Problem A: Tree Grafting

Input: `graft.in`
Output: `graft.out`

Trees have many applications in computer science. Perhaps the most commonly used trees are rooted binary trees, but there are other types of rooted trees that may be useful as well. One example is ordered trees, in which the subtrees for any given node are ordered. The number of children of each node is variable, and there is no limit on the number. Formally, an ordered tree consists of a finite set of nodes T such that

- there is one node designated as the root, denoted root(T);
- the remaining nodes are partitioned into subsets T1, T2, ..., Tm, each of which is also a tree (subtrees).

Also, define root(T1), ..., root(Tm) to be the children of root(T), with root(Ti) being the i−th child. The nodes root(T1), ..., root(Tm) are siblings.

It is often more convenient to represent an ordered tree as a rooted binary tree, so that each node can be stored in the same amount of memory. The conversion is performed by the following steps:

1. remove all edges from each node to its children;
2. for each node, add an edge to its first child in T (if any) as the left child;
3. for each node, add an edge to its next sibling in T (if any) as the right child.

This is illustrated by the following:

```
   0                             0
 / | \                          /
1  2  3       ===>             1
  / \                           \
 4   5                           2
                                / \
                               4   3
                                \
                                 5
```

In most cases, the height of the tree (the number of edges in the longest root−to−leaf path) increases after the conversion. This is undesirable because the complexity of many algorithms on trees depends on its height.

You are asked to write a program that computes the height of the tree before and after the conversion.

Input

The input is given by a number of lines giving the directions taken in a depth−first traversal of the trees. There is one line for each tree. For example, the tree above would give dudduduudu, meaning 0 down to 1, 1 up to 0, 0 down to 2, etc. The input is terminated by a line whose first character is #. You may assume that each tree has at least 2 and no more than 10000 nodes.

Output

For each tree, print the heights of the tree before and after the conversion specified above. Use the format:

```
        Tree t: h1 => h2
```

where t is the case number (starting from 1), h1 is the height of the tree before the conversion, and h2 is the height of the tree after the conversion.

Sample Input

```
dudduduudu
ddddduuuuu
ddddduduuuu
dddduududuu
#
```

Sample Output

```
Tree 1: 2 => 4
Tree 2: 5 => 5
Tree 3: 4 => 5
Tree 4: 4 => 4
```

Problem B: Look and Say

Input: `lookandsay.in`
Output: `lookandsay.out`

The look and say sequence is defined as follows. Start with any string of digits as the first element in the sequence. Each subsequent element is defined from the previous one by "verbally" describing the previous element. For example, the string 122344111 can be described as "one 1, two 2's, one 3, two 4's, three 1's". Therefore, the element that comes after 122344111 in the sequence is 1122132431. Similarly, the string 101 comes after 1111111111. Notice that it is generally not possible to uniquely identify the previous element of a particular element. For example, a string of 112213243 1's also yields 1122132431 as the next element.

Input

The input consists of a number of cases. The first line gives the number of cases to follow. Each case consists of a line of up to 1000 digits.

Output

For each test case, print the string that follows the given string.

Sample Input

```
3
122344111
1111111111
12345
```

Sample Output

```
1122132431
101
1112131415
```

Problem C: Server Relocation

Input: `server.in`
Output: `server.out`

Michael has a powerful computer server that has hundreds of parallel processors and terabytes of main memory and disk space. Many important computations run continuously on this server, and power must be supplied to the server without interruption.

Michael's server must be moved to accommodate new servers that have been purchased recently. Fortunately, Michael's server has two redundant power supplies−−−as long as at least one of the two power supplies is connected to an electrical outlet, the server can continue to run. When the server is connected to an electrical outlet, it can be moved to any location which is not further away from the outlet than the length of the cord used to connect to the outlet.

Given which outlet Michael's server is plugged into initially and finally, and the locations of outlets in the server room, you should determine the smallest number of times you need to plug a cord into an electrical outlet in order to move the server while keeping the server running at all times. Note that, in the initial and final configuration, only one cord is connected to the power outlet.

