ACM International Collegiate Programming Contest 1998/99

Sponsored by IBM

Supported by Wilken GmbH, Schwarz Pharma and the German ACM Chapter

Southwestern European Regional Contest

University of Ulm, Germany

November 1st, 1998

This problem set should contain nine (9) problems on twenty (20) numbered pages. Please inform a runner immediately if something is missing from your problem set.

ACM International Collegiate Programming Contest 1998/99
Southwestern European Regional Contest

Problem A

Optimal Programs

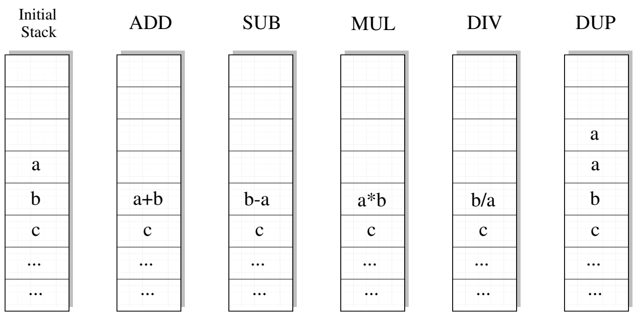
Source: `optimal.(c|cc|pas|java)`
Input: `optimal.in`

As you know, writing programs is often far from being easy. Things become even harder if your programs have to be as fast as possible. And sometimes there is reason for them to be. Many large programs such as operating systems or databases have "bottlenecks" – segments of code that get executed over and over again, and make up for a large portion of the total running time. Here it usually pays to rewrite that code portion in assembly language, since even small gains in running time will matter a lot if the code is executed billions of times.

In this problem we will consider the task of automating the generation of optimal assembly code. Given a function (as a series of input/output pairs), you are to come up with the shortest assembly program that computes this function.

The programs you produce will have to run on a stack based machine, that supports only five commands: `ADD`, `SUB`, `MUL`, `DIV` and `DUP`. The first four commands pop the two top elements from the stack and push their sum, difference, product or integer quotient[1], respectively, on the stack. The `DUP` command pushes an additional copy of the top-most stack element on the stack.

So if the commands are applied to a stack with the two top elements a and b (shown to the left), the resulting stacks look as follows:



At the beginning of the execution of a program, the stack will contain a single integer only: the input. At the end of the computation, the stack must also contain only one integer; this number is the result of the computation.

There are three cases in which the stack machine enters an error state:

- A `DIV`-command is executed, and the top-most element of the stack is 0.
- A `ADD`, `SUB`, `MUL` or `DIV`-command is executed when the stack contains only one element.
- An operation produces a value greater than 30000 in absolute value.

[1] This corresponds to `/` applied to two integers in C/C++, and `DIV` in Pascal.

Input

The input consists of a series of function descriptions. Each description starts with a line containing a single integer n ($n \leq 10$), the number of input/output pairs to follow. The following two lines contains n integers each: $x_1, x_2, \ldots, x_n$ in the first line (all different), and $y_1, y_2, \ldots, y_n$ in the second line. The numbers will be no more than 30000 in absolute value.

The input is terminated by a test case starting with $n = 0$. This test case should not be processed.

Output

You are to find the shortest program that computes a function f, such that $f(x_i) = y_i$ for all $i \in \{1, \ldots, n\}$. This implies that the program you output may not enter an error state if executed on the inputs x_i (although it may enter an error state for other inputs). Consider only programs that have at most 10 statements.

For each function description, output first the number of the description. Then print out the sequence of commands that make up the shortest program to compute the given function. If there is more than one such program, print the lexicographically smallest. If there is no program of at most 10 statements that computes the function, print the string "`Impossible`". If the shortest program consists of zero commands, print "`Empty Sequence`".

Output a blank line after each test case.

