2
45 31 53 4087
30 16 1 1251
11 51 34
0
```

```
Case 1: 19.0000
Case 2: Inconsistent data
Case 3: Too expensive!
Case 4: 26.2338
Case 5: 3440.3088
```

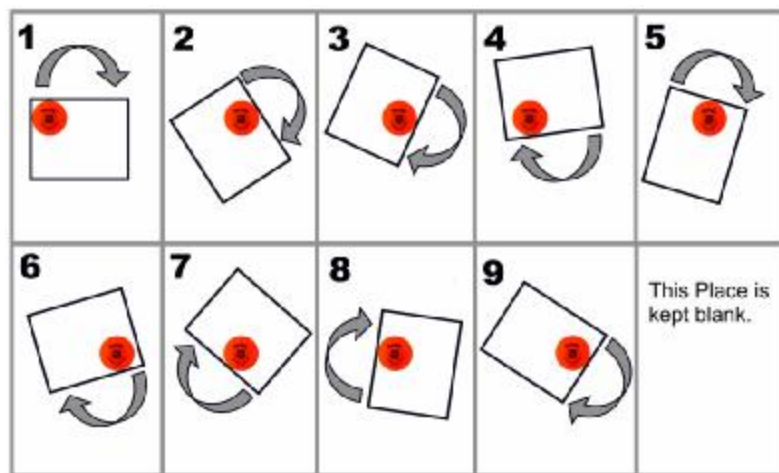


## Problem F

### Rotating a Frame

Input: rotate.in

Suppose you have an ice hockey ball (Which has the shape of a cylinder as shown in the figure) and a photo frame (a rectangle). You place the ball in one corner of the photo frame and keep the balls position and orientation fixed. Now you start rotating the frame in clockwise direction. But as the sides of the frame and of the ball have high friction so while the frame is rotating its surface and the balls surface never slips. So the frame always has a constant angular velocity as well as a tangential velocity as shown in the figure below.



Write a program to find the position of the frame after certain time.

### Input

The input contains several test cases. Each set of input is contained in two lines. The first line of a set contains eight integers which denote the values  $x_1, y_1, x_2, y_2, x_3, y_3$  and  $x_4, y_4$  respectively. These values indicate that the four vertices of the frame in clockwise order are denoted by the Cartesian coordinates  $P_1(x_1, y_1), P_2(x_2, y_2), P_3(x_3, y_3)$  and  $P_4(x_4, y_4)$ . You can assume that  $0 \leq |x_i|, |y_i|, |x_2|, |y_2|, |x_3|, |y_3|, |x_4|, |y_4| \leq 1000$ . The second line contains three floating point numbers which denotes the values of  $R$  ( $0 < R < 1000$ ),  $T$  ( $0 \leq T < 100000$ ) and  $\omega$  ( $0 \leq \omega < 360$ ) respectively. Here  $R$  is the radius of the ice hockey ball,  $T$  denotes that we want to know the position of the frame after  $T$  seconds and  $\omega$  is the angular velocity of the frame in degree/second. You can assume that ball is placed touching two the sides that intersect at point  $(x_1, y_1)$  and it never moves from or rotates in that position. There will be no such input where the hockey ball cannot be placed within the frame. The last test case is followed by a single zero, which should not be processed.

### Output

For each test case, print the case number and eight floating point numbers  $x_{1f}, y_{1f}, x_{2f}, y_{2f}, x_{3f}, y_{3f}, x_{4f}$  and  $y_{4f}$ , to three decimal places. These floating-point numbers actually denote the final positions of the points  $(x_1, y_1), (x_2, y_2), (x_3, y_3)$  and  $(x_4, y_4)$  (The four corners of the frame) respectively after time  $T$  seconds.

### Sample Input

### Output for the Sample Input

0 1 1 1 1 0 0 0 0.2 10 0	Case 1: 0.000 1.000 1.000 1.000 1.000 0.000 0.000 0.000
0 1 1 1 1 0 0 0 0.2 10 100 0	Case 2: -0.619 0.132 -0.445 1.117 0.539 0.943 0.366 -0.042

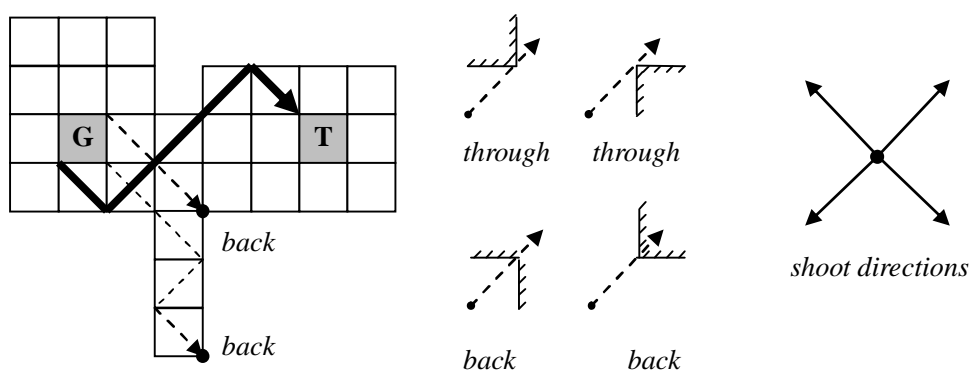


# Problem G

## Shoot Your Gun!

Input: gun.in

There are two rectangular polygons (simple polygons with interior angles of only 90 or 270 degrees)  $G$  and  $T$ , inside another rectangular polygon  $M$ . You can place a gun *anywhere* on the boundary of  $G$ , then shoot a bullet in one of four diagonal directions, and then touch the boundary of  $T$ . You may shoot across an edge of  $T$ , but touching only a corner is also allowed. Your bullet is not allowed to touch  $G$  again (even touching a corner of  $G$  is *not* allowed), before touching  $T$ .



The edges of  $M$  can reflect the bullet. When the bullet touches a vertex of  $M$ , it may simply go through it (and not regarded as a reflection), or go back. These special cases are shown in the figure above.

Write a program to find the minimal number of reflections needed from  $G$  to  $T$ .

### Input

The input contains several test cases. The first line of each case contains three positive integers  $n_G, n_T, n_M$  ( $4 \leq n_G, n_T, n_M \leq 50$ ). The next line contains  $n_G$  pairs of integers, the coordinates (non-negative integers not greater than 4000) of the vertices of  $G$ , in counter-clockwise order. The next two lines describe polygon  $T$  and  $M$ , in the same format. It is guaranteed that  $G$  and  $T$  are outside each other (their boundaries will not touch), and are both inside  $M$  (they do not touch the boundary of  $M$ ). The last test case is followed by a single zero, which should not be processed.

### Output

For each test case, print the case number and the minimal number of reflections to touch  $T$ . If it's impossible, output -1.

### Sample Input