Input

The first line of input is an integer giving the number of cases to follow. For each case, the first line is of the form

OUTLETS OUTLET_INITIAL OUTLET_FINAL LENGTH1 LENGTH2

where

- OUTLETS is the number of outlets in the server room (2 <= OUTLETS <= 1000).
- OUTLET_INITIAL is the index (starting from 1) of the outlet the server is initially connected to.
- OUTLET_FINAL is the index (starting from 1) of the outlet the server is finally connected to.
- LENGTH1 and LENGTH2 are the positive lengths of the two power cords, with at most three digits of precision after the decimal point (0 < LENGTH1, LENGTH2 <= 30000).

These are followed by OUTLETS lines giving the integer coordinates of the wall outlets, one per line, with the k−th line giving the location of the k−th outlet. All coordinates are specified as two integers (x− and y−coordinates) separated by a space, with absolute values at most 30000. You may assume that all coordinates are distinct, and that the initial outlet and the final outlet are different.

Output

For each case, print the minimum number of times you need to plug a cord into an electrical outlet in order to move the server to the final location while keeping the server running at all times. If this is not possible, print "Impossible".

Sample Input

```
2
4 1 4 2.000 1.000
0 0
0 4
4 0
4 4
9 1 4 2.000 3.000
0 7
-6 2
-3 3
6 2
-6 -3
3 -3
6 -3
-3 -7
0 -7
```

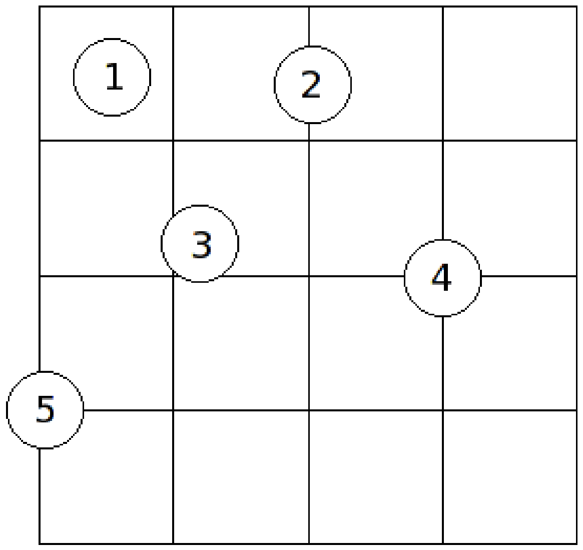
Sample Output

```
Impossible
8
```

Problem D: Coin Toss

Input: `cointoss.in`
Output: `cointoss.out`

In a popular carnival game, a coin is tossed onto a table with an area that is covered with square tiles in a grid. The prizes are determined by the number of tiles covered by the coin when it comes to rest: the more tiles it covers, the better the prize. In the following diagram, the results from five coin tosses are shown:



In this example:

- coin 1 covers 1 tile
- coin 2 covers 2 tiles
- coin 3 covers 3 tiles
- coin 4 covers 4 tiles
- coin 5 covers 2 tiles

Notice that it is acceptable for a coin to land on the boundary of the playing area (coin 5). In order for a coin to cover a tile, the coin must cover up a positive area of the tile. In other words, it is not enough to simply touch the boundary of the tile. The center of the coin may be at any point of the playing area with uniform probability. You may assume that (1) the coin always comes to a rest lying flat, and (2) the player is good enough to guarantee that the center of the coin will always come to rest on the playing area (or the boundary).

The probability of a coin covering a certain number of tiles depends on the tile and coin sizes, as well as the number of rows and columns of tiles in the playing area. In this problem, you will be required to write a program which computes the probabilities of a coin covering a certain number of tiles.

Input

The first line of input is an integer specifying the number of cases to follow. For each case, you will be given

4 integers m, n, t, and c on a single line, separated by spaces. The playing area consists of m rows and n columns of tiles, each having side length t. The diameter of the coin used is c. You may assume that 1 <= m, n <= 5000, and 1 <= c < t <= 1000.

Output

For each case, print the case number on its own line. This is followed by the probability of a coin covering 1 tile, 2 tiles, 3 tiles, and 4 tiles each on its own line. The probability should be expressed as a percentage rounded to 4 decimal places. Use the format as specified in the sample output. You should use double−precision floating−point numbers to perform the calculations. "Negative zeros" should be printed without the negative sign.

