Problem A
Air Conditioning Machinery
Input file: ducts.in

You are a technician for the Air Conditioning Machinery company (ACM). Unfortunately, when you arrive at a customer site to install some air conditioning ducts, you discover that you are running low on supplies. You have only six duct segments, and they are all of the same kind, called an "elbow."

You must install a duct in a confined space: a rectangular prism whose sides are multiples of a unit length. Think of the confined space as consisting of an array of unit cubes. Each elbow occupies exactly four unit cubes, as shown in Figure 1 below. A unit cube can be occupied by at most one elbow. Each elbow has exactly two openings, as indicated by the gray squares in the elbow shown in Figure 1. You may assemble the elbows into longer ducts, but your duct must be completely contained inside the given space. One way to connect two elbows is shown in Figure 2. Your task is to connect an inflow to an outflow. The inflow and the outflow are located on the exterior surface of the confined space, aligned with the unit cubes, as shown in Figure 3. To keep expenses down, you must accomplish this task while using the minimum number of elbows.

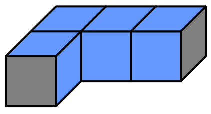


Figure 1

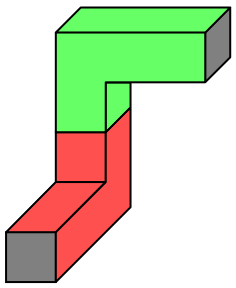


Figure 2

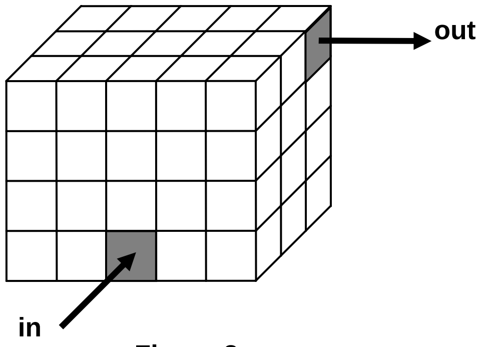


Figure 3

Input

The input consists of several test cases, each of which consists of a single line containing eleven input values separated by blanks. The input values for each test case are as follows.

The first three input values are integers (x_{max}, y_{max}, and z_{max}) that indicate the size of the confined space in the x, y, and z dimensions, respectively. Each unit cube in the confined space can be identified by coordinates (x, y, z) where $1 \leq x \leq x_{max}$, $1 \leq y \leq y_{max}$, and $1 \leq z \leq z_{max}$. x_{max}, y_{max}, and z_{max} are all positive and not greater than 20.

The next three input values are integers that indicate the location of the inflow by identifying the x, y, and z coordinates of the unit cube that connects to the inflow.

The next input value is a two-character string that indicates the direction of the inward flow, using one of the following codes: $+x$, $-x$, $+y$, $-y$, $+z$, $-z$. The inflow connection is on the face of the unit cube that receives this inward flow. For example, if the data specifies an inflow direction of $+y$, the inflow connection is on the face of the unit cube that faces in the negative y direction.

The next three input values are integers that indicate the location of the outflow by identifying the x, y, and z coordinates of the unit cube that connects to the outflow.

The last input value is a two-character string that indicates the direction of the outward flow, using the same codes described above. The outflow connection is on the face of the unit cube that generates this outward flow. For example, if the data specifies an outflow direction of +*y*, the outflow connection is on the face of the unit cube that faces in the positive *y* direction.

The last line of the input file consists of a single zero to indicate end of input.

Output

For each test case, print the case number (starting with 1) followed by the minimum number of elbows that are required to connect the inflow to the outflow without going outside the confined space. If the task cannot be accomplished with your supply of six elbow segments, print the word `Impossible` instead. Use the format in the sample data.

Sample Input	Output for the Sample Input

`5 4 3 3 1 1 +z 5 4 3 +x`
`5 4 3 3 1 1 +z 1 2 3 -x`
`0`
	`Case 1: 2`
	`Case 2: Impossible`

Problem B
Always an Integer
Input file: always.in

Combinatorics is a branch of mathematics chiefly concerned with counting discrete objects. For instance, how many ways can you pick two people out of a crowd of *n* people? Into how many regions can you divide a circular disk by connecting *n* points on its boundary with one another? How many cubes are in a pyramid with square layers ranging from 1×1 to $n \times n$ cubes?

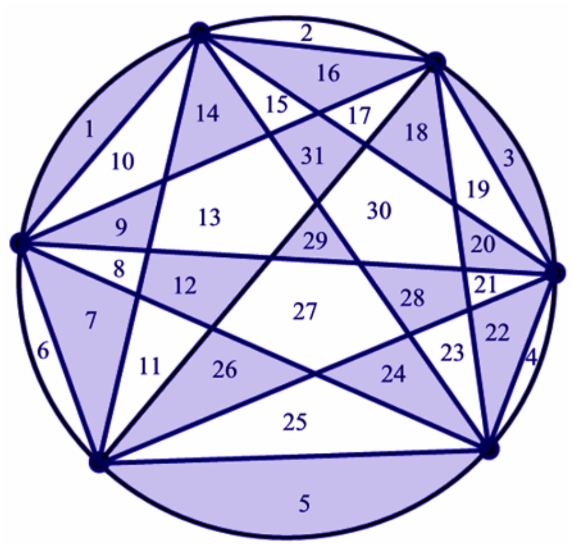


Figure 1: If we connect six points on the boundary of a circle, at most 31 regions are created.