```
4 4 12
1 4 2 4 2 5 1 5
6 4 7 4 7 5 6 5
0 3 3 3 3 0 4 0 4 3 8 3 8 6 4 6 4 5 3 5 3 7 0 7
4 4 4
1 1 2 1 2 2 1 2
3 1 4 1 4 2 3 2
0 0 5 0 5 3 0 3
4 4 4
1 1 2 1 2 2 1 2
6 1 7 1 7 2 6 2
0 0 8 0 8 3 0 3
0
```

### Output for the Sample Input

```
Case 1: 2
Case 2: 0
Case 3: 1
```

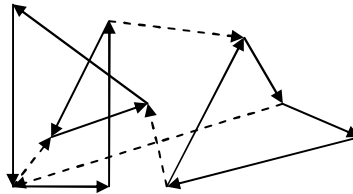


# Problem H

## Help Little Laura

Input: painting.in

Laura Luo has just invented a game. Given a beautiful pencil sketch with  $n$  points, you're to colorize it with water pens by painting circuits. Each time you paint a new circuit, starts with one point, follow some line segments and return to the starting point. Every point can be reached more than once, but every segment can be painted at most once. To make the picture look interesting, different segments must be painted different colors. For each segment, Laura has already decided a direction to paint it. The picture below illustrates a possible way to paint the picture (dashed lines are segments that are not painted).



After you finish painting, your score is computed as follows: for each unit length you paint, you earn  $x$  points, for each color you use, you lost  $y$  points (Laura has prepared enough water pens of different colors).

Write a program to find the maximal score you can get.

### Input

The input contains several test cases. The first line of each case contains three positive integers  $n, x, y$  ( $1 \leq n \leq 100, 1 \leq x, y \leq 1000$ ). The next  $n$  lines each describe a point (points are numbered from 1 to  $n$  in the order they appear in the input). The first two integers  $(x, y)$  specify its coordinates ( $0 \leq x, y \leq 1000$ ). The rest integers are the points it connects to, ended by a zero. If point  $v$  appears in the list of point  $u$ , there is a line segment connecting  $u$  and  $v$  (then there will not a segment connecting  $u$  and  $v$  in the reverse direction). Furthermore, Laura will paint it from  $u$  to  $v$ . There will be no duplicated points and no more than 500 segments. The last test case is followed by a single zero, which should not be processed.

### Output

For each test case, print the case number and the maximal score you can get, to two decimal places.

### Sample Input

