Problem A
Lego Land
Input: A.txt

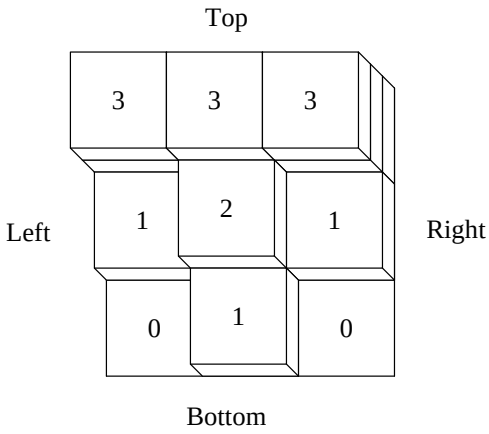

Figure 1: VR view of a Lego downtown

As an urban Lego engineer, it is vital to obtain every possible technological advantage over your competition. You have recently noticed how popular virtual reality technology is with human engineers and wish to have something similar to help woo venture capitalists. Luckily, a VR view of a Lego downtown is very simple. First, you only have to show buildings, and you already know the heights and locations of all the buildings you want to build. Secondly, Lego VR is only capable of presenting a view of cube-shaped section of the city. It does this by projecting the cube onto one of the planes containing one of the cubefs faces. Lego engineers may browse the view by rotating the cube-shaped city section 90 degrees at a time about any one of the X, Y, or Z-axes passing through its center.

You are to write a Lego VR program that will take as input an $N \times N$ matrix view of a downtown area and a series of rotational commands and produce the resulting $N \times N$ matrix view of the same downtown area from the resulting perspective. The $N \times N$ input matrix will represent an overhead view of an $N \times N \times N$ cube of a Lego downtown (with one face of the cube at ground level).

Each number in the input matrix indicates the height of the building in the corresponding location. For example, 0 indicates that no building is present, while 3 indicates a building 3 stories (Legos) high. Note that, in the initial overhead view, there is no hidden open space. In other words, the buildings are solid. Also note that only buildings are represented in the view, not the ground. Any region of the view that does not contain a block of a building will have value 0.

Input

Input consists of an unspecified number (at least one) of sets of three sections:

1. Grid size — A single line containing a single integer N ($1 \leq N \leq 9$) which represents the size of the cube of buildings represented.
2. Building heights — An $N \times N$ array of whole numbers in the range 0 to N (inclusive). Each row in the array is given on its own input line, and array elements are not delimited. This initial $N \times N$ array represents an overhead view of an $N \times N$ solid structure of Lego buildings, with each number representing the height of the building in the corresponding location. Height of 0 indicates that no Lego exists in the given location.
3. A series of rotational commands about the center of the $N \times N \times N$ cube containing the Legos, one of:
   - "`X`" — rotates about the horizontal axis in the current view. The cube rotates such that the face of the cube that is at the *top* of the current view moves forward to become the new viewing face.
   - "`Y`" — rotates about the vertical axis in the current view. The cube rotates such that the face of the cube that is at the *left* of the current view moves forward to become the new viewing face.
   - "`Z`" — rotates about the line-of-sight axis in the current view. The viewing face does not change but is rotated 90 degrees counterclockwise.

   These commands are a single-unbroken string with at most 80 commands.

Between each two sets of input data is exactly one blank line. There will be no blank line before the first input set or following the last input set.

No invalid input will be provided.

Output

For each input case, output consists of an $N \times N$ array of numbers in the range 0 to N (inclusive) representing the final view for each input array and its associated set of rotations. Output arrays must be separated by exactly one blank line, and there must be no white space following the last digit of the last output array. Similarly, the first digit of the first output array must be the first character in the output.

No extraneous output of any kind should be printed.