Sample Input

```
4
1 2 3 4
0 -2 -6 -12
3
1 2 3
1 11 1998
1
1998
1998
0
```

Sample Output

```
Program 1
DUP DUP MUL SUB

Program 2
Impossible

Program 3
Empty sequence
```

Problem B

The die is cast

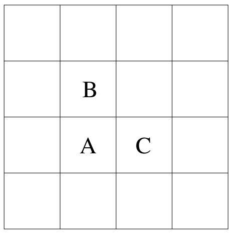
Source: `dice.(c|cc|pas|java)`
Input: `dice.in`

InterGames is a high-tech startup company that specializes in developing technology that allows users to play games over the Internet. A market analysis has alerted them to the fact that games of chance are pretty popular among their potential customers. Be it Monopoly, ludo or backgammon, most of these games involve throwing dice at some stage of the game.

Of course, it would be unreasonable if players were allowed to throw their dice and then enter the result into the computer, since cheating would be way to easy. So, instead, InterGames has decided to supply their users with a camera that takes a picture of the thrown dice, analyzes the picture and then transmits the outcome of the throw automatically.

For this they desperately need a program that, given an image containing several dice, determines the numbers of dots on the dice.

We make the following assumptions about the input images. The images contain only three different pixel values: for the background, the dice and the dots on the dice. We consider two pixels *connected* if they share an edge – meeting at a corner is not enough. In the figure, pixels A and B are connected, but B and C are not.



A set S of pixels is connected if for every pair (a,b) of pixels in S, there is a sequence $a_1, a_2, \dots, a_k$ in S such that $a = a_1$ and $b = a_k$, and a_i and a_{i+1} are connected for $1 \leq i < k$.

We consider all maximally connected sets consisting solely of non-background pixels to be dice. \`Maximally connected' means that you cannot add any other non-background pixels to the set without making it dis-connected. Likewise we consider every maximal connected set of dot pixels to form a dot.

Input

The input consists of pictures of several dice throws. Each picture description starts with a line containing two numbers w and h, the width and height of the picture, respectively. These values satisfy $5 \leq w, h \leq 50$.

The following h lines contain w characters each. The characters can be: "`.`" for a background pixel, "`*`" for a pixel of a die, and "`X`" for a pixel of a die's dot.

Dice may have different sizes and not be entirely square due to optical distortion. The picture will contain at least one die, and the numbers of dots per die is between 1 and 6, inclusive.

The input is terminated by a picture starting with $w = h = 0$, which should not be processed.

Output

For each throw of dice, first output its number. Then output the number of dots on the dice in the picture, sorted in increasing order.

Print a blank line after each test case.

Sample Input

```
30 15
..............................
..............................
...............*..............
...*****......****............
...*X***.....**X***...........
...*****....***X**............
...***X*.....****.............
...*****.......*..............
..............................
........***........******.....
.......**X****.....*X**X*.....
......*******......******.....
.....****X**.......*X**X*.....
........***........******.....
..............................
0 0
```

Sample Output

```
Throw 1
1 2 2 4
```

Problem C

It's not a Bug, it's a Feature!

Source: `bugs.(c|cc|pas|java)`
Input: `bugs.in`

It is a curious fact that consumers buying a new software product generally do *not* expect the software to be bug-free. Can you imagine buying a car whose steering wheel only turns to the right? Or a CD-player that plays only CDs with country music on them? Probably not. But for software systems it seems to be acceptable if they do not perform as they should do. In fact, many software companies have adopted the habit of sending out patches to fix bugs every few weeks after a new product is released (and even charging money for the patches).

Tinyware Inc. is one of those companies. After releasing a new word processing software this summer, they have been producing patches ever since. Only this weekend they have realized a big problem with the patches they released. While all patches fix some bugs, they often rely on other bugs to be present to be installed. This happens because to fix one bug, the patches exploit the special behavior of the program due to another bug.

More formally, the situation looks like this. Tinyware has found a total of n bugs $B = \{b_1, b_2, \ldots, b_n\}$ in their software. And they have released m patches p_1, p_2, ..., p_m. To apply patch p_i to the software, the bugs $B_i^+ \subseteq B$ have to be present in the software, and the bugs $B_i^- \subseteq B$ must be absent (of course $B_i^+ \cap B_i^- = \emptyset$ holds). The patch then fixes the bugs $F_i^- \subseteq B$ (if they have been present) and introduces the new bugs $F_i^+ \subseteq B$ (where, again, $F_i^- \cap F_i^+ = \emptyset$).