```
4 5 1
0 0 2 3 0
1 0 3 4 0
1 1 4 0
0 1 1 0
1 2 1
0 0 0
10 7 2
0 0 2 4 0
5 0 3 0
5 10 4 10 0
2 3 5 0
7 5 6 0
0 11 1 0
8 0 10 5 0
18 3 7 0
14 5 8 1 0
12 9 9 0
0
```

### Output for the Sample Input

```
Case 1: 16.00
Case 2: 0.00
Case 3: 522.18
```



# Problem I

## Integer Transmission

Input: integer.in

You're transmitting an  $n$ -bits unsigned integer  $k$  through a simulated network. The  $i$ -th bit counting from left is transmitted at time  $i$  (e.g. 4-bit unsigned integer 5 is transmitted in this order: 0-1-0-1). The network delay is modeled as follows: if a bit is transmitted at time  $i$ , it may arrive at as early as  $i+1$  and as late is  $i+d+1$ , where  $d$  represents the maximal network delay. If more than one bit arrived at the same time, they could be received in any order.

For example, if you're transmitting a 3-bit unsigned integer 2 (010) for  $d=1$ , you may receive 010, 100 (first bit is delayed) or 001 (second bit is delayed).

Write a program to find the number of different integers that could be received, and the smallest/largest ones among them.

### Input

The input contains several test cases. Each case consists of three integers  $n, d, k$  ( $1 \leq n \leq 64, 0 \leq d \leq n, 0 \leq k < 2^n$ ), the number of bits transmitted, the maximal network delay, and the integer transmitted. The last test case is followed by a single zero, which should not be processed.

### Output

For each test case, print the case number and the number of different integers that could be received, followed by the minimal and maximal one among them.

### Sample Input

### Output for the Sample Input

3 0 2	Case 1: 1 2 2
3 1 2	Case 2: 3 1 4
10 2 888	Case 3: 25 490 984
7 3 107	Case 4: 19 47 122
0	



## Problem J

### Jewel Trading

Input: trade.in

You want to sell a diamond to a merchant for a good price. You know so much about how merchant likes the diamond that you have even built a mathematical model for it: He will definitely accept the price  $p$  if it's not greater than a certain threshold  $a$ , but for a price  $p$  higher than it, he must have a think. The higher the price, the lower probability he will accept. Precisely, the probability that he accept price  $p > a$  is  $1 / (1 + (p - a)^b)$ , where  $b > 1$  is a positive constant in your model.



The exact trading process is as follows: you first propose a price (a non-negative *integer*), then the merchant decides whether to accept. If he accepts, the trade is over and you have no chance to regret. If he does not accept, you propose another price, and so on. You know that the merchant would get angry if you always propose unacceptable high prices, so you promised that the  $n$ -th proposal (if there is) is always not greater than  $a$  (which he can accept for sure).

Write a program to find an optimal way to propose prices to maximize your expected earning (i.e. the final price).

### Input

The input consists of several test cases. Each case is described by two positive integers  $n$ ,  $a$ , and a real number  $b$  ( $1 \leq n \leq 100$ ,  $1 \leq a \leq 1000$ ,  $1 < b < 10$ ),  $b$  is given to up to three decimal places. The last test case is followed by a single zero, which should not be processed.

### Output

For each test case, print the case number and the expected earning, to two decimal places. It is guaranteed that the maximal earning exists.

### Sample Input

```
1 10 2
10 33 3.14
0
```

### Output for the Sample Input

```
Case 1: 10.00
Case 2: 34.41
```