Separate the output of consecutive cases by a blank line.

Sample Input

```
3
5 5 10 3
7 4 25 20
10 10 10 4
```

Sample Output

```
Case 1:
Probability of covering 1 tile  = 57.7600%
Probability of covering 2 tiles = 36.4800%
Probability of covering 3 tiles = 1.2361%
Probability of covering 4 tiles = 4.5239%

Case 2:
Probability of covering 1 tile  = 12.5714%
Probability of covering 2 tiles = 46.2857%
Probability of covering 3 tiles = 8.8293%
Probability of covering 4 tiles = 32.3135%

Case 3:
Probability of covering 1 tile  = 40.9600%
Probability of covering 2 tiles = 46.0800%
Probability of covering 3 tiles = 2.7812%
Probability of covering 4 tiles = 10.1788%
```

Problem E: Vacation Rentals

Input: `vacation.in`
Output: `vacation.out`

The Fifth Season Resort consists of a number of condominiums which are frequently occupied by their owners. At other times, however, they are available as vacation rentals. Since the resort has no more than 26 condominiums, they are identified by upper case letters.

One day the resort manager's telephone rings. She receives a reservation request for a vacation rental with an arrival date of December 2 and a departure date of December 9. She looks at the table of reservations, but doesn't find a condominium that would be available for the entire period. Most of the existing reservations were made by the owners of the respective condominiums (who want to stay in their own units), so it is not desirable to move an existing reservation from one unit to another. As she continues to scrutinize the table of reservations, however, she has an idea and says: "I can put you up in unit B for the first three nights, and transfer you to unit F for the rest of your stay. Will that work?" The person agrees and the reservation is made. Notice that reservations are done by "nights", so that a one night reservation implies that the guest departs one day after the arrival.

The goal of this problem is to satisfy such reservation requests (without changing existing reservations), with a minimum number of transfers (from one unit to another) during the requested period.

Input

The input consists of a number of cases. The first line of each case contains two positive integers M and N. M is the number of consecutive days for which the resort's manager has a reservation table, and N is the number of units (condominiums) in the resort. The units are labeled by upper case letters starting at 'A'. There are at most 100 days and at least 3 rooms in the reservation table. The days are numbered 1, 2, ..., M for simplicity.

The reservation table is given in the next M lines. Each line (row) of the table refers to a particular day (in the order 1, 2, 3, etc.), and each column of the table to a particular unit of the resort (in the order 'A', 'B', 'C', etc.). An entry of 'X' means that the corresponding unit is reserved for that day, while an 'O' means that the unit is available.

The reservation table is followed by one line of input, the reservation request, consisting of two integers: the arrival date and the departure date. The arrival date is in the range 1..M. The departure date is greater than the arrival date and less than or equal to M+1.

The end of input is indicated by M = N = 0.

Output

For each test case, first print the case number followed by a colon and a blank line. If the reservation request can be met, the output will show a reservation schedule with a minimum number of transfers (from one unit to another) during the stay at the resort. Each line of the schedule corresponds to a consecutive stay in the same unit, and should be printed in the following way:

```
<unit>: <start date>-<end date>
```

where `<unit>` is the unit, `<start date>` is the date on which the guests moves into the unit, and `<end date>` is the date on which the guests moves out of the unit. The lines in the schedule should be ordered in ascending order by the start date.

Tie breaking rule. There may be several schedules with a minimum number of transfers. In these cases, choose the schedule which uses the lowest "unit label" in the first day (so unit A is given preference to unit B). If there is still a tie, choose the schedule with the lowest unit number in the second day, and so on.

If the reservation request cannot be met, print the line:

```
Not available
```

instead of the schedule.

Separate the output of consecutive cases by a blank line.

Sample Input

```
10 7
XXXXXXX
XOXXXXO
XOXXXXO
XOXXXOX
OXXOXOX
XOXOXOX
OXXOXOX
OXXXXOX
XXXXXXX
XXXXXXX
2 9
0 0
```

Sample Output

```
Case 1:

B: 2-5
F: 5-9
```

Problem F: Baseball

Input: `baseball.in`
Output: `baseball.out`

Baseball is a game full of statistics. But what good are these statistics? Maybe they can be used to predict the outcome of a game.