Since this program essentially generates views of 3-dimensional objects, it is good to be very specific in describing how these views are calculated. Looking at the figure above, note that the cube has six sides, top, left, bottom, and right are labeled. The front is the side of the cube nearest the viewer, and the back is the side farthest from the viewer. The initial input view is

meant to represent a Lego city from above. Each number represents the heights of the building at that location (or correspondingly, the maximum distance from the back of the cube for any Lego piece in that row and column). When the cube is rotated, the numbers no longer indicate the height of buildings since the view is no longer (necessarily) a view from above. Instead, the numbers continue to represent the maximum distance from the back of the cube for any Lego piece in that row and column of the cube. The idea is that the numbers will represent those Legos that are nearest the viewer, and therefore those Legos that are visible to the viewer.

Sample Input

```
3
010
021
010
Z

5
11111
22222
33333
44444
55555
XXXY

9
000010000
000020000
000030000
000040000
135797531
000040000
000030000
000020000
000010000
ZXXXXZYYYYZXXXXZYYYYY
```

Output for the Sample Input

```
010
121
000

00005
00055
00555
```

05555
55555

500000000
550000000
555000000
555500000
988776655
555500000
555000000
550000000
500000000

Problem B
Wall
Input: B.txt

Once upon a time there was a greedy King who ordered his chief Architect to build a wall around the King's castle. The King was so greedy, that he would not listen to his Architect's proposals to build a beautiful brick wall with a perfect shape and nice tall towers. Instead, he ordered to build the wall around the whole castle using the least amount of stone and labor, but demanded that the wall should not come closer to the castle than a certain distance. If the King finds that the Architect has used more resources to build the wall than it was absolutely necessary to satisfy those requirements, then the Architect will loose his head. Moreover, he demanded Architect to introduce at once a plan of the wall listing the exact amount of resources that are needed to build the wall.

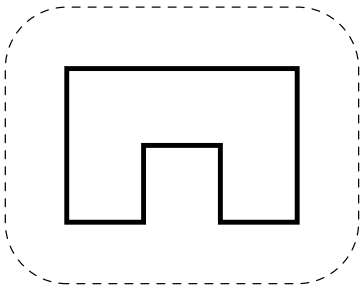

Figure 2: Wall around the castle

Your task is to help poor Architect to save his head, by writing a program that will find the minimum possible length of the wall that he could build around the castle to satisfy King's requirements.

The task is somewhat simplified by the fact, that the King's castle has a polygonal shape and is situated on a flat ground. The Architect has already established a Cartesian coordinate system and has precisely measured the coordinates of all castle's vertices in feet.

Input

The input file consists of multiple data sets.

The first line of each data set contains two integer numbers N and L separated by a space. N ($3 \leq N \leq 1000$) is the number of vertices in the King's castle, and L ($1 \leq L \leq 1000$) is the minimal number of feet that King allows for the wall to come close to the castle.

Next N lines describe coordinates of castle's vertices in a clockwise order. The i-th line contains two integer numbers X_i and Y_i separated by a space ($-10000 \leq X_i, Y_i \leq 10000$) that represent the coordinates of the i-th vertex. All vertices are different and the sides of the castle do not intersect anywhere except for vertices.

The end of the input file is indicated by $N = L = 0$.

Output

For each case, write a line that contains the single number that represents the minimal possible length of the wall in feet that could be built around the castle to satisfy King's requirements. You must present the integer number of feet to the King, because the floating numbers are not invented yet. However, you must round the result in such a way, that it is accurate to 8 inches (1 foot is equal to 12 inches), since the King will not tolerate larger error in the estimates.

Sample Input

```
9 100
200 400
300 400
300 300
400 300
400 400
500 400
500 200
350 200
200 200
0 0
```

Output for the Sample Input

```
1628
```

Problem C
Trees
Input: C.txt

A Binary Search Tree (BST) is a binary tree where every node has a value and the tree is arranged so that, for any node all the values in its left subtree are less than the node's value, and all the values in its right subtree are greater than the node's value.

To build a BST from a sequence of distinct integers the following procedure is used. The first integer becomes the root of the tree. Then the second integer in the sequence is considered. If it is less than the value of the root node then it becomes the left child. Otherwise, it becomes the right child. Subsequent items in the sequence move down either left or right in the same fashion depending on the comparison against earlier nodes, starting at the root and progressing down the tree. The new node is inserted (as a leaf) into the partially constructed tree when it reaches a missing subtree in the particular direction of travel.

