Problem A: Quicksum

Source file: `quicksum.{c, cpp, java}`
Input file: `quicksum.in`

A checksum is an algorithm that scans a packet of data and returns a single number. The idea is that if the packet is changed, the checksum will also change, so checksums are often used for detecting transmission errors, validating document contents, and in many other situations where it is necessary to detect undesirable changes in data.

For this problem, you will implement a checksum algorithm called Quicksum. A Quicksum packet allows only uppercase letters and spaces. It always begins and ends with an uppercase letter. Otherwise, spaces and letters can occur in any combination, including consecutive spaces.

A Quicksum is the sum of the products of each character's position in the packet times the character's value. A space has a value of zero, while letters have a value equal to their position in the alphabet. So, A=1, B=2, etc., through Z=26. Here are example Quicksum calculations for the packets "`ACM`" and "`MID CENTRAL`":

```
        ACM: 1*1  + 2*3 + 3*13 = 46

MID CENTRAL: 1*13 + 2*9 + 3*4 + 4*0 + 5*3 + 6*5 + 7*14 + 8*20 + 9*18 + 10*1 + 11*12 = 650
```

Input: The input consists of one or more packets followed by a line containing only # that signals the end of the input. Each packet is on a line by itself, does not begin or end with a space, and contains from 1 to 255 characters.

Output: For each packet, output its Quicksum on a separate line in the output.

Example Input:	**Example Output:**
ACM MID CENTRAL REGIONAL PROGRAMMING CONTEST ACN A C M ABC BBC #	46 650 4690 49 75 14 15

Last modified on October 29, 2006 at 3:55 PM.

Problem B: Linear Pachinko

Source file: `linear.{c, cpp, java}`
Input file: `linear.in`

This problem is inspired by Pachinko, a popular game in Japan. A traditional Pachinko machine is a cross between a vertical pinball machine and a slot machine. The player launches small steel balls to the top of the machine using a plunger as in pinball. A ball drops through a maze of pins that deflect the ball, and eventually the ball either exits at a hole in the bottom and is lost, or lands in one of many gates scattered throughout the machine which reward the player with more balls in varying amounts. Players who collect enough balls can trade them in for prizes.

For the purposes of this problem, a *linear* Pachinko machine is a sequence of one or more of the following: holes ("."), floor tiles ("_"), walls ("|"), and mountains ("/\"). A wall or mountain will never be adjacent to another wall or mountain. To play the game, a ball is dropped at random over some character within a machine. A ball dropped into a hole falls through. A ball dropped onto a floor tile stops immediately. A ball dropped onto the left side of a mountain rolls to the left across any number of consecutive floor tiles until it falls into a hole, falls off the left end of the machine, or stops by hitting a wall or mountain. A ball dropped onto the right side of a mountain behaves similarly. A ball dropped onto a wall behaves as if it were dropped onto the left or right side of a mountain, with a 50% chance for each. If a ball is dropped at random over the machine, with all starting positions being equally likely, what is the probability that the ball will fall either through a hole or off an end?

For example, consider the following machine, where the numbers just indicate character positions and are not part of the machine itself:

```
123456789
/\.|__/\.
```

The probabilities that a ball will fall through a hole or off the end of the machine are as follows, by position: 1=100%, 2=100%, 3=100%, 4=50%, 5=0%, 6=0%, 7=0%, 8=100%, 9=100%. The combined probability for the whole machine is just the average, which is approximately 61.111%.

Input: The input consists of one or more linear Pachinko machines, each 1–79 characters long and on a line by itself, followed by a line containing only "#" that signals the end of the input.

Output: For each machine, compute as accurately as possible the probability that a ball will fall through a hole or off the end when dropped at random, then output a single line containing that percentage *truncated* to an integer by dropping any fractional part.

Example input:	**Example output:**

`/\.|__/\.`
`_._/\_|.__/\./\_`
`...`
`___`
`./\.`
`_/\_`
`_|.|_|.|_|.|_`
`_____|_____`
`#`
	`61`
	`53`
	`100`
	`0`
	`100`
	`50`
	`53`
	`10`

Last modified on October 29, 2006 at 2:37 AM.

Problem C: Surprising Strings

Source file: `surprise.{c, cpp, java}`
Input file: `surprise.in`

The *D-pairs* of a string of letters are the ordered pairs of letters that are distance D from each other. A string is *D-unique* if all of its D-pairs are different. A string is *surprising* if it is D-unique for every possible distance D.

Consider the string ZGBG. Its 0-pairs are ZG, GB, and BG. Since these three pairs are all different, ZGBG is 0-unique. Similarly, the 1-pairs of ZGBG are ZB and GG, and since these two pairs are different, ZGBG is 1-unique. Finally, the only 2-pair of ZGBG is ZG, so ZGBG is 2-unique. Thus ZGBG is surprising. (Note that the fact that ZG is both a 0-pair and a 2-pair of ZGBG is irrelevant, because 0 and 2 are different distances.)

