Colombian Collegiate Programming League
CCPL 2013

Contest 10 -- September 28

Problems

This set contains 10 problems; pages 1 to 23.

(Borrowed from several sources online.)

Official site `http://programmingleague.org`

Official Twitter account @CCPL2003

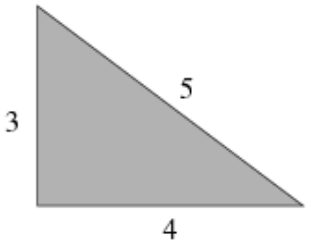
A - Pythagora's Revenge

Source file name: `pythagoras.c`, `pythagoras.cpp` *or* `pythagoras.java`

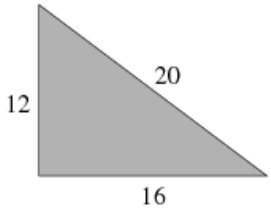
The famous Pythagorean theorem states that a right triangle, having side lengths A and B and hypotenuse length C, satisfies the formula

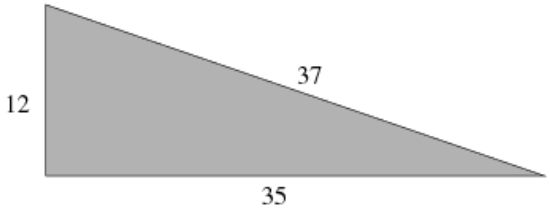
$$A^2 + B^2 = C^2$$

It is also well known that there exist some right triangles in which all three side lengths are integral, such as the classic:



Further examples, both having $A = 12$, are the following:





The question of the day is, given a fixed integer value for A, how many distinct integers $B > A$ exist such that the hypotenuse length C is an integer number?

Input

The input contains several test cases. Each test case comprises one line containing a single integer A, such that $2 \leq A < 1048576 = 2^{20}$. The end of the input is designated by a line containing the value $A = 0$.

The input must be read from standard input.

Output

For each value of A, output the number of integers $B > A$ such that a right triangle having side lengths A and B has a hypotenuse with length in the integer's domain.

The output must be written to standard output.

Sample Input	**Sample Output**
3 12 2 1048574 1048575 0	1 2 0 1 175

B - Digit Solitaire

Source file name: `digit.c`, `digit.cpp` *or* `digit.java`

Despite the glorious fall colors in the Midwest, there is a great deal of time to spend while on a train from St. Louis to Chicago. On a recent trip, we spent some time playing a game.

We start with a positive integer S. So long as it has more than one digit, we compute the product of its digits and repeat. For example, if starting with 95, we compute $9 \times 5 = 45$. Since 45 has more than one digit, we compute $4 \times 5 = 20$. Continuing with 20, we compute $2 \times 0 = 0$. Having reached 0, which is a single-digit number, the game is over.

As a second example, if we begin with 396, we get the following computations:

$$3 \times 9 \times 6 = 162$$
$$1 \times 6 \times 2 = 12$$
$$1 \times 2 = 2$$

and we stop the game having reached 2.

Input

The input comprises several test cases, each in a single line. Each line contains a single integer $1 \leq S \leq 100000$, designating the starting value. The value S will not have any leading zeros.

A value of $S = 0$ designates the end of the input.

The input must be read from standard input.

Output

For each nonzero input value, a single line of output should express the ordered sequence of values, single-space separated and without leading or trailing spaces, that are considered during the game, starting with the original value.

The output must be written to standard output.

Sample Input	**Sample Output**
95 396 28 4 40 0	95 45 20 0 396 162 12 2 28 16 6 4 40 0

C - Is the Name of This Problem

Source file name: `name.c`, `name.cpp` *or* `name.java`

The philosopher Willard Van Orman Quine (1908-2000) described a novel method of constructing a sentence in order to illustrate the contradictions that can arise from self-reference. This operation takes as input a single phrase and produces a sentence from that phrase. (The author Douglas R. Hofstadter refers to this process as to *Quine a phrase*.) We can define the Quine operation like so:

Quine(*A*) = ''*A*'' *A*

In other words, if *A* is a phrase, then *Quine*(*A*) is *A* enclosed in quotes (''), followed by a space, followed by *A*. For example:

Quine(`HELLO WORLD`) = ''*`HELLO WORLD`*'' `HELLO WORLD`

Below are some other examples of sentences that can be created by the Quine operation. Note that Quining allows sentences to be indirectly self-referential, such as the last sentence below.

```
"IS A SENTENCE FRAGMENT" IS A SENTENCE FRAGMENT
"IS THE NAME OF THIS PROBLEM" IS THE NAME OF THIS PROBLEM
"YIELDS FALSEHOOD WHEN QUINED" YIELDS FALSEHOOD WHEN QUINED
```

Your goal for this problem is to take a sentence and decide whether the sentence is the result of a Quine operation. In order to be a Quine sentence, a sentence must match the following pattern exactly:

1. A quotation mark
2. Any nonempty sequence of letters and spaces (call this phrase *A*)
3. A quotation mark
4. A space
5. Phrase *A* -exactly as it appeared in (2)

If a given sentence matches this pattern, such a sentence is a Quine of the phrase *A*. Note that phrase *A* must contain the exact same sequence of characters both times it appears.

Input

The input will consist of a sequence of sentences, one sentence per line, ending with a line that has the single word `END`. Each sentence will contain only uppercase letters, spaces, and quotation marks. Each sentence will contain between 1 and 80 characters and will not have any leading, trailing, or consecutive spaces.

The input must be read from standard input.

Output

There will be one line of output for each sentence in the data set. If the sentence is the result of a Quine operation, your output should be of the form, *Quine*(*A*), where *A* is the phrase to Quine to create the sentence.

If the sentence is not the result of a Quine operation, your output should be the phrase `not a quine`.

The output must be written to standard output.

Sample Input	**Sample Output**
"HELLO WORLD" HELLO WORLD "IS A SENTENCE FRAGMENT" IS A SENTENCE FRAGMENT "IS THE NAME OF THIS PROBLEM" IS THE NAME OF THIS PROBLEM "YIELDS FALSEHOOD WHEN QUINED" YIELDS FALSEHOOD WHEN QUINED "HELLO" I SAID WHAT ABOUT "WHAT ABOUT" " NO EXTRA SPACES " NO EXTRA SPACES "NO"QUOTES" NO"QUOTES "" END	Quine(HELLO WORLD) Quine(IS A SENTENCE FRAGMENT) Quine(IS THE NAME OF THIS PROBLEM) Quine(YIELDS FALSEHOOD WHEN QUINED) not a quine not a quine not a quine not a quine not a quine

D - The Mark of a Wizard

Source file name: `mark.c`, `mark.cpp` *or* `mark.java`

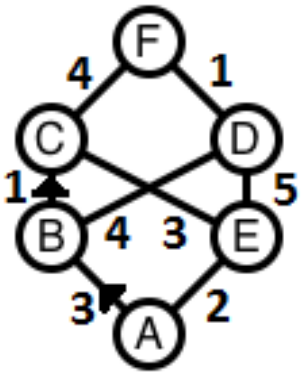


Figure 1

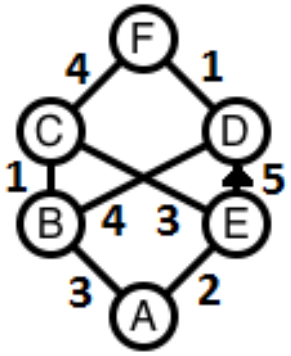


Figure 2

Goblins have a web of tunnels up from their underground lairs. A schematic for one of their simpler systems is shown in Figure 1. The vertical direction is oriented upward on the page. The lair is at label *A*. The surface is at label *F*. The other labels mark tunnel intersections. Tunnels heading toward the surface from a given intersection are oriented as such in our diagram. This is a 2D schematic of a 3D tunnel system, so the paths shown crossing elsewhere than at a lettered intersection do not actually intersect. (For example, in the above model, the edge *BD* does not intersect the edge *CE*.)

A group of good wizards finds the need to rush through such tunnels to the surface in many of their adventures, following an upward path that takes the minimum amount of time. Many upward tunnels can leave an intersection, and there is no obvious way to choose the best tunnel in general. The wizards took the trouble to map the tunnels systems, noting the time required to travel up from one intersection to the next, as labeled in the above figures. They do not have opportunity to consult elaborate notes when in a hurry. Nor do they want to help their enemies that might be rushing through the same tunnels. They decide to put up a very discreet private mark at intersections where needed. A mark on a tunnel leaving an intersection means that wizards should follow that tunnel.