For example, a BST generated by the sequence $[2, 1, 4, 3]$ is built up as shown below as the numbers are inserted.

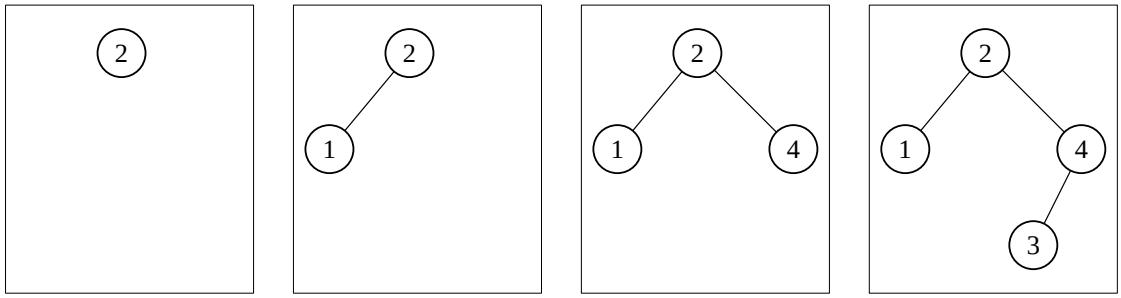

Figure 3: Generation of BST

The tree (d) of Figure 3 can also be generated by two other sequences: $[2, 4, 3, 1]$ and $[2, 4, 1, 3]$.

Such sequences can be compared according to a lexicographic order, where the comparison proceeds left-to-right and items at corresponding positions are compared using their numerical values. The result for $[2, 1, 4, 3]$, $[2, 4, 3, 1]$ and $[2, 4, 1, 3]$ is as follows, using "$<$" to denote this lexicographic order between sequences: $[2, 1, 4, 3] < [2, 4, 1, 3] < [2, 4, 1, 3]$.

For the tree (d) in Figure 3 the lexicographically least sequence is $[2, 1, 4, 3]$.

Write a program that will read a representation of a tree and will generate the lexicographically least sequence of distinct positive integers that will generate that tree. Note that for a tree with n nodes this sequence will be a permutation of the numbers from 1 to n.

For the input to our problem we represent (recursively) the structure of a tree as follows:

1. A single node (a leaf) is a tree:
   - `()`
2. If T_L and T_R are trees then the following are also trees:
   - `(`T_L`,)`
   - `(,`T_R`)`
   - `(`T_L`,`T_R`)`

Input

Input will consist of a sequence of lines. Each line will consist of up to 250 characters and will contain the specification of a single tree according to the above definition (with no intervening spaces). The sequence will be terminated by a tree consisting of a single node `()`. This line should not be processed.

Output

Output will be a sequence of lines, one for each line in the input. Each line will consist of the required permutation of the integers from 1 to n (where n is the number of nodes in the tree) separated by single spaces.

Sample Input

```
((),(()))
(((),),())
(((),()),((),()))
(,())
((),)
()
```

Output for the Sample Input

```
2 1 4 3
3 2 1 4
4 2 1 3 6 5 7
1 2
2 1
```

Problem D
Colors
Input: D.txt

A manager for a toy company wants to reduce the cost of manufacturing their line of toys. Briefly, the toys are created by robots that operate on assembly tracks by adding and linking track components into modules and by merging existing modules into more complex modules. Components can be either *active* or *inactive*. At any moment there is exactly one *active component* per each *module* and *track*; and this is the only component that can be linked or merged on that track for that module.

The new budget for this company will only support components which consist of *three colors* and *adjacent components* must have *different colors*. Your job is to decide which of the current toys in the inventory can be produced with this coloring limitation.

The company uses the following BNF formalism to describe more precisely the blueprints of its toys:

1. ⟨*toy*⟩ ::= ⟨*last-track*⟩⟨*module*⟩

   The current ⟨*toy*⟩ consists of a main ⟨*module*⟩. ⟨*last-track*⟩ gives the number of the last track used to build the current ⟨*toy*⟩; the tracks are numbered from 0 to ⟨*last-track*⟩.