Many questions like these have answers that can be reduced to simple polynomials in *n*. The answer to the first question above is *n*(*n*-1)/2, or (*n*^2-*n*)/2. The answer to the second is (*n*^4-6*n*^3+23*n*^2-18*n*+24)/24. The answer to the third is *n*(*n*+1)(2*n*+1)/6, or (2*n*^3+3*n*^2+*n*)/6. We write these polynomials in a standard form, as a polynomial with integer coefficients divided by a positive integer denominator.

These polynomials are answers to questions that can have integer answers only. But since they have fractional coefficients, they look as if they could produce non-integer results! Of course, evaluating these particular polynomials on a positive integer always results in an integer. For other polynomials of similar form, this is not necessarily true. It can be hard to tell the two cases apart. So that, naturally, is your task.

Input

The input consists of multiple test cases, each on a separate line. Each test case is an expression in the form (*P*)/*D*, where *P* is a polynomial with integer coefficients and *D* is a positive integer denominator. *P* is a sum of terms of the form *Cn*^*E*, where the coefficient *C* and the exponent *E* satisfy the following conditions:

1. *E* is an integer satisfying $0 \le E \le 100$. If *E* is 0, then *Cn*^*E* is expressed as *C*. If *E* is 1, then *Cn*^*E* is expressed as *Cn*, unless *C* is 1 or -1. In those instances, *Cn*^*E* is expressed as *n* or -*n*.
2. *C* is an integer. If *C* is 1 or -1 and *E* is not 0 or 1, then the *Cn*^*E* will appear as *n*^*E* or -*n*^E.
3. Only non-negative *C* values that are not part of the first term in the polynomial are preceded by +.
4. Exponents in consecutive terms are strictly decreasing.
5. *C* and *D* fit in a 32-bit signed integer.

See the sample input for details.

Input is terminated by a line containing a single period.

Output

For each test case, print the case number (starting with 1). Then print `Always an integer` if the test case polynomial evaluates to an integer for every positive integer *n*. Print `Not always an integer` otherwise. Print the output for separate test cases on separate lines. Your output should follow the same format as the sample output.

Sample Input	Output for the Sample Input

`(n^2-n)/2`
`(2n^3+3n^2+n)/6`
`(-n^14-11n+1)/3`
`.`
	`Case 1: Always an integer`
	`Case 2: Always an integer`
	`Case 3: Not always an integer`

Problem C
Conveyor Belt
Input file: belt.in

Many mechanical systems work with rotating shafts connected with conveyor belts. The shafts have a variety of sizes and rotate in either a clockwise or a counterclockwise manner. The exact way in which a belt will connect two shafts depends on their rotations, as shown in Figures 1 and 2.

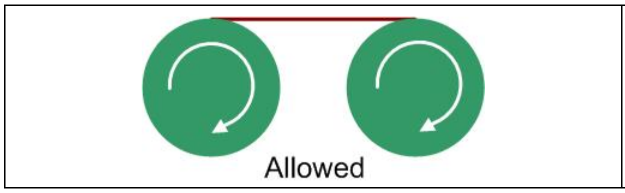


Figure 1

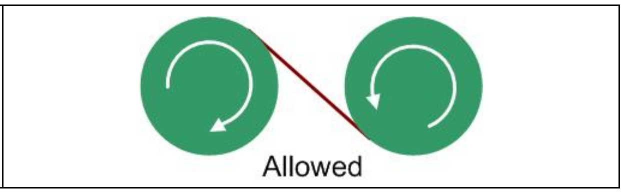


Figure 2

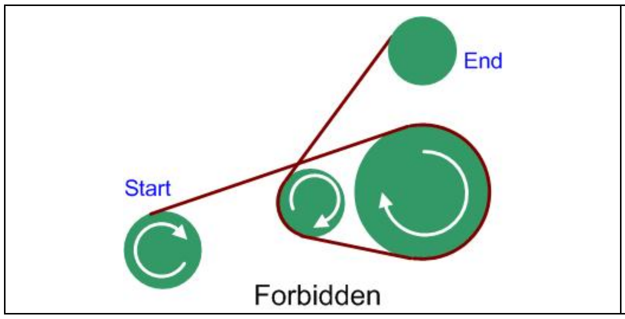


Figure 3

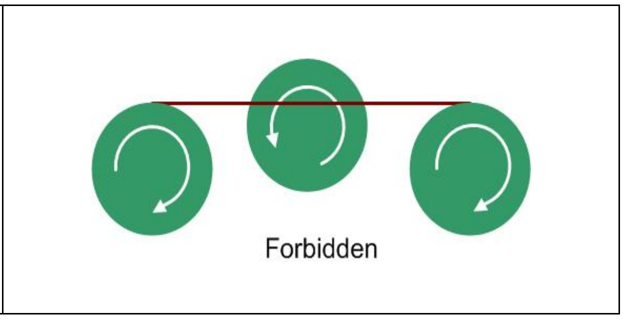


Figure 4

One task in setting up such mechanical systems is to link together two given shafts, subject to these constraints:

- If the two shafts being connected are too far apart, the belt may start to vibrate chaotically when perturbed slightly. To prevent this, you can connect shafts only when the distance between the points where the belt leaves one shaft and touches the other is less than some distance *d* (the exact value of *d* varies depending on the type of belt).
- No belt can cross over itself, as shown in Figure 3.
- No belt can pass through another shaft (or touch one rotating the wrong way), as shown in Figure 4.
- The belt is not a loop; it goes in only one direction, from the starting shaft to the ending shaft. The starting shaft "pushes" the belt and the ending shaft "pulls" it.