A baseball game consists of 9 innings. If the game is tied after 9 innings, one additional inning is played at a time until the game is no longer tied at the end of the additional inning. Each inning is divided into two halves, in which one team "attacks" and the other team "defends". The visiting team attacks in the first half, while the home team attacks in the second half. If the home team is leading at the start of the second half of the 9th inning, the game ends because the winner has already been determined.

Each team submits a "batting order" at the beginning of the game indicating the order in which its 9 players will bat during the attack half of the inning. In each half of an inning, the players in the attacking team bat according to the batting order. The first batter in the first inning is the first one in the batting order. Each subsequent batter is the next batter in the batting order. If the previous batter is the last player in the batting order, the next batter is the first player in the batting order again. In each subsequent inning, the first player to bat is the player following the last player who has batted in the batting order.

When a player is successful at bat (a hit), the player at bat advances to the first base, and any other players already on base advance by one base (in real games a player may advance by multiple bases, but we will not consider this case in this problem). A player must advance to first base, second base, third base, and finally the home base in order to score a run. When a player reaches the home base, he is returned to the bench to wait for his next opportunity to bat. The half of an inning continues indefinitely until 3 attacking players are unsuccessful at bat (each unsuccessful batter is an "out"). When this happens, any players left on first, second, or third base return to the bench and do not score. The team that scores the most runs at the end of the game wins.

In certain situations, a player may "sacrifice" himself in order to advance the players already on base. If a sacrifice is successful, the player at bat is out but each player already on base is advanced by one base. A player advancing to the home base this way scores a run, unless the sacrificed batter is the third out of the inning. In the latter case, the half of the inning is over and no run is scored. If a sacrifice is unsuccessful, the batter is out and none of the players advances. A batter will attempt to sacrifice whenever there is a player on second base with zero out, or when there is a player on third base with at most one out.

One of the most commonly cited statistics for players is the "hit percentage" (between 0 and 1) for each player, indicating the proportion of time the player is successful at bat. Similarly, the "sacrifice percentage" of a player is the proportion of time the player is successful at a sacrifice. In this problem, you will be given the hit percentage and the sacrifice percentage of each of the 9 players in a team as well as a batting order. You will be asked to simulate a baseball game. Random numbers are required in a simulation, and you will use the following random number generator:

```
x(n+1) = (x(n) * 25173 + 13849) % 65536
```

where `x(n)` is the previous random number and `x(n+1)` is the next random number. The calculations above should be performed using 32–bit integers. You should start with the "seed" `x(0) = 1`; the first random number generated by your program is `x(1)`. Each time the simulation needs to determine if a hit or a sacrifice is successful, it should generate the next random number. A hit (or a sacrifice) is successful if

```
random_number / 65536 <= hit (or sacrifice) percentage
```

The division in this formula is floating−point division. Do not reset the random number generator (i.e. resetting the seed to 1) except at the beginning of each game.

Input

The input consists of a number of games. The first line of the input file specifies the number of games to follow. Each game contains the batting order of the visiting team followed by the batting order of the home team. The batting order of each team starts with a line specifying the team name (at most 15 upper and lower case letters). The next 9 lines specify the 9 players, listed in the order they bat. Each of these lines contains the player's name (at most 15 upper and lower case characters), a space, then a floating−point number (to 3 decimal places) specifying his hit percentage, a space, and finally his sacrifice percentage (to 3 decimal places). You may assume that all hit percentages are between .200 and .400, and all sacrifice percentages are between .300 and .750. You may assume that no game will last more than 200 innings.

Output

For each game, print the game number as well as the visiting and home team names in the first line, as follows:

```
Game <x>: <visiting> vs. <home>
```

This is followed by a blank line.

Next print the players who score hits and runs, separately, for each inning in the order they occur in the game. Use the format as follows:

```
Inning 1:
Hits:
  <player1> <team>
  <player2> <team>

Runs:
  <player3> <team>
  <player4> <team>
```

Indent the list of players by two spaces. Print the player name and the team name right−justified in a field width of 15 (in addition to the indentation before the player's name and one space separation between the player's name and team name). If no one scores a hit or a run, print the single line

```
  none
```

(indented by two spaces) in the appropriate section instead of a list of players. Print a blank line after the output for each inning.