They hope that they can make marks that only they will recognize, as there is a possibility that their system is discovered, particularly if they put up too many markers. It is important to mark as few intersections of tunnels as possible, while still guaranteeing that a wizard who always heads up at any intersection, and always take a marked tunnel where shown, will emerge in the minimum time. In Figure 1, possible direction markings are shown as arrow heads at *A* pointing toward *B*, and then at *B* pointing toward *C*. There is only one route up from *C*, so then any wizard following the markings to *C* would head for *F* and be out of the tunnels, with the total time of 3 + 1 + 4 = 8, the minimum possible.

Figure 2 shows the same schematic, except there is just one mark, directing any wizard at *E* to go to *D*. This does not totally determine a path, but it does ensure the minimum time: A wizard can go from *A* toward either *B* or *E*. If the wizard goes to *B*, there are two choices

again, but all these paths heading upward cover the same minimum distance, 8. The only marker at *E* cannot be removed or a wizard might try the route *AECF* for a total distance of 2 + 3 + 4 = 9. Hence to guarantee getting through the tunnels in minimum time (8), the minimum number of markers needed is 1. As an aside, there is another way to place just one marker for this example: at *A* pointing towards *B*.

You are to help the wizard plan the use of markers. Be careful of your algorithm. This could be time-consuming.

Input

The input will consist of one to 16 data sets, followed by a line containing only 0.

The first line of a data set contains an integer n, ($2 \leq n \leq 17$), which is the number of labeled places; those consist of the starting point, ending point, and any intersections in the tunnel system in between.

Each of the next n lines describes the upward tunnels from one labeled point. Each line has the same form, with blank separated parts: The first part is a letter labeling a tunnel starting point, then an integer u that is the number of upward tunnels from that point. After the label letter and u come u pairs, letter time, giving the integer time ($1 \leq time \leq 500$) to go through a tunnel from the current labeled point to the point with label letter. The labels at the beginning of the n lines will be taken in order from the beginning of the capital letters. *A* will always be the starting point and the last letter used will always be the exit. Only the final line (for the exit) will have $u = 0$. For previous lines, $1 \leq u \leq 6$.

Limiting assumptions:

- Wizards may only follow upward tunnels out of any intersection.
- The total number of tunnels is at most 35.
- There is always a path from the lair to the surface using the minimum time and involving at most 7 tunnels.
- All labeled points other than the starting point and the exit have at least one tunnel coming up to it, at least one heading up *from* it.

The first sample input corresponds to that given in the introductory figures.

The input must be read from standard input.

Output

There is one line of output for each data set, with two space-separated numbers: the minimum time to travel from the lair to the surface and the minimum number of markers needed to assure that all wizards travel that minimum time.

The output must be written to standard output.

Sample Input	**Sample Output**
6 A 2 B 3 E 2 B 2 C 1 D 4 C 1 F 4 D 1 F 1 E 2 C 3 D 5 F 0 7 A 3 B 1 C 5 D 4 B 2 C 2 E 5 C 2 E 4 F 3 D 2 C 2 F 3 E 1 G 6 F 1 G 4 G 0 7 A 2 B 2 C 4 B 2 D 4 C 1 C 2 D 3 E 5 D 2 F 4 E 2 E 2 F 2 G 5 F 1 G 2 G 0 0	8 1 10 3 12 2

E - LRU Caching

Source file name: `lru.c`, `lru.cpp` *or* `lru.java`

When accessing large amounts of data is deemed too slow, a common speed up technique is to keep a small amount of the data in a more accessible location known as a *cache*. The first time a particular piece of data is accessed, the slow method must be used. However, the data is then stored in the cache so that the next time you need it you can access it much more quickly. For example, a database system may keep data cached in memory so that it doesn't have to read the hard drive. Or a web browser might keep a cache of web pages on the local machine so that it doesn't have to download them over the network.