As an example, consider the problem of connecting shaft A to shaft D in Figure 5. Suppose that the distance needed to connect A to C (shown in blue dashed line) or to connect B to D is greater than the limit allowed. Then the shortest distance to connect A to D is shown in solid line, going from shaft A to B to C and then D. Notice that the connection cannot go from B to E and then D as the belt would cross itself.

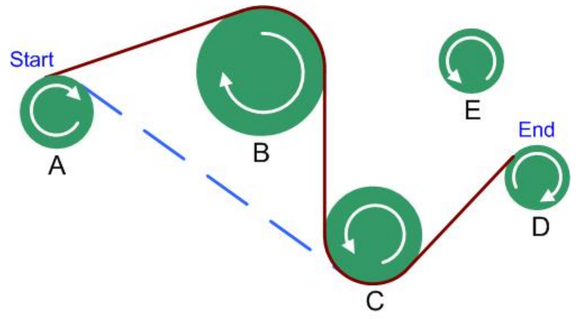


Figure 5

You must write a program that calculates the minimum length of the belt to connect the two given shafts.

Input

The input consists of multiple test cases. Each test case starts with a line containing an integer N ($1 \le N \le 20$) indicating the number of shafts, numbered from 0 to N -1. Starting on the next line are N four-tuples of the form $x\ y\ r\ s$, where x and y are the integer coordinates of a shaft center, r is the integer radius of the shaft, and s is either "C" for clockwise or "CC" for counterclockwise ($0 \le x, y \le 10000$ and $0 < r \le 1000$). Positive x is right and positive y is up. The first four-tuple specifies shaft 0, the second one shaft 1 and so on. These four-tuples may extend over multiple lines, though no four-tuple will be split across two lines. No two shafts touch or overlap. The last line of each test case contains two integers i and j indicating the starting and ending shafts, followed by a floating point value d specifying the maximum distance constraint.

The last test case is followed by a line containing a single zero.

Output

For each test case, print the case number (starting with 1) followed by the length of the path of minimum distance. Print `Cannot reach destination shaft` if the destination shaft cannot be reached. Use the format shown in the sample output.

When measuring the distance, start where the belt first leaves the starting shaft and end where the belt first touches the ending shaft. Your distance calculation must include both the length of belt between shafts as well as the distance the belt travels around intermediate shafts. Your answer should be rounded to the nearest hundredth, though you need not print trailing zeroes after the decimal point.

Sample Input

```
5
24 50 14 C 93 78 20 C 118 8 15 CC
167 32 13 C 159 88 15 CC
0 3 82.5
5
24 50 14 C 93 78 20 C 118 8 15 CC
167 32 13 C 159 88 15 C
0 3 82.5
5
24 50 14 C 93 78 20 C 118 8 15 CC
167 32 13 C 159 88 15 C
0 3 8.5
0
```

Output for the Sample Input

```
Case 1: length = 271
Case 2: length = 228.23
Case 3: Cannot reach destination shaft
```

Problem D
The Hare and the Hounds
Input file: hounds.in

A *hare and hounds* road rally requires contestants (the "hounds") to identify a route of one or more roads selected by the organizer (the "hare"). Both parties move over roads that meet at various intersections. Upon entering any intersection (except special ones to be described shortly), both the hound and the hare select routes using the *main road rule*. The main road rule is always "straight as possible," meaning make the smallest turn (perhaps none) necessary to continue. If there are two such choices possible (such as at some "Y" intersections), the main road rule dictates the rightmost (from the point of view of the hound) of the two acceptable alternatives should be selected.

At certain intersections, the hare may violate the main road rule by taking a random road away from the intersection (though never the original road used to reach the intersection). These intersections are marked by the hare as *choice points* (usually by a colored mark on the pavement in the intersection). When reaching a choice point, the hound must try each road leaving the intersection and travel that road (potentially traveling through other intersections) until reaching a *confirmation marker* (usually some flour dumped by the hare on the road) confirming the correct route selection. An incorrect route selection is indicated by one of the following:

- The hound travels a specified maximum distance from the choice point before reaching a confirmation marker.
- The hound reaches a "dead end" (an "intersection" with only one road to it) before reaching the confirmation marker.
- The hound reaches a choice point before reaching the confirmation marker. (The hare always places a confirmation marker on the route following a choice point. Also, the hare never returns to a previous choice point.)

After detecting an incorrect route, the hound must trace back along the route taken to the choice point and select a different route alternative. If the hound encounters the endpoint while looking for a confirmation marker, he ignores it.

When selecting routes from a choice point, the hound uses the main road rule in a slightly different way. The first road selected is the same as if the intersection were not a choice point. But if the hound must return to the choice point (i.e., the first road taken from the choice point was not part of the hare's route), then the hound chooses the second road using the main road rule from the direction with which he returns to the choice point (ignoring any direction that he has already tried as well as the direction that he originally arrived from). This process repeats each time the hound returns to the choice point until he finds the proper route. This multiple use of the main road rule occurs only at choice points. For the purposes of this problem, the hound will not remember the result of traveling down any road, even if he returns to it multiple times while exploring one choice point.

For this problem you will be given a road map (a configuration of roads and intersections), the list of choice point intersections, the placement of confirmation markers, the maximum distance from a choice point to a confirmation marker, the starting and ending intersections (neither of which will be choice points), and the direction to be used in leaving the starting intersection. Using this information, you will simulate the hound's search for the hare's route. You may assume that the hound, using the strategy described, will always discover and trace the hare's route.