2. ⟨*module*⟩ ::= '`(`' ⟨*operator-sequence*⟩ '`)`'

   ⟨*module*⟩ represents a simple module that only contains operators. The operators given by the ⟨*operator-sequence*⟩ are processed in a left-to-right order.

   This ⟨*module*⟩ starts with empty tracks and then automatically adds one active component on each of the available tracks.

3. ⟨*merged-module*⟩ ::= '`(`' ⟨*module*⟩$_1$⟨*module*⟩$_2$ '`)`'

   This is a merge operation that builds a complex ⟨*merged-module*⟩ by merging the active components of ⟨*module*⟩$_1$ and ⟨*module*⟩$_2$, after both modules are completely built.

4. ⟨*module*⟩ ::= '`(`' ⟨*merged-module*⟩⟨*operator-sequence*⟩ '`)`'

   The components of the ⟨*merged-module*⟩ remain on the tracks and its active components are further worked upon by the operators given by ⟨*operator-sequence*⟩.

5. ⟨*operator-sequence*⟩ ::= λ | ⟨*operator*⟩⟨*operator-sequence*⟩

6. ⟨*operator*⟩ ::= ⟨*node-operator*⟩ | ⟨*edge-operator*⟩

7. ⟨*node-operator*⟩ ::= ⟨*track-number*⟩

   A ⟨*node-operator*⟩ adds an active component on the specified ⟨*track-number*⟩ for the current module and the previously active component on that track becomes inactive.

8. ⟨*edge-operator*⟩ ::= ⟨*track-number-pair*⟩

   An ⟨*edge-operator*⟩ links the active components of the two track-numbers.

The following examples show several simple toy blueprints; in the figures tracks are represented as horizontal dotted lines, components as circles, links as full lines, and the time axis flows from left to right.

Example 1 Figure 4 depicts a 3-colorable toy that can be built using the following blueprint:

```
2 ( 20 10 21 2 20 )
```

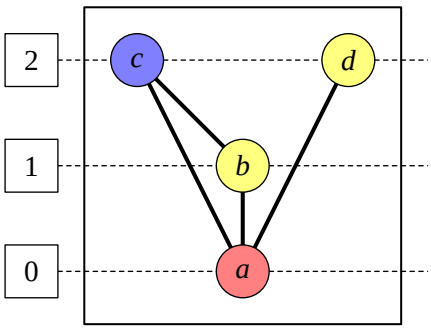

Figure 4: 3-colorable toy

There are 3 tracks numbered 0, 1, 2. The toy consists of a single module containing 4 components labeled a, b, c, d, linked by the lines c–a, b–a, c–b, d–a. To build this toy the robot will execute in order the following operations:

1. Add a on track 0, b on track 1, c on track 2. At this stage a, b and c are the active components.
2. Make the links c–a, b–a and c–b.
3. Add d on track 2, which makes d active and inactivates c.
4. Make the link d–a. At this stage a, b and d are the active components.

Note that the same toy can also be built using several other blueprints, such as the following two:

```
2 ( 10 20 21 2 20 )
2 ( ( ( 20 10 ) ( 21 ) ) 2 20 )
```

Example 2 Figures 5 and 6 illustrate a sequence of operations involving a merge, for another 3-colorable (in fact even 2-colorable) toy that can be built as specified by the following blueprint:

```
1 ( ( ( 10 1 10 0 ) ( 10 1 10 0 ) ) 10 1 10 )
```