Tinyware's problem is a simple one. Given the original version of their software, which contains all the bugs in B, it is possible to apply a sequence of patches to the software which results in a bug-free version of the software? And if so, assuming that every patch takes a certain time to apply, how long does the fastest sequence take?

Input

The input contains several product descriptions. Each description starts with a line containing two integers n and m, the number of bugs and patches, respectively. These values satisfy $1 \leq n \leq 20$ and $1 \leq m \leq 100$. This is followed by m lines describing the m patches in order. Each line contains an integer, the time in seconds it takes to apply the patch, and two strings of n characters each.

The first of these strings describes the bugs that have to be present or absent before the patch can be applied. The i-th position of that string is a "+" if bug b_i has to be present, a "−" if bug b_i has to be absent, and a "`0`" if it doesn't matter whether the bug is present or not.

The second string describes which bugs are fixed and introduced by the patch. The i-th position of that string is a "+" if bug b_i is introduced by the patch, a "−" if bug b_i is removed by the patch (if it was present), and a "`0`" if bug b_i is not affected by the patch (if it was present before, it still is, if it wasn't, is still isn't).

The input is terminated by a description starting with $n = m = 0$. This test case should not be processed.

Output

For each product description first output the number of the product. Then output whether there is a sequence of patches that removes all bugs from a product that has all n bugs. Note that in such a sequence a patch may be used multiple times. If there is such a sequence, output the time taken by the fastest sequence in the format shown in the sample output. If there is no such sequence, output "`Bugs cannot be fixed.`".

Print a blank line after each test case.

Sample Input

```
3 3
1 000 00-
1 00- 0-+
2 0-- -++
4 1
7 0-0+ ----
0 0
```

Sample Output

```
Product 1
Fastest sequence takes 8 seconds.

Product 2
Bugs cannot be fixed.
```

Problem D

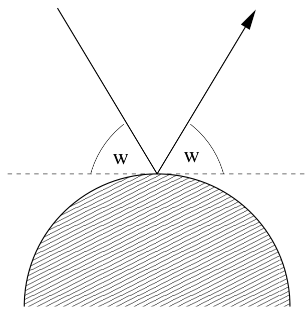
Reflections

Source: `reflect.(c|cc|pas|java)`
Input: `reflect.in`

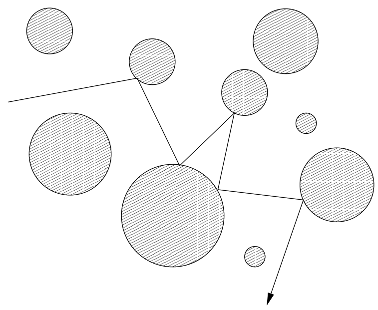
Rendering realistic images of imaginary environments or objects is an interesting topic in computer graphics. One of the most popular methods for this purpose is *ray-tracing*.

To render images using ray-tracing, one computes (traces) the path that rays of light entering a scene will take. We ask you to write a program that computes such paths in a restricted environment.

For simplicity, we will consider only two-dimensional scenes. All objects in the scene are totally reflective (mirror) spheres. When a ray of light hits such a sphere, it is reflected such that the angle of the incoming ray and the leaving ray against the tangent are the same:



The following figure shows a typical path that a ray of light may take in such a scene:



Your task is to write a program, that given a scene description and a ray entering the scene, determines which spheres are hit by the ray.

Input

The input consists of a series of scene descriptions. Each description starts with a line containing the number n ($n \leq 25$) of spheres in the scene. The following n lines contain three integers x_i, y_i, r_i each, where (x_i, y_i) is the center, and $r_i > 0$ is the radius of the i-th sphere. Following this is a line containing four integers x, y, d_x, d_y, which describe the ray. The ray originates from the point (x, y) and initially points in the direction (d_x, d_y). At least one of d_x and d_y will be non-zero.

The spheres will be disjoint and non-touching. The ray will not start within a sphere, and never touch a sphere tangentially.