Input

There may be multiple input test cases. The data for each case begins with a line containing 7 integers: *ncp* (number of choice point intersections), *nroad* (number of roads, never larger than 150), *ncm* (number of confirmation markers, never more than 100), *confdist* (maximum distance from a choice point intersection to the confirmation marker, at most 2000), *startisect* (starting intersection number), *endisect* (ending intersection number), and *startdir* (the direction of the starting road). Intersections are identified using integers between 1 and 100.

The starting line for each case is immediately followed by a line containing *ncp* integers giving the identifying numbers of the choice point intersections.

Next there are *nroad* lines. Roads are identified using sequential integers starting with 1, matching the order in which their specification lines appear in the input. Each such line contains 5 integers: the identifying numbers of the two intersections connected by the road, the compass directions (from 0 to 359 degrees, 0 is north, 90 is east) with which the road leaves the intersections, and the length of the road. Each road can be traveled in both directions and no two roads will enter an intersection at the same angle.

Finally there are *ncm* lines that identify the placement of the confirmation markers. Each of these lines contains three integers giving the identifying number of an intersection, the identifying number of a road leaving that intersection, and the distance from the intersection to the confirmation marker. Confirmation markers will not be placed at intersections. Confirmation markers also will not be dropped along roads that start/end in the same intersection.

Input for the last case is followed by a line containing 7 zeroes.

Output

For each test case, print the case number (starting with 1), the length of the hare's route, the length of the hound's search (including all incorrect paths taken at choice points), and the road numbers in the hare's route in the order the hare traveled them, using the format shown in the sample data. Print a blank line after the output for each case.

Sample Input

```
1 5 1 3 3 1 180
2
2 4 180 0 5
1 4 180 90 6
4 2 270 270 5
3 2 180 0 8
1 2 270 90 3
4 3 3
0 0 0 0 0 0 0
```

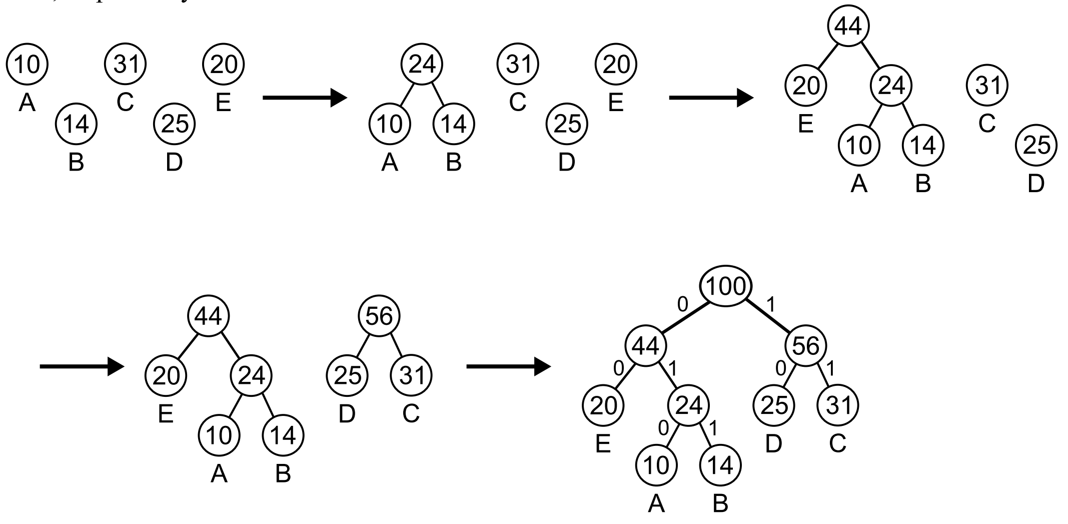
Output for the Sample Input

```
Case 1:
   Length of hare's route is 19
   Length of hound's search is 31
   Route: 4 3 2
```

Problem E
Huffman Codes
Input file: huffman.in

Dan McAmbi is a member of a crack counter-espionage team and has recently obtained the partial contents of a file containing information vital to his nation's interests. The file had been compressed using Huffman encoding. Unfortunately, the part of the file that Dan has shows only the Huffman codes themselves, not the compressed information. Since Huffman codes are based on the frequencies of the characters in the original message, Dan's boss thinks that some information might be obtained if Dan can reverse the Huffman encoding process and obtain the character frequencies from the Huffman codes. Dan's gut reaction to this is that any given set of codes could be obtained from a wide variety of frequency distributions, but his boss is not impressed with this reasoned analysis. So Dan has come to you to get more definitive proof to take back to his boss.

Huffman encoding is an optimal data compression method if you know in advance the relative frequencies of letters in the text to be compressed. The method works by first constructing a *Huffman tree* as follows. Start with a forest of trees, each tree a single node containing a character from the text and its frequency (the character value is used only in the leaves of the resulting tree). Each step of the construction algorithm takes the two trees with the lowest frequency values (choosing arbitrarily if there are ties), and replaces them with a new tree formed by joining the two trees as the left and right subtrees of a new root node. The frequency value of the new root is the sum of the frequencies of the two subtrees. This procedure repeats until only one tree is left. An example of this is shown below, assuming we have a file with only 5 characters – A, B, C, D and E – with frequencies 10%, 14%, 31%, 25% and 20%, respectively.



After you have constructed a Huffman tree, assign the Huffman codes to the characters as follows. Label each left branch of the tree with a 0 and each right branch with a 1. Reading down from the root to each character gives the Huffman code for that character. The tree above results in the following Huffman codes: A - 010, B - 011, C - 11, D - 10 and E - 00.