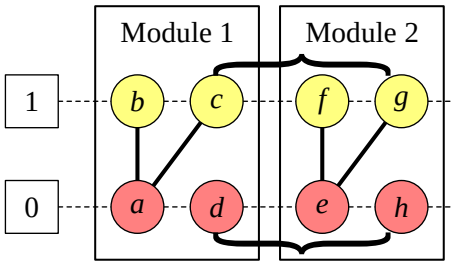

Figure 5: Modules to be merged

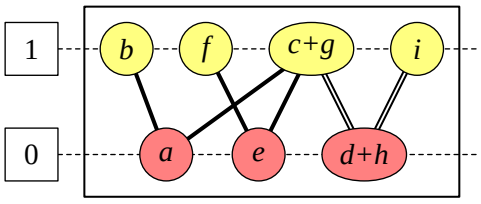

Figure 6: Resulting toy

1. First, module 1 is built:
   (a) Add a on track 0 and b on track 1.
   (b) Link b–a.
   (c) Add c on track 1.
   (d) Link c–a.
   (e) Add d on track 0. c and d are now the active components of module 1.
2. Secondly, module 2 is built using similar operations. g and h are now the active components of module 2.
3. Thirdly, modules 1 and 2 are *merged* together, which means that *active* components of each track are identified, i.e., $c = g$ and $d = h$. The snapshot of Figure 5 illustrates this moment, with braces showing component identification. $(c + g)$ and $(d + h)$ are now the active components of the merged module $1 + 2$.
4. Fourthly, the just merged components are linked, i.e., $(c + g)$–$(d + h)$.
5. Lastly, i is added to track 1 and linked with $(d+h)$. The final result is depicted in Figure 6 and links made after the merging are depicted with double lines. At the end, i and $(d+h)$ are the active components of the main module.

Input

The input will consist of a sequence of toy blueprints, one per line of at most 250 characters. Each toy blueprint contains a sequence of tokens separated by single spaces, and conforming to the BNF rules stated earlier.

The first token is a positive integer t, $0 \leq t \leq 6$, denoting the maximum track number for the robot's arms to grab (i.e., there are $t + 1$ current components for the robot).

The interpretations for the remaining tokens are given in the next table.

Token	Meaning of the Robot's Instruction
i	Add a new component on track i, where i is a decimal digit, $0 \leq i \leq t$. Note that this component becomes the active component on this track.
ij	Link the active components on tracks i and j, where i and j are decimal digits, $t \geq i > j \geq 0$. Note this token consists of two track numbers, with no intervening space.
`(`	Begin marker for a new module. Note that, according to the BNF description two tokens '`)`' and '`(`' adjacent in sequence denote a merge operation.
`)`	End marker for a new module.

The input will be terminated by a toy description with $t = 0$, which is not processed.

Output

The required output is a line of the form "`Toy #: ?`", where `#` denotes the toy sequence number starting at 1 and `?` is either `Yes` or `No` depending whether the toy can be properly built using at most 3 colors.

Sample Input

```
2 ( 20 10 21 2 20 )
1 ( ( ( 10 1 10 0 ) ( 10 1 10 0 ) ) 10 1 10 )
3 ( 32 31 20 21 10 0 10 30 20 )
2 ( ( ( 10 1 10 21 1 21 10 ) ( 21 ) ) 0 10 20 )
0 ( )
```

Output for the Sample Input

```
Toy 1: Yes
Toy 2: Yes
Toy 3: No
Toy 4: Yes
```

Problem E
A Treasure Or A Bomb
Input: E.txt

An adventurer seeking a legendary treasure discovered a mysterious door in a deep cavern. The door had many keyholes, and near the door the same number of keys were placed. Keyholes were numbered from 1 to N, and so were the keys.

According to a treasure map he had, the door would open and lead him to the treasure room if he inserted all the keys into the keyholes at the same time. Each key might be inserted into an arbitrary keyhole, so it seemed to be an easy task — in fact it was not true because there was one big trap on the door. When each of the keys was inserted to a keyhole, a bomb planted in the door might explode in some probability.

On the kindly treasure map, all p_{ij}'s (the probability of explosion when the i-th keyhole was plugged by the j-th key) were listed. The cautious but greedy adventurer had decided to insert the keys in the way to maximize the safety, that is, to maximize the probability *not* to explode. Suppose the situation with two keys and keyholes where $p_{11} = 0.4$, $p_{12} = 0.5$, $p_{21} = 0.5$, $p_{22} = 0.6$, if he inserts the first key to the first keyhole and the second to the second, the probability of non-explosion is $(1 - 0.4) \times (1 - 0.6) = 0.24$. On the other hand, if he inserts the second key to first hole and the first key to the second, then the probability is 0.25, which is better.