A test case starting with $n = 0$ terminates the input. This case should not be processed.

Output

For each scene first output the number of the scene. Then print the numbers of the spheres that the ray hits in its first ten deflections (the numbering of spheres is according to their order in the input).

If the ray hits at most ten spheres (and then heads towards infinity), print `inf` after the last sphere it hits. If the ray hits more than 10 spheres, print three points (`...`) after the tenth sphere.

Output a blank line after each test case.

Sample Input

```
3
3 3 2
7 7 1
8 1 1
3 8 1 -4
2
0 0 1
5 0 2
2 0 1 0
0
```

Sample Output

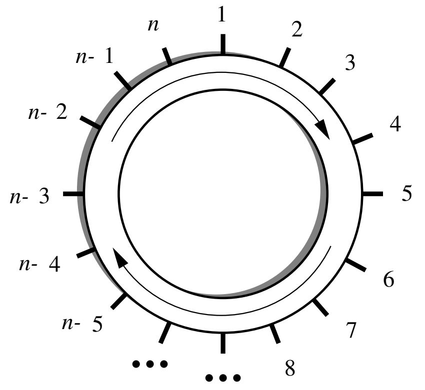
```
Scene 1
1 2 1 3 inf

Scene 2
2 1 2 1 2 1 2 1 2 1 ...
```

Problem E

Going in circles on Alpha Centauri

Source: `centauri.(c|cc|pas|java)`
Input: `centauri.in`

In the early 27th century, Alpha Centauri has become the main shipping hub of this part of the galaxy. At a space station near the fourth planet, goods from almost every space-faring civilization are traded and shipped to all major star systems. The space station is shaped like a large circle, and has docking ports on its outer rim, labelled clockwise from 1 to n:



When a trading spaceship docks to a port, it usually makes a request to transfer its cargo to another ship docked to some other port. This task is taken care of by transportation robots (transrobs) operating within the ring of the space station. The transrobs can travel clockwise around the station, and load and unload cargo at the ports.

Every ships cargo fits into one transport container, and all transrobs can carry only one container at a time. The transrobs only differ in maximal weight they can carry.

The consortium operating the space station has recently decided to upgrade its transportation system. But before doing so, they want to gather some statistics on the performance of their current system. More specifically, they are interested in

- the average time it takes for a request to be fulfilled, i.e. the time between a ship requesting a cargo to be taken to another port, and the cargo actually being delivered to its destination, and
- the utilization of the transrobs, i.e. the average percentage of transrobs serving requests during some interval of time

For this, they need a simulation program, which you have to write. To facilitate this task, the consortium has released the following details on their transrob control program.

- The transrobs are numbered 1 to m.
- It takes a transrob 1 minute to get from a port to the next one, and it takes 5 minutes to load or unload a container at a port.

- Transrobs move on different tracks, and therefore do not hinder each other when performing their duties.
- Transrobs are either *idle*, or they are *servicing a request*, which means that they move to the origin of that request, load the cargo, move to the destination, unload the cargo, and become idle again.
- All incoming requests are put in the *request list*. A request from that list is *possible* to satisfy if there is an idle transrob for which the cargo is not too heavy.
- As long as (or as soon as) there are possible requests on the list, they are assigned to transrobs, giving precedence for older requests over newer requests. Each request is assigned to the transrob which is closest (in anti-clockwise direction) to the origin of the request, and for which the cargo is not too heavy. If there are two transrobs at the same distance, the one with the lower number gets assigned the request. Assigned requests are deleted from the request list.
- The assignment procedure is instantenous, i.e. a robot starts moving in the instant it gets assigned a request, and a robot becomes idle (and can get a new request) in the instant it finishes unloading.

Input

The input consists of the description of several simulations you have to perform. Each description starts with a line containing two integers, n and m, the number of ports and transrobs, respectively, satisfying $2 \le n \le 100$ and $1 \le m \le 20$. The next m lines contain a single integer l_i each, the maximum load that transrob i can carry, measured in galactic tons.