Input: The input consists of one or more nonempty strings of at most 79 uppercase letters, each string on a line by itself, followed by a line containing only an asterisk that signals the end of the input.

Output: For each string of letters, output whether or not it is surprising using the *exact* output format shown below.

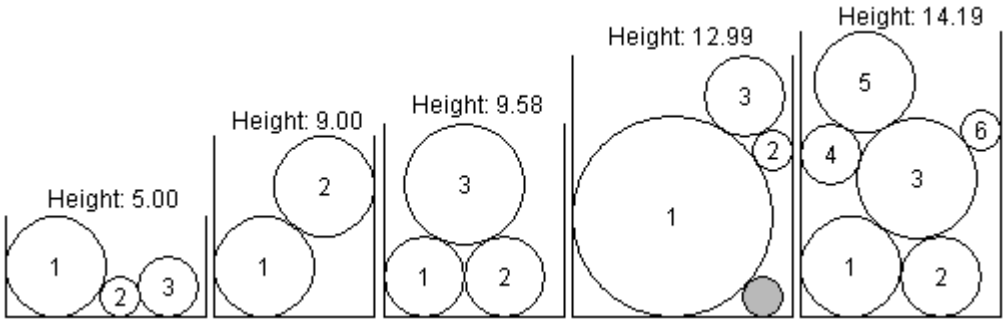
Acknowledgement: This problem is inspired by the "Puzzling Adventures" column in the December 2003 issue of *Scientific American*.

Example input:	**Example output:**
ZGBG X EE AAB AABA AABB BCBABCC *	ZGBG is surprising. X is surprising. EE is surprising. AAB is surprising. AABA is surprising. AABB is NOT surprising. BCBABCC is NOT surprising.

Last modified on October 28, 2006 at 8:49 PM.

Problem D: Falling Ice

Source file: `falling.{c, cpp, java}`
Input file: `falling.in`



Imagine disks of ice falling, one at a time, into a box, each ending up at the lowest point it can reach without overlapping or moving previous disks. Each disk then freezes into place, so it cannot be moved by later disks. Your job is to find the overall height of the final combination of disks.

So that the answer is unique, assume that any disk reaching the bottom of the box rolls as far to the left as possible. Also the data is chosen so there will be a unique lowest position for any disk that does not reach the bottom. The data is also such that there are no "perfect fits": each disk that lands will be in contact with only two other points, on previous circles or the sides of the box. The illustrations above show white filled disks labeled with the order in which they fall into their boxes. The gray circle in the fourth illustration is not intended to be a disk that fell in. The gray disk is included to demonstrate a point: the gray disk is the same size as disk 2, so there is *space* for disk 2 on the very bottom of its box, but disk 2 cannot *reach* that position by falling from the top. It gets caught on disk 1 and the side of the box.

One way to find the top intersection point of two intersecting circles is as follows. Suppose circle 1 has center (x1, y1) and radius r1, and suppose circle 2 has center (x2, y2), and radius r2. Also assume that circle 1 is to the left of circle 2, i.e., x1 < x2. Let

dx = x2 - x1,
dy = y2 - y1,
D = sqrt(dx*dx + dy*dy),
E = (r1*r1 - r2*r2 + D*D)/(2*D),
F = sqrt(r1*r1 - E*E);

then the upper intersection point is (x1 + (E*dx - F*dy)/D, y1 + (F*dx + E*dy)/D).

Input: The input consists of one or more data sets, followed by a line containing only 0 that signals the end of the input. Each data set is on a line by itself and contains a sequence of three or more blank-separated positive integers, in the format w, n, d_1, d_2, d_3, ..., d_n, where w is the width of the box, n is the number of disks, and the remaining numbers are the diameters of the disks, in the order in which they fall into the box. You can assume that $w < 100$, that $n < 10$, and that each diameter is less than w.

Output: For each data set, output a single line containing the height of the pile of disks, rounded to two places beyond the decimal point.

The example data matches the illustrations above.

Example input:	**Example output:**
10 3 5 2 3 8 2 5 5 11 3 10 2 4 9 3 4 4 6 10 6 5 4 6 3 5 2 0	5.00 9.00 12.99 9.58 14.19

Last modified on October 29, 2006 at 2:21 AM.

Problem E: Frugal Search

Source file: `frugal.{c, cpp, java}`
Input file: `frugal.in`

For this problem you will write a search engine that takes a query, searches a collection of words, and finds the lexicographically smallest word that matches the query (*i.e.*, the matching word that would appear first in an English dictionary). A *query* is a sequence of one or more terms separated by single vertical bars ("|"). A *term* is one or more letters followed by zero or more signed letters. A *signed* letter is either *+s* ("positive" *s*) or *−s* ("negative" *s*), where *s* is a single letter. All letters are lowercase, and no letter will appear more than once within a term. A query will not contain spaces. A term matches a word if the word contains at least one of the unsigned letters, all of the positive letters, and none of the negative letters; a query matches a word if at least one of its terms matches the word.