At the end of the game, print a summary on the number of runs and hits scored for each team, starting with the visiting team:

```
End of Game:
  <visiting> <x> runs, <x> hits
  <home> <x> runs, <x> hits
```

The summary should be indented by two spaces. Print the team name right–justified in a field width of 15 (in addition to the indentation).

Separate the output of consecutive games by a line containing 60 '=' signs.

Sample Input

```
1
Rangers
Young .213 .523
Kinsler .207 .602
Sosa .254 .300
Laird .220 .432
Byrd .206 .749
Wilkerson .236 .508
Catalanotto .272 .483
Teixeira .297 .573
Saltalamacchia .243 .632
BlueJays
Wells .378 .502
Hill .276 .544
Overbay .372 .694
McDonald .373 .618
Adams .320 .690
Rios .300 .450
Johnson .379 .559
Stairs .302 .621
Zaun .346 .515
```

Sample Output

```
Game 1: Rangers vs. BlueJays

Inning 1:
Hits:
             Hill        BlueJays
          Overbay        BlueJays
            Adams        BlueJays

Runs:
  none

Inning 2:
Hits:
  none

Runs:
  none

Inning 3:
Hits:
      Catalanotto         Rangers
            Wells        BlueJays
             Hill        BlueJays
            Adams        BlueJays
```

```
Runs:
            Wells        BlueJays

Inning 4:
Hits:
          Kinsler         Rangers

Runs:
  none

Inning 5:
Hits:
      Catalanotto         Rangers

Runs:
  none

Inning 6:
Hits:
             Sosa         Rangers
            Laird         Rangers
             Rios        BlueJays

Runs:
  none

Inning 7:
Hits:
        Wilkerson         Rangers
         Teixeira         Rangers
           Stairs        BlueJays

Runs:
  none

Inning 8:
Hits:
  none

Runs:
  none

Inning 9:
Hits:
  none

Runs:
  none

End of Game:
          Rangers 0 runs, 7 hits
         BlueJays 1 runs, 8 hits
```

Problem G: Team Work

Input: `teamwork.in`
Output: `teamwork.out`

A certain programming contest coach is frustrated with the lack of team work in his teams. He decided to demonstrate to his students the importance of team work by using the following well−known analogy: it is very easy to take a single piece of stick and snap it into two halves. But if you bundle three sticks together (i.e. the three members of the team) you now have to apply a lot more force to snap the sticks. The coach is certain that his teams will understand the importance of team work after this demonstration.

So the coach went out to the forest to collect the sticks. The coach knows that a demonstration should always be rehearsed to debug any problems. He ensures that it is practically impossible to snap three sticks bundled together and that it is very easy to snap individual sticks. Oh no! Every stick he has carefully collected snapped into smaller pieces. He can go out to collect more sticks, but how would he know that the sticks would work with the demonstration?

The coach came up with a clever idea: just glue the pieces back together to form larger sticks! It appears that every piece can be glued together securely with any other piece even if the two pieces came from different sticks originally. Thus, he can reconstruct sticks by putting two or more pieces together. The reconstructed sticks have the added advantage that each individual stick is really easy to snap at the connection point. However, if two sticks have a connection point at the same location, the two sticks will snap just as easily when bundled together. Therefore, he must reconstruct three sticks in such a way that none of the connection points coincide. Furthermore, it is desirable for the reconstructed sticks to be as long as possible. Finally, the three reconstructed sticks must have the same length−−−the coach does not want to imply that one team member is better than another. It is acceptable to leave some pieces unused. Each piece can only be used once, of course.

Input

The input consists of a number of cases. Each case is specified on one line. The first number specifies the number of pieces (N). Each of the following N numbers is a positive integer specifying the length of each piece. There are at most 13 pieces, and the length of each piece is at most 25. The end of input is specified by a case in which N = 0.

Output

For each case, print the case number followed by a colon, followed by the longest possible length of the three reconstructed sticks on a single line. If it is impossible to reconstruct three sticks satisfying the constraints stated, then 0 is the longest possible length of the reconstructed sticks.