This is followed by one or more shipments to perform. Each shipment is described by a line containing four integers, t, o, d, w: the time t the request was made at (measured in minutes since the beginning of the simulation), the port number o where the shipment comes from (origin), the port number d of the shipment's destination, and the weight w of the container in galactic tons. The request times are in strictly increasing order in the input file. The values satisfy $1 \le t$, $1 \le o, d \le n$, $o \ne d$ and $1 \le w \le \max\{ l_i \mid 1 \le i \le m \}$.

The description of shipments is terminated by the line "`-1 -1 -1 -1`".

The input is terminated by a test case starting with $n = m = 0$. This test case should not be processed.

Output

For each simulation description in the input, first output the number of the description. Then, simulate the operation of the transrobs on the shipment requests and output the average wait time, and the utilization percentage. The utilization percentage is computed for the interval of the time between the first request was made until the moment all requests were satisfied.

At the beginning of the simulation (time 0), all transrobs are idle, and located at port number 1.

All values must be exact to three digits to the right of the decimal point.

Output a blank line after each test case.

Input Sample

```
10 3
5
10
```

```
20
1 2 9 8
2 7 8 5
5 3 2 17
20 1 2 4
-1 -1 -1 -1
0 0
```

Output Sample

```
Simulation 1
Average wait time   = 17.250 minutes
Average utilization = 71.875 %
```

Problem F

Blowing Fuses

Source: `fuses.(c|cc|pas|java)`
Input: `fuses.in`

Maybe you are familiar with the following situation. You have plugged in a lot of electrical devices, such as toasters, refrigerators, microwave ovens, computers, stereos, etc, and have them all running. But at the moment when you turn on the TV, the fuse blows, since the power drawn from all the machines is greater than the capacity of the fuse. Of course this is a great safety feature, avoiding that houses burn down too often due to fires ignited by overheating wires. But it is also annoying to walk down to the basement (or some other inconvenient place) to replace to fuse or switch it back on.

What one would like to have is a program that checks *before* turning on an electrical device whether the combined power drawn by all running devices exceeds the fuses capacity (and it blows), or whether it is safe to turn it on.

Input

The input consists of several test cases. Each test case describes a set of electrical devices and gives a sequence of turn on/off operations for these devices.

The first line of each test case contains three integers n, m and c, where n is the number of devices ($n \leq 20$), m the number of operations performed on these devices and c is the capacity of the fuse (in Amperes). The following n lines contain one positive integer c_i each, the consumption (in Amperes) of the i-th device.

This is followed by m lines also containing one integer each, between 1 and n inclusive. They describe a sequence of turn on/turn off operations performed on the devices. For every number, the state of that particular devices is toggled, i.e. if it is currently running, it is turned off, and if it is currently turned off, it will by switched on. At the beginning all devices are turned off.

The input will be terminated by a test case starting with $n = m = c = 0$. This test case should not be processed.

Output

For each test case, first output the number of the test case. Then output whether the fuse was blown during the operation sequence. The fuse will be blown if the sum of the power consumptions c_i of turned on devices at some point exceeds the capacity of the fuse c.

If the fuse is not blown, output the maximal power consumption by turned on devices that occurred during the sequence.

Output a blank line after each test case.

Sample Input

```
2 2 10
5
7
1
2
3 6 10
2
5
7
2
1
2
3
1
3
0 0 0
```

Sample Output

```
Sequence 1
Fuse was blown.

Sequence 2
Fuse was not blown.
Maximal power consumption was 9 amperes.
```

Problem G

Fast Food

Source: `fastfood.(c|cc|pas|java)`
Input: `fastfood.in`

The fastfood chain McBurger owns several restaurants along a highway. Recently, they have decided to build several depots along the highway, each one located at a restaurent and supplying several of the restaurants with the needed ingredients. Naturally, these depots should be placed so that the average distance between a restaurant and its assigned depot is minimized. You are to write a program that computes the optimal positions and assignments of the depots.

To make this more precise, the management of McBurger has issued the following specification: You will be given the positions of n restaurants along the highway as n integers $d_1 < d_2 < ... < d_n$ (these are the distances measured from the company's headquarter, which happens to be at the same highway). Furthermore, a number k ($k \leq n$) will be given, the number of depots to be built.