For the purpose of this problem, the tree with the lower frequency *always* becomes the left subtree of the new tree. If both trees have the same frequencies, either of the two trees can be chosen as the left subtree. Note that this means that for some frequency distributions, there are several valid Huffman encodings.

The same Huffman encoding can be obtained from several different frequency distributions: change 14% to 13% and 31% to 32%, and you still get the same tree and thus the same codes. Dan wants you to write a program to determine the total number of distinct ways you could get a given Huffman encoding, assuming that all percentages are positive integers. Note that two frequency distributions that differ only in the ordering of their percentages (for example 30% 70% for one distribution and 70% 30% for another) are not distinct.

Input

The input consists of several test cases. Each test case consists of a single line starting with a positive integer *n* ($2 \leq n \leq 20$), which is the number of different characters in the compressed document, followed by *n* binary strings giving the Huffman encoding of each character. You may assume that these strings are indeed a Huffman encoding of some frequency distribution (though under our additional assumptions, it may still be the case that the answer is 0 – see the last sample case below).

The last test case is followed by a line containing a single zero.

Output

For each test case, print a line containing the test case number (beginning with 1) followed by the number of distinct frequency distributions that could result in the given Huffman codes.

Sample Input

```
5 010 011 11 10 00
8 00 010 011 10 1100 11010 11011 111
8 1 01 001 0001 00001 000001 0000001 0000000
0
```

Output for the Sample Input

```
Case 1: 3035
Case 2: 11914
Case 3: 0
```

Problem F
Glenbow Museum
Input file: museum.in

The famous Glenbow Museum in Calgary is Western Canada's largest museum, with exhibits ranging from art to cultural history to mineralogy. A brand new section is being planned, devoted to brilliant computer programmers just like you. Unfortunately, due to lack of space, the museum is going to have to build a brand new building and relocate into it.

The size and capacity of the new building differ from those of the original building. But the floor plans of both buildings are orthogonal polygons. An orthogonal polygon is a polygon whose internal angles are either 90° or 270°. If 90° angles are denoted as R (Right) and 270° angles are denoted as O (Obtuse) then a string containing only R and O can roughly describe an orthogonal polygon. For example, a rectangle (Figure 1) is the simplest orthogonal polygon and it can be described as RRRR (the angles are listed in counter-clockwise order, starting from any corner). Similarly, a cross-shaped orthogonal polygon (Figure 2) can be described by the sequence RRORRORRORRO, RORRORRORROR, or ORRORRORRORR. These sequences are called *angle strings.*

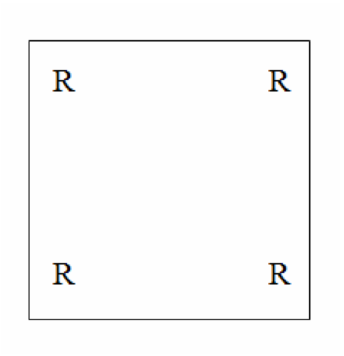


Figure 1: A rectangle

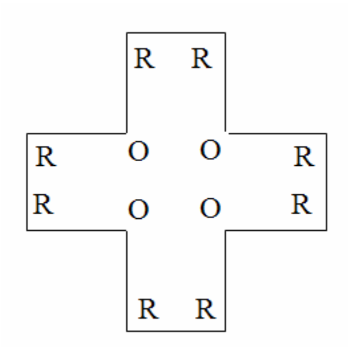


Figure 2: A cross-shaped polygon

Of course, an angle string does not completely specify the shape of a polygon – it says nothing about the length of the sides. And some angle strings cannot possibly describe a valid orthogonal polygon (RRROR, for example).

To complicate things further, not all orthogonal polygons are acceptable floor plans for the museum. A museum contains many valuable objects, and these objects must be guarded. Due to cost considerations, no floor can have more than one guard. So a floor plan is acceptable only if there is a place within the floor from which one guard can see the entire floor. Similarly, an angle string is acceptable only if it describes at least one acceptable polygon. Note that the cross-shaped polygon in Figure 2 can be guarded by someone standing in the center, so it is acceptable. Thus the angle string RRORRORRORRO is acceptable, even though it also describes other polygons that cannot be properly guarded by a single guard.

Help the designers of the new building determine how many acceptable angle strings there are of a given length.

Input

The input file contains several test cases. Each test case consists of a line containing a positive integer L ($1 \le L \le 1000$), which is the desired length of an angle string.

The input will end with a line containing a single zero.

Output

For each test case, print a line containing the test case number (beginning with 1) followed by the number of acceptable angle strings of the given length. Follow the format of the sample output.

Sample Input

```
4
6
0
```

Output for the Sample Input

```
Case 1: 1
Case 2: 6
```

Problem G
Net Loss
Input file: netloss.in

Rose N. Blatt is designing an embedded neural network to place inside a cell phone. When trained by the phone's owner, the neural network will enable the user to dictate text messages in a hands-free way. The key idea in Rose's design is the use of complicated polynomial response functions in each of the nodes of the network (rather than the more traditional thresholding functions used in many other neural nets). Figure 1 shows a portion of such a neural network (the polynomials are not accurately graphed).