You are to find the best way to insert the keys. You may assume the answer is unique. The maximum probability of no-exlopsion is strictly 1.00001 times larger than the probability with any non-optimal key insertion.

Input

The input for this problem consists of multiple test cases. Each case begins with a line containing single integer N ($1 \leq N \leq 100$), which specifies the number of the keys and keyholes. In the following N lines, the i-th line contains N real numbers $p_{i1}, \ldots, p_{iN}$ ($0.00001 \leq p_{ij} \leq 0.99999$). p_{ij} gives the probability of explosion when the i-th keyhole is plugged by the j-th key. Real numbers are represented by decimal form with at most five digits after the decimal point.

The input is terminated by a line which contains a single zero.

Output

For each test case, output N lines. The i-th line for each test case should contain only one integer to represent the key which the adventurer should insert into the i-th keyhole. Outputs

for different test cases must be separated by one blank line.

Sample Input

```
3
0.8 0.9 0.1
0.1 0.4 0.5
0.6 0.1 0.7
2
0.4 0.5
0.5 0.6
0
```

Output for the Sample Input

```
3
1
2

2
1
```

Problem F
Up and Down
Input: F.txt

A permutation of the set of numbers $\{1, \ldots, N\}$ is a reordering of these numbers, where each number appears just once. We define the *up-sequence* and the *down-sequence* of a permutation as follows.

Up-sequence $[u_1, \ldots, u_N]$**:** u_i is a minimum positive number k where $a_i < a_{i+k}$ holds. If there is not such a number, then $u_i = N + 1 - i$.

Down-sequence $[d_1, \ldots, d_N]$**:** d_i is a minimum positive number k where $a_i > a_{i+k}$ holds. If there is not such a number, then $d_i = N + 1 - i$.

For example, given the permutation $[1, 4, 3, 2, 6, 5]$, the up-sequence is $[1, 3, 2, 1, 2, 1]$ and the down-sequence is $[6, 1, 1, 3, 1, 1]$.

In general there is more than one permutation that shares up-sequence and down-sequence. For the example above, two other permutations $[1, 5, 4, 2, 6, 3]$ and $[1, 5, 3, 2, 6, 4]$ share the same up-sequence and down-sequence.

You are requested to count the number of permutations in regard to given up-sequence and down-sequence.

Input

The input contains a series of data sets.

Each set consists of three lines. The first line contains one positive integer N ($N \leq 17$). The following two lines contain N integers each, where the former line denotes an up-sequence and the latter a down-sequence. These integers are separated by a space.

The input is terminated by a line containing a single zero. This line should not be processed.

Output

For each set, print a line containing the number of possible permutations in regard to given up-sequence and down-sequence.

Sample Input

```
6
1 3 2 1 2 1
6 1 1 3 1 1
0
```

Output for the Sample Input

```
3
```

Problem G
Help the Museum
Input: G.txt

The National Association of Museum Curators came to you with an interesting problem. The President of the country, in order to improve his public image, has decided to visit the various Art Museums in the country, to convey the impression that he is a refined man. Being a very busy person, however, and knowing nothing about art, he imposed two restrictions for the visits:

1. In each museum, he wants to see the works of one and only one artist, so that he can easily prepare himself for the visit and pose as an art connoisseur. However he does not necessarily have to see all of the works of that artist.
2. He does not want to waste time, and demands to walk through the exposition following the shortest possible path.

The curators are willing to follow the President's demands, but they do not want to redistribute the masterpieces in the exposition only to obtain a straight path. Their only concession is to exchange temporarily the place of two masterpieces, if this helps to obtain a shorter path.

You should write a program that receives as input the layout of an exposition and finds the shortest path, according to the above constraints. To make your task easier, the curators have already defined a standard layout. Figure 7 shows one such layout.