Input: The input consists of one or more test cases followed by a line containing only "#" that signals the end of the input. Each test case consists of 1–100 words, each on a line by itself, followed by a line containing only "*" that marks the end of the word list, followed by one or more queries, each on a line by itself, followed by a line containing only "**" that marks the end of the test case. Each word will consist of 1–20 lowercase letters. All words within a test case will be unique. Each query will be as defined above and will be 1–79 characters long.

Output: For each query, output a single line containing the lexicographically smallest word *within that test case* that matches the query, or the word `NONE` if there is no matching word. At the end of each test case, output a dollar sign on a line by itself.

Example input:	**Example output:**
elk cow bat * ea acm+e nm+o|jk+l ** debian slackware gentoo ubuntu suse fedora mepis * yts cab-e+n r-e|zjq|i+t|vs-p+e-u-c ** #	bat NONE elk $ gentoo ubuntu NONE $

Last modified on October28 19, 2006 at 10:49 PM.

Problem F: Go Go Gorelians

Source file: `gorelian.{c, cpp, java}`
Input file: `gorelian.in`

The Gorelians travel through space using warp links. Travel through a warp link is instantaneous, but for safety reasons, an individual can only warp once every 10 hours. Also, the cost of creating a warp link increases directly with the linear distance between the link endpoints.

The Gorelians, being the dominant force in the known universe, are often bored, so they frequently conquer new regions of space in the following manner.

1) The initial invasion force finds a suitable planet and conquers it, establishing a Regional Gorelian Galactic Government, hereafter referred to as the RGGG, that will govern all Gorelian matters in this region of space.

2) When the next planet is conquered, a single warp link is constructed between the new planet and the RGGG planet. Planets connected via warp links in this manner are said to be part of the Regional Gorelian Planetary Network, that is, the RGPN.

3) As additional planets are conquered, each new planet is connected with a single warp link to the nearest planet already in the RGPN, thus keeping the cost of connecting new planets to the network to a minimum. If two or more planets are equidistant from the new planet, the new planet is connected to whichever of them was conquered first.

This causes a problem however. Since planets are conquered in a more-or-less random fashion, after a while, the RGGG will probably not be in an ideal location. Some Gorelians needing to consult with the RGGG might only have to make one or two warps, but others might require dozens---very inconvenient when one considers the 10-hour waiting period between warps.

So, once each Gorelian year, the RGGG analyzes the RGPN and relocates itself to an optimal location. The optimal location is defined as a planet that minimizes the maximum number of warps required to reach the RGGG from any planet in the RGPN. As it turns out, there is always exactly one or two such planets. When there are two, they are always directly adjacent via a warp link, and the RGGG divides itself evenly between the two planets.

Your job is to write a program that finds the optimal planets for the RGGG. For the purposes of this problem, the region of space conquered by the Gorelians is defined as a cube that ranges from (0,0,0) to (1000,1000,1000).

Input: The input consists of a set of scenarios where the Gorelians conquer a region of space. Each scenario is independent. The first line of the scenario is an integer N that specifies the total number of planets conquered by the Gorelians. The next N lines of the input specify, in the order conquered, the IDs and coordinates of the conquered planets to be added to the RGPN, in the format ID X Y Z. An ID is an integer from 1 to 1000. X, Y, and Z are integers from 0 to 1000. A single space separates the numbers. A value of N = 0 marks the end of the input.

Output: For each input scenario, output the IDs of the optimal planet or planets where the RGGG should relocate. For a single planet, simply output the planet ID. For two planets, output the planet IDs, smallest ID first, separated by a single space.

Example Input:	Example Output:
5 1 0 0 0 2 0 0 1 3 0 0 2 4 0 0 3 5 0 0 4 5 1 0 0 0 2 1 1 0 3 3 2 0 4 2 1 0 5 3 0 0 10 21 71 76 4 97 32 5 69 70 33 19 35 3 79 81 8 31 91 17 67 52 31 48 75 48 90 14 4 41 73 2 21 83 74 41 69 26 32 30 24 0	3 2 4 31 97

Last modified on October 29, 2006 at 4:12 AM.

Problem G: Root of the Problem

Source file: `root.{c, cpp, java}`
Input file: `root.in`

Given positive integers B and N, find an integer A such that A^N is as close as possible to B. (The result A is an approximation to the Nth root of B.) Note that A^N may be less than, equal to, or greater than B.

Input: The input consists of one or more pairs of values for B and N. Each pair appears on a single line, delimited by a single space. A line specifying the value zero for both B and N marks the end of the input. The value of B will be in the range 1 to 1,000,000 (inclusive), and the value of N will be in the range 1 to 9 (inclusive).

Output: For each pair B and N in the input, output A as defined above on a line by itself.

Example Input:	Example Output:
4 3 5 3 27 3 750 5 1000 5 2000 5 3000 5 1000000 5 0 0	1 2 3 4 4 4 5 16

Last modified on October 29, 2006 at 11:04 AM.