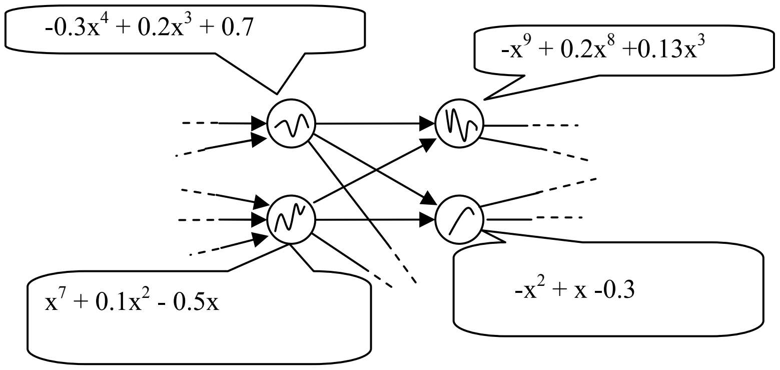


Figure 1

When Rose was ready to select her polynomials, she discovered a problem. Due to the limited amount of memory available, she did not have enough space to store all of the coefficients of the polynomials in her network. She has decided to use an approximation to each polynomial in the form of a continuous polygonal curve with two segments, $y = a_1x + a_0$ and $y = b_1x + b_0$. The segments meet at a point whose x-coordinate, c, is between -1 and +1. Rose wants the approximation to be the best in the sense that the distance between p and the approximation function g is minimal. We define the distance between p and g as the integral of the square of their difference:

$$d(p,g) = \int_{-1}^{1} (p(x)-g(x))^2 \, dx$$

For instance, if the polynomial is $x^2 - 0.2$, then the best polygonal approximation, with lines meeting at $c = 0$, is shown in Figure 2 (the dotted line shows the graph of the polygonal approximation).

$$g(x) = \begin{cases} -x - 0.367, & -1 \le x \le 0 \\ x - 0.367, & 0 \le x \le 1 \end{cases}$$

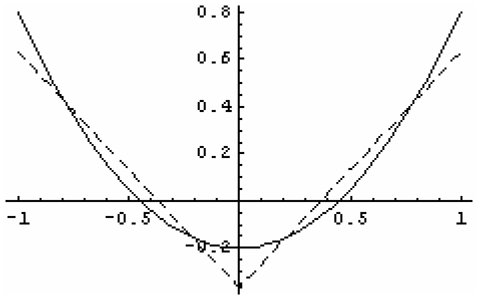


Figure 2

In the few bytes that are available for each node, Rose can store the values of a_1, a_0, b_1, b_0, and c as signed numbers. Fortunately Rose has a program that supplies her with a good guess for the value of c. Given this value, you are asked to help Rose find the optimal values for a_1, a_0, b_1, and b_0 in the approximations to her polynomials.

Input

The input file contains multiple test cases. Each test case consists of three lines. The first line contains a positive integer n, $1 \le n \le 10$, representing the degree of the polynomial $p(x)$. This is followed by a line containing $n + 1$ numbers between -1 and 1 inclusive, which are the coefficients of $p(x)$ from highest order term down to the constant term, expressed with at most three places after the decimal point. The last line for each test case contains the value for c, $-1 < c < 1$, expressed with at most three places after the decimal point.

A line containing the integer zero follows the last test case.

Output

For each test case, print the case number (beginning with 1) and the four optimal values, displaying each with exactly three places after the decimal point. The first and second values are the parameters a_1 and a_0 of the line segment $y = a_1x + a_0$ defining g in the range $-1 \le x \le c$. The third and fourth values are the parameters b_1 and b_0 of the line segment $y = b_1x + b_0$ defining g in the range $c \le x \le 1$. The distance $d(p,g)$ between p and g (as defined earlier) should be the minimum over all such choices for a_1, a_0, b_1, and b_0.

Sample Input

```
2
1.0 0.0 -0.2
0.0
3
1 0 -1 0
0.707
0
```

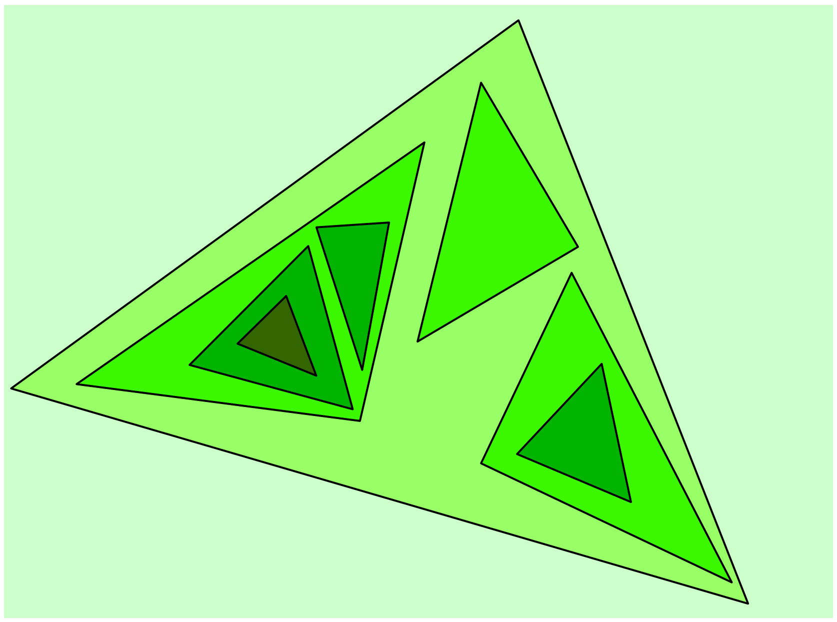
Output for the Sample Input

```
Case 1: -1.000 -0.367 1.000 -0.367
Case 2: -0.499 -0.036 1.198 -1.236
```

Problem H
Painter
Input file: painter.in

You probably never heard of the painter Peer. He is not well known, much to his regret. Peer was one of the inventors of *monochromy*, which means that each of his paintings has a single color, but in different shades. He also believed in the use of simple geometric forms.