In general, a cache is much too small to hold all the data you might possibly need, so at some point you are going to have to remove something from the cache in order to make room for new data. The goal is to retain those items that are more likely to be retrieved again soon. This requires a sensible algorithm for selecting what to remove from the cache. One simple but effective algorithm is the Least Recently Used, or LRU, algorithm. When performing LRU caching, you always throw out the data that was least recently used.

As an example, let's imagine a cache that can hold up to five pieces of data. Suppose we access three pieces of data --*A*, *B*, and *C*. As we access each one, we store it in our cache, so at this point we have three pieces of data in our cache and two empty spots (Figure 1). Now suppose we access *D* and *E*. They are added to the cache as well, filling it up. Next suppose we access *A* again. *A* is already in the cache, so the cache does not change; however, this access counts as a use, making *A* the most recently used. Now if we were to access *F*, we would have to throw something out to make room for *F*. At this point, *B* has been used least recently, so we throw it out and replace it with *F* (Figure 2). If we were now to access *B* again, it would be exactly as the first time we accessed it: we would retrieve it and store it in the cache, throwing out the least recently used data --this time *C*- to make room for it.

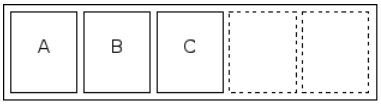


Figure 1: Cache after *A*, *B*, *C*

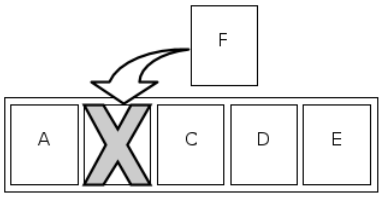


Figure 2: Cache after *A*, *B*, *C*, *D*, *E*, *A*, *F*

Your task for this problem is to take a sequence of data accesses and simulate an LRU cache. When requested, you will output the contents of the cache, ordered from least recently used to most recently used.

Input

The input will be a series of data sets, one per line. Each data set will consist of an integer *N* and a string of two or more characters. The integer *N* represents the size of the cache for the data set ($1 \leq N \leq 26$). The string of characters consists solely of uppercase letters and

exclamation marks. An upppercase letter represents an access to that particular piece of data. An exclamation mark represents a request to print the current contents of the cache.

For example, the sequence `ABC!DEAF!B!` means to acces *A*, *B*, and *C* (in that order), print the contents of the cache, access *D*, *E*, *A*, and *F* (in that order), then print the contents of the cache, then access *B*, and again print the contents of the cache.

The sequence will always begin with an uppercase letter and contain at least one exclamation mark.

The end of input will be signaled by a line containing only the number zero.

The input must be read from standard input.

Output

For each data set you should output the line ''`Simulation S`'', where *S* is 1 for the first data set, 2 for the second data set, etc. Then for each exclamation mark in the data set you should output the contents of the cache on one line as a sequence of characters representing the pieces of data currently in the cache. The characters should be sorted in order from least recently used to most recently used, with least recently occuring first. You only output the letters that are in the cache; if the cache is not full, then you simply will have fewer characters to output (that is, do not print any empty spaces). Note that because the sequence always begins with an uppercase letter, you will never be asked to output a completely empty cache.

The output must be written to standard output.

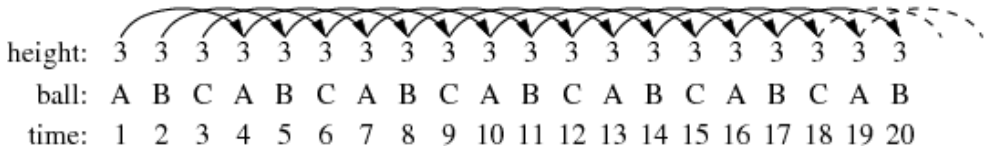
Sample Input	**Sample Output**

`5 ABC!DEAF!B!`
`3 WXWYZ!YZWYX!XYXY!`
`5 EIEIO!`
`0`
	`Simulation 1`
	`ABC`
	`CDEAF`
	`DEAFB`
	`Simulation 2`
	`WYZ`
	`WYX`
	`WXY`
	`Simulation 3`
	`EIO`

F - Jugglefest

Source file name: `jugglefest.c`, `jugglefest.cpp` *or* `jugglefest.java`

Many people are familiar with a standard