The k depots will be built at the locations of k different restaurants. Each restaurant will be assigned to the closest depot, from which it will then receive its supplies. To minimize shipping costs, the *total distance sum*, defined as

$$\sum_{i=1}^{n} \mid d_i - (\text{position of depot serving restaurant } i) \mid$$

must be as small as possible.

Write a program that computes the positions of the k depots, such that the total distance sum is minimized.

Input

The input file contains several descriptions of fastfood chains. Each description starts with a line containing the two integers n and k. n and k will satisfy $1 \leq n \leq 200, 1 \leq k \leq 30, k \leq n$. Following this will n lines containing one integer each, giving the positions d_i of the restaurants, ordered increasingly.

The input file will end with a case starting with $n = k = 0$. This case should not be processed.

Output

For each chain, first output the number of the chain. Then output an optimal placement of the depots as follows: for each depot output a line containing its position and the range of restaurants it serves. If there is more than one optimal solution, output any of them. After the depot descriptions output a line containing the total distance sum, as defined in the problem text.

Output a blank line after each test case.

Sample Input

```
6 3
5
6
12
19
20
27
0 0
```

Sample Output

```
Chain 1
Depot 1 at restaurant 2 serves restaurants 1 to 3
Depot 2 at restaurant 4 serves restaurants 4 to 5
Depot 3 at restaurant 6 serves restaurant 6
Total distance sum = 8
```

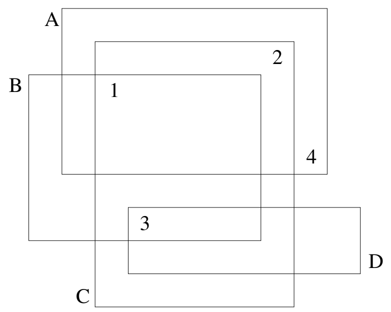
Problem H

Sorting Slides

Source: `slides.(c|cc|pas|java)`
Input: `slides.in`

Professor Clumsey is going to give an important talk this afternoon. Unfortunately, he is not a very tidy person and has put all his transparencies on one big heap. Before giving the talk, he has to sort the slides. Being a kind of minimalist, he wants to do this with the minimum amount of work possible.

The situation is like this. The slides all have numbers written on them according to their order in the talk. Since the slides lie on each other and are transparent, one cannot see on which slide each number is written.



Well, one cannot *see* on which slide a number is written, but one may *deduce* which numbers are written on which slides. If we label the slides which characters A, B, C, ... as in the figure above, it is obvious that D has number 3, B has number 1, C number 2 and A number 4.

Your task, should you choose to accept it, is to write a program that automates this process.

Input

The input consists of several heap descriptions. Each heap descriptions starts with a line containing a single integer n, the number of slides in the heap. The following n lines contain four integers x_{min}, x_{max}, y_{min} and y_{max}, each, the bounding coordinates of the slides. The slides will be labeled as A,B,C,... in the order of the input.

This is followed by n lines containing two integers each, the x- and y-coordinates of the n numbers printed on the slides. The first coordinate pair will be for number 1, the next pair for 2, etc. No number will lie on a slide boundary.

The input is terminated by a heap description starting with $n = 0$, which should not be processed.

Output

For each heap description in the input first output its number. Then print a series of all the slides whose numbers can be uniquely determined from the input. Order the pairs by their letter identifier.

If no matchings can be determined from the input, just print the word `none` on a line by itself.

Output a blank line after each test case.