10	B	B	B	B	B	B	F	F	F	F	F
9	A	A	A	A	A	B	D	C	C	F	F
8	A	F	F	F	A	B	A	A	C	F	C
7	B	F	E	F	A	B	B	B	B	B	D
6	F	F	D	E	A	B	A	A	A	B	A
5	E	E	D	E	E	E	E	E	A	B	B
4	D	D	D	E	E	E	E	E	A	A	B
3	D	C	C	F	F	F	C	C	A	B	A
2	D	C	C	F	F	F	C	C	A	A	A
1	C	C	C	C	C	C	C	C	C	C	C
Y/X	1	2	3	4	5	6	7	8	9	10	11

Figure 7: Layout of the Museum

The President's walk begins always at the left wall ($X = 1$, any Y) and ends at the right wall ($X = X_{\max}$, any Y). The walk can be done horizontally or vertically; diagonal movements are

not allowed. The works of a given artist are all labeled with the same capital letter (A, B, C, etc). From Figure 1, several cases can be illustrated:

1. If the president wants to see the works of artist A, there is not a path from left to right. Such a path can be obtained if the work by artist B at $(6, 6)$ is exchanged with one by artist A from $(1, 8)$, $(7, 8)$, $(8, 8)$, $(10, 6)$, $(11, 6)$ or $(11, 3)$.
2. If the president wants to see the works of artist B, there is already a path, beginning at $(1, 10)$ and ending at $(11, 5)$. A shorter path can be obtained, if the work of D at $(11, 7)$ is exchanged with one work by artist B, for example from $(10, 6)$.
3. If the president wants to see the works of artist C, there is already a straight path from $(1, 1)$ to $(1, 11)$, and a shorter path cannot be obtained.
4. If the president wants to see the works of D, E or F, there is no possibility to obtain a path from left to right.

Input

The input file may contain several instances of the problem. Each instance has the following format (all numbers are positive integers):

1. The first line contains the integers $X_{\max}$ and $Y_{\max}$, the layout dimensions. You may assume that $1 \leq X_{\max}, Y_{\max} \leq 100$.
2. The second line contains the artist (upper-case) letter that will have his/her works visited.
3. $Y_{\max}$ lines, each with $X_{\max}$ letters (without spaces between them). The first input line corresponds to index $Y_{\max}$, the second line to index $Y_{\max} - 1$, and so on, until the last line, that corresponds to index 1.

A line containing two zeros terminates the input file. Numbers are separated by spaces.

Output

For each instance of the problem, your program should produce output as follows.

If a path exists, your program should first print one line with the message "`Exchange (`x`,`y`) and (`u`,`v`)`" if an exchange will occur, or "`No exchange`" otherwise. Following that, the shortest path should be printed, one coordinate per line. In case more than one path is the shortest, any one of them may be printed, except that a path without an exchange should be preferred to those with exchanges.

If a path does not exist, your program should print only one line with the message "`No path`".

The output of each instance is terminated with a blank line.

Sample Input

```
11 10
A
BBBBBBFFFFF
AAAAABDCCFF
AFFFABAACFC
BFEFABBBBBD
FFDEABAAABA
EEDEEEEEABB
DDDEEEEEAAB
DCCFFFCCABA
DCCFFFCCAAA
CCCCCCCCCCC
11 10
C
BBBBBBFFFFF
AAAAABDCCFF
AFFFABAACFC
BFEFABBBBBD
FFDEABAAABA
EEDEEEEEABB
DDDEEEEEAAB
DCCFFFCCABA
DCCFFFCCAAA
CCCCCCCCCCC
0 0
```

Output for the Sample Input

```
Exchange (6,6) and (1,8)
1 9
2 9
3 9
4 9
5 9
5 8
5 7
5 6
6 6
7 6
8 6
9 6
9 5
9 4
```

```
9 3
9 2
10 2
11 2

No exchange
1 1
2 1
3 1
4 1
5 1
6 1
7 1
8 1
9 1
10 1
11 1
```

Problem H
How Many Days Are There?
Input: H.txt

> "I'm sorry but I can't attend the party on the 10th of this month. You know, my holidays are only on Thursdays. I have been longing for the party, but I have to give up being present there."
>
> "It doesn't matter. I hope that you will be present the next time."