Sample Input

```
10 4 2 3 7 8 9 1 2 3 4
10 1 2 3 4 5 6 7 8 9 10
8 2 3 4 1 1 3 2 2
10 25 25 25 25 25 25 25 25 25 25
0
```

Sample Output

```
Case 1: 14
Case 2: 18
Case 3: 6
Case 4: 0
```

Problem H: Wavelet Compression

Input: `wavelet.in`
Output: `wavelet.out`

The discrete wavelet transform is a popular tool for signal compression. In this problem, your job is to write a program to decompress a one–dimensional signal (a list of integers) that has been compressed by a simple wavelet transform.

To understand how this simple wavelet transform works, suppose that we have a list of an even number of integers. We compute the sum and difference of each pair of consecutive samples, resulting in two lists of sums and differences each having half the original length. Formally, if the original samples are

```
a(1),..., a(n)
```

the i–th sum `s(i)` and difference `d(i)` are computed as:

```
for i = 1,...,n/2:
  s(i) = a(2*i-1) + a(2*i)
  d(i) = a(2*i-1) - a(2*i)
```

This is then rearranged to give the transformed signal by first listing the sums and then the differences. For example, if the input signal is:

```
  5, 2, 3, 2, 5, 7, 9, 6
```

Then the sum and difference signals are:

```
   s(i) = 7, 5, 12, 15
   d(i) = 3, 1, -2, 3
```

Thus, the transformed signal is:

```
  7, 5, 12, 15, 3, 1, -2, 3
```

The same process is applied recursively to the first half of the transformed signal, treating `s(i)` as the input signal, until the length of the input signal is 1. In the example above, the final transformed signal is:

```
   39, -15, 2, -3, 3, 1, -2, 3
```

It is assumed that the length of the original input is a power of 2, and the input signal consists of integers between 0 and 255 (inclusive) only.

Input

The input consists of a number of cases. Each case is specified on a line, starting with an integer `N` (1 <= `N` <= 256) indicating the number of samples. The next `N` integers are the transformed samples. The end of input is indicated by a case in which `N` = 0.

Output

For each test case, output the original samples on a single line, separated by a single space.

Sample Input

```
8 39 -15 2 -3 3 1 -2 3
4 10 -4 -1 -1
0
```

Sample Output

```
5 2 3 2 5 7 9 6
1 2 3 4
```

Problem I: Elementary Additions

Input: `addition.in`
Output: `addition.out`

In today's environment, students rely on calculators and computers to perform simple arithmetic too much. Sadly, it is not uncommon to see university students who cannot do simple arithmetic without electronic aids. Professor Peano has had enough. He has decided to take the matter into his own hands and force his students to become proficient in the most basic arithmetic skill: addition of non−negative integers. Since the students do not have a good foundation in this skill, he decided to go back to the basics and represent non−negative integers with set theory.

The non−negative integers are represented by the following sets:

- 0 is represented by the empty set {}.
- For any number n > 0, n is represented by a set containing the representations of all non−negative integers smaller than n.

For example, the first 4 non−negative integers are represented by:

```
0 => {}
1 => {{}}
2 => {{},{{}}}
3 => {{},{{}},{{},{{}}}}
```

and so on. Notice that the cardinality (size) of the set is exactly the integer it represents. Although the elements of a set are generally unordered, Professor Peano requires that the elements of a set be ordered in increasing cardinality to make the assignments easier to grade. As an added advantage, Professor Peano is sure that there are no calculators or computer programs that can deal with numbers written in this notation.

Not surprisingly, many students cannot cope with this basic task and will fail the course if they do not get help soon. It is up to you, an enterprising computer science student, to help them. You have decided to write a computer program, codenamed Axiomatic Cheating Machine (ACM), to sell to the students and help them perform the additions to pass the course.

Input

The first line of the input contains a positive integer giving the number of cases to follow. For each case, there are two lines of input each containing a non−negative integer represented in set notation. Each line contains only the characters '{', '}', and ','. The sum of the two given integers will be at most 15.

Output

For each test case, output the sum of the two input integers in the set notation described above.

Sample Input

```
3
{}
{}
{{}}
{{},{{}}}
{{},{{}},{{},{{}}}}
{{}}
```

Sample Output

```
{}
{{},{{}},{{},{{}}}}
{{},{{}},{{},{{}}},{{},{{}},{{},{{}}}}}
```