During his triangle period, Peer drew triangles on a rectangular canvas, making sure their borders did not intersect. He would then choose a color, and fill the regions. Peer would paint the outermost region (the canvas itself) with the lightest shade of the color chosen. Then step by step, he would fill more inner regions with a darker shade of the same color. The image below is one of his “Forms in Green” paintings.



In a way the process was quite mechanical. The only thing Peer considered difficult was to decide, after drawing the triangles, how many different shades he would need. You must write a program to do that calculation for him. Your program will have a collection of triangles as its input. It should calculate the number of different shades needed to paint the regions according to the given rule.

Your program must also detect the rare times that Peer makes a mistake and draws triangles that intersect. Two triangles are considered intersecting if the edges of one triangle have at least one point in common with the edges of the other. In that case, the collection of triangles is invalid.

Input

The input file contains multiple test cases. The first line of each test case contains a single non-negative integer n ($n \leq 100000$), which is the number of triangles in the test case. The following n lines of the test case contain the descriptions of triangles in the format $x_1\ y_1\ x_2\ y_2\ x_3\ y_3$, where x_i, y_i are integers ($-100000 < x_i, y_i < 100000$) that are the coordinates of the vertices of the triangles. The three points are guaranteed not to be collinear.

The last test case is followed by -1 on a line by itself.

Output

For each test case, print the case number (beginning with 1) and the number of shades needed to fill the regions if the test case is valid. Print the word `ERROR` if the test case is invalid (two or more triangles in the test case intersect).

Sample Input

```
8
8 3 8 4 7 4
14 13 -1 9 9 0
1 8 7 7 4 10
5 10 11 8 13 12
9 10 11 10 11 9
2 7 9 1 10 6
5 5 5 6 8 6
9 2 9 5 6 4
2
0 0 1 0 0 1
2 0 1 1 1 -1
-1
```

Output for the Sample Input

```
Case 1: 5 shades
Case 2: ERROR
```

Problem I
Password Suspects
Input file: password.in

You are the computer whiz for the secret organization known as the *Sneaky Underground Smug Perpetrators of Evil Crimes and Thefts*. The target for SUSPECT's latest evil crime is their greatest foe, the *Indescribably Clever Policemen's Club*, and everything is prepared. Everything, except for one small thing: the secret password for ICPC's main computer system.

The password is known to consist only of lowercase letters 'a'-'z'. Furthermore, through various sneaky observations, you have been able to determine the length of the password, as well as a few (possibly overlapping) substrings of the password, though you do not know exactly where in the password they occur.

For instance, say that you know that the password is 10 characters long, and that you have observed the substrings "`hello`" and "`world`". Then the password must be either "`helloworld`" or "`worldhello`".

The question is whether this information is enough to reduce the number of possible passwords to a reasonable amount. To answer this, your task is to write a program that determines the number of possible passwords and, if there are at most 42 of them, prints them all.

Input

The input file contains several test cases. Each test case begins with a line containing two integers N and M ($1 \le N \le 25$, $0 \le M \le 10$), giving the length of the password and the number of known substrings respectively. This is followed by M lines, each containing a known substring. Each known substring consists of between 1 and 10 lowercase letters 'a'-'z'.

The last test case is followed by a line containing two zeroes.

Output

For each test case, print the case number (beginning with 1) followed by Y `suspects`, where Y is the number of possible passwords for this case. If the number of passwords is at most 42, then output all possible passwords in alphabetical order, one per line.

The input will be such that the number of possible passwords at most 10^{15}.

Sample Input

```
10 2
hello
world
10 0
4 1
icpc
0 0
```

Output for the Sample Input

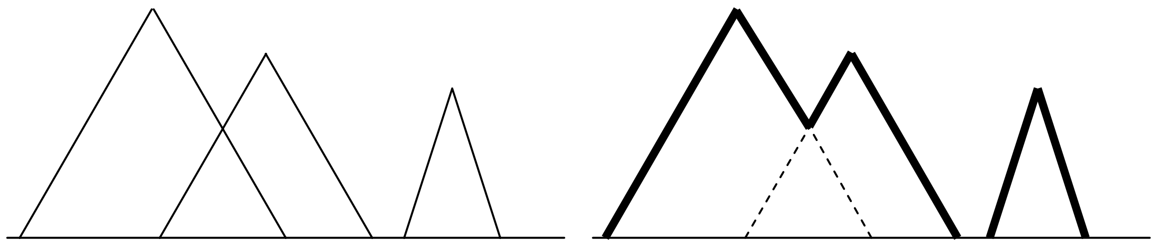
```
Case 1: 2 suspects
helloworld
worldhello
Case 2: 141167095653376 suspects
Case 3: 1 suspects
icpc
```

Problem J
The Sky is the Limit
Input file: skyline.in

The city of Banff hired an advertising agency to promote the city's attractions to potential visitors. One of the planned slogans stated that the mountain ranges around the city form the most beautiful skyline in Canada. But the Institute for Consumer Protection in Canada (ICPC) decided that "the most beautiful skyline" was a subjective and unverifiable claim, and could therefore be considered misleading.

The advertising agency then came up with the slogan "Banff – the longest skyline in Canada." Although not as catchy, it is hopefully verifiable, and therefore admissible under Canada's tricky advertising laws.