Sample Input

```
4
6 22 10 20
4 18 6 16
8 20 2 18
10 24 4 8
9 15
19 17
11 7
21 11
2
0 2 0 2
0 2 0 2
1 1
1 1
0
```

Sample Output

```
Heap 1
(A,4) (B,1) (C,2) (D,3)

Heap 2
none
```

Problem I

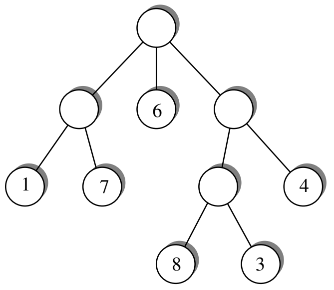
Single-Player Games

Source: `games.(c|cc|pas|java)`
Input: `games.in`

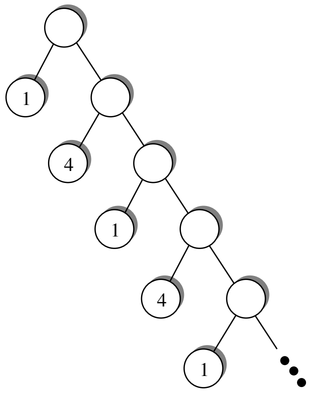
Playing games is the most fun if other people take part. But other players are not always available if you need them, which led to the invention of single-player games. One of the most well-known examples is the infamous "Solitaire" packaged with Windows, probably responsible for more wasted hours in offices around the world than any other game.

The goal of a single-player game is usually to make "moves" until one reaches a final state of the game, which results in a win or loss, or a score assigned to that final state. Most players try to optimize the result of the game by employing good strategies. In this problem we are interested in what happens if one plays randomly. After all, these games are mostly used to waste time, and playing randomly achieves this goal as well as any other strategy.

Games can very compactly represented as (possibly infinite) trees. Every node of the tree represents a possible game state. The root of the tree corresponds to the starting position of the game. For an inner node, its children are the game states to which one can move in a single move. The leaf nodes are the final states, and every one of them is assigned a number, which is the score one receives when ending up at that leaf.



`a = ((1 7) 6 ((8 3) 4))`




```
a = (1 b)
b = (4 a)
```

Trees are defined using the following grammar.

$$
\begin{aligned}
Definition &::= Identifier \text{ “=” } RealTree \\
RealTree &::= \text{ “(” } Tree^{+} \text{ “)” } \\
Tree &::= Identifier \mid Integer \mid \text{ “(” } Tree^{+} \text{ “)” } \\
Identifier &::= \texttt{a} \mid \texttt{b} \mid \ldots \mid \texttt{z} \\
Integer &\in \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots,\}
\end{aligned}
$$

By using a *Definition*, the *RealTree* on the right-hand side of the equation is assigned to the *Identifier* on the left. A *RealTree* consists of a root node and one or more children, given as a sequence enclosed in brackets. And a *Tree* is either

- the tree represented by a given *Identifier*, or
- a leaf node, represented by a single *Integer*, or
- an inner node, represented by a sequence of one or more *Tree*s (its children), enclosed in brackets.

Your goal is to compute the expected score, if one plays randomly, i.e. at each inner node selects one of the children uniformly at random. This expected score is well-defined even for the infinite trees definable in our framework as long as the probability that the game ends (playing randomly) is 1.

Input

The input file contains several gametree descriptions. Each description starts with a line containing the number n of identifiers used in the description. The identifiers used will be the first n lowercase letters of the alphabet. The following n lines contain the definitions of these identifiers (in the order `a`, `b`, ...). Each definition may contain arbitrary whitespace (but of course there will be no spaces within a single integer). The right hand side of a definition will contain only identifiers from the first n lowercase letters.

The inputs ends with a test case starting with $n = 0$. This test case should not be processed.

Output

For each gametree description in the input, first output the number of the game. Then, for all n identifiers in the order `a`, `b`, ..., output the following. If an identifier represents a gametree for which the probability of finishing the game is 1, print the expected score (when playing randomly). This value should be exact to three digits to the right of the decimal point.

If the game described by the variable does not end with probability 1, print "`Expected score of` *id* `undefined`" instead.

Output a blank line after each test case.

Sample Input

```
1
a = ((1 7) 6 ((8 3) 4))
2
a = (1 b)
b = (4 a)
1
a = (a a a)
0
```

Sample Output

```
Game 1
Expected score for a = 4.917
```

```
Game 2
Expected score for a = 2.000
Expected score for b = 3.000

Game 3
Expected score for a undefined
```