Have you ever encountered such situations? No matter how regretful you are about them, you still could do nothing. So, we desire to find a way to know how many days in a period meet our demand not only of the date but of the day of week.

In this problem, we consider a special calendar from ancient to future times. In the calendar, we stipulate as follows:

1. In the calendar, the year preceding 1 year A.D. is 1 year B.C. For the convention, we represent 1 year B.C. by 0, 2 year B.C. by -1, and so on.
2. There are seven days in a week: Monday, Tuesday, Wednesday, Thursday, Friday, Saturday and Sunday. They occur in this order repeatedly. In the calendar, Jan. 1st, 1 year A.D. is Monday. The day of week of another date is determined consistent with Jan. 1st, 1 year A.D.
3. A year divisible by 4 but not by 100 is a leap year. A year divisible by 400 is also a leap year. This rule applies to years of B.C. For example, 5 year B.C. is a leap year because that year is represented by -4.
4. A year consists of twelve months. In January, March, May, July, August, October and December, there are 31 days each. In April, June, September and November, there are 30 days each. And in February, there are 29 days if the year is a leap year; 28 days otherwise.
5. Your program should process dates from 2000000 year B.C. to 2000000 year A.D.

Input

The input file for this problem will contain a series of data sets. Each data set will given in a line, and begin with two integers. The first integer m ($0 \leq m \leq 6$) is the day of week in demand, where $m = 0$ represents Sunday, $m = 1$ represents Monday, $m = 2$ represents Tuesday, and so

on. The second integer n ($0 \leq n \leq 31$) is the day part of the date in demand. After that, there will be six integers, y_1, m_1, d_1, y_2, m_2, d_2. y_1, m_1 and d_1 are the year part, the month part and the day part of the start date, and y_2, m_2 and d_2 are the year part, the month part and the day part of the end date. You may assume that the start date and the end date will conform to the calendar. The end date will not be earlier than the start date.

The end of input is signified by a line which contains two zeros. This line should not be processed.

Output

The output for each data set should include a line containing a integer which tells how many days conform to our demands from the start date to the end date (including the start date and the end date). Here, we say the date conform to our demands if the day of week of a date equals to m and the day part of this date equals to n. You should not print any more whitespaces or blank lines in the output.

Sample Input

```
0 28 1999 11 18 1999 11 28
5 28 1999 11 18 1999 11 28
5 13 1999 1 1 1999 12 31
0 30 1976 6 30 1999 11 28
0 0
```

Output for the Sample Input

```
1
0
1
35
```

Problem I
Cut out
Input: I.txt

You are given a 3-dimensional convex polyhedron, and required to find its cut vertical to z-axis that maximizes the area of the resulting section.

Input

There are multiple test cases in the input. Each test case begins with a single integer n ($n \leq 20$), the number of faces of the polyhedron. The following n lines are the description of the faces. Each line is given in the format below.

$$m\ x_1\ y_1\ z_1\ x_2\ y_2\ z_2\ \ldots\ x_m\ y_m\ z_m$$

m ($3 \leq m < 20$) is the number of vertices of the polygonal face. x_i, y_i and z_i are the integral coordinates of the i-th vertex ($0 \leq x_i, y_i, z_i < 10000$). The distance between every pair of vertices is greater than 0.01.

The end of input is indicated by a line containing only a single zero.

It is guaranteed that a set of polygons given as a test case forms a convex polyhedron.

Output

Output the area of the largest cutting plane in one line for each test case. The area may have an arbitrary number of digits after the decimal point, but should not contain an error greater than 10^{-5}.

Sample Input

```
6
4  0 0 0  1 0 0  1 1 0  0 1 0
4  0 0 0  1 0 0  1 0 1  0 0 1
4  0 0 0  0 1 0  0 1 1  0 0 1
4  1 1 1  0 1 1  0 1 0  1 1 0
4  1 1 1  0 1 1  0 0 1  1 0 1
4  1 1 1  1 0 1  1 0 0  1 1 0
6
```

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

Output for the Sample Input

```
1.00
9.00
```