This is where you come in. What the advertising agency needs is a program that determines the length of a skyline. Consider each mountain as a two-dimensional triangle having two upper sides the same length. A skyline is the outline of one or more mountains. The skyline's length is the total length of the outline. The left illustration below shows three mountains. The right illustration shows (with bold lines) the skyline and (with dashed lines) the portion of the mountains' upper edges that are not part of the skyline. Note that parts of the horizon line that lie between mountains are not considered part of the skyline.



Input

Each input file contains one or more test cases, which are descriptions of mountain ranges. Each description starts with a line containing a positive integer *N,* which specifies the number of mountains in the range. Each of the next *N* lines describes a mountain with three integers *X*, *H*, and *B*, which specify the horizontal position of the mountain's peak relative to some fixed point, the height of the peak, and the width of the base of the mountain, respectively. The base of each mountain coincides with a horizontal line. The values satisfy the conditions $N \le 100$, $H > 0$, and $B > 0$.

The last test case is followed by a line containing a zero.

Output

For each test case, print the case number (beginning with 1) and the length of the skyline. Print the length rounded to the nearest integer, with 0.5 rounded up. Print a blank line after the output of each test case. Use the format shown in the sample output below.

Sample Input

```
1
100 50 100
3
20  30  35
37 24   29
60 20 13
0
```

Output for the Sample Input

```
Case 1: 141

Case 2: 138
```

This page is intentionally blank.

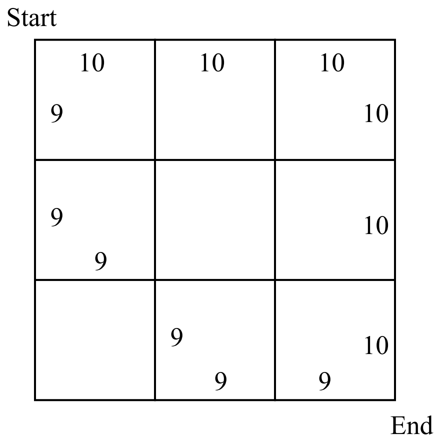
Problem K
Steam Roller
Input file: steam.in

Johnny drives a steam roller, which like all steam rollers is slow and takes a relatively long time to start moving, change direction, and brake to a full stop. Johnny has just finished his day's work and is driving his steam roller home to see his wife. Your task is to find the fastest path for him and his steam roller.

The city where Johnny lives has a regular structure (the streets form an orthogonal system). The city streets are laid out on a rectangular grid of intersections. Each intersection is connected to its neighbors (up to four of them) by a street. Each street is exactly one block long. When Johnny enters a street, he must always travel to the other end (continue to the next intersection). From that point, he can continue in any of the four possible directions to another intersection, and so on.

By studying the road conditions of the streets, Johnny has calculated the time needed to go from one end to the other of every street in town. The time is the same for both directions. However, Johnny's calculations hold only under the ideal condition that the steam roller is already in motion when it enters a street and does not need to accelerate or brake. Whenever the steam roller changes direction at a intersection directly before or after a street, the estimated ideal time for that street must be doubled. The same holds if the roller begins moving from a full stop (for example at the beginning of Johnny's trip) or comes to a full stop (for example at the end of his trip).

The following picture shows an example. The numbers show the "ideal" times needed to drive through the corresponding streets. Streets with missing numbers are unusable for steam rollers. Johnny wants to go from the top-left corner to the bottom-right one.



The path consisting of streets labeled with 9's seems to be faster at the first sight. However, due to the braking and accelerating restrictions, it takes double the estimated time for every street on the path, making the total time 108. The path along the streets labeled with 10's is faster because Johnny can drive two of the streets at the full speed, giving a total time of 100.

Input

The input consists of several test cases. Each test case starts with six positive integer numbers: R, C, r_1, c_1, r_2, and c_2. R and C describe the size of the city, r_1, c_1 are the starting coordinates, and r_2, c_2 are the coordinates of Johnny's home. The starting coordinates are different from the coordinates of Johnny's home. The numbers satisfy the following condition: $1 \leq r_1, r_2 \leq R \leq 100$, $1 \leq c_1, c_2 \leq C \leq 100$.

After the six numbers, there are C-1 non-negative integers describing the time needed to drive on streets between intersections (1,1) and (1,2), (1,2) and (1,3), (1,3) and (1,4), and so on. Then there are C non-negative integers describing the time need to drive on streets between intersections (1,1) and (2,1), (1,2) and (2,2), and so on. After that, another $C - 1$ non-negative integers describe the next row of streets across the width of the city. The input continues in this way to describe all streets in the city. Each integer specifies the time needed to drive through the corresponding street (not higher than 10000), provided the steam roller proceeds straight through without starting, stopping, or turning at either end of the street. If any combination of one or more of these events occurs, the time is multiplied by two. Any of these integers may be zero, in which case the corresponding street cannot be used at all.

The last test case is followed by six zeroes.

All numbers are separated with at least one whitespace character (space, tab, or newline), but any amount of additional whitespace (including empty lines) may be present to improve readability.

Output

For each test case, print the case number (beginning with 1) followed by the minimal time needed to go from intersection x_1, y_1 to x_2, y_2. If the trip cannot be accomplished (due to unusable streets), print the word `Impossible` instead.

Sample Input

```
4 4 1 1 4 4
 10  10  10
9  0  0  10
  0  0  0
9  0  0  10
  9  0  0
0  9  0  10
  0  9  9

2 2 1 1 2 2 0 1 1 0

0 0 0 0 0 0
```

Output for the Sample Input

```
Case 1: 100
Case 2: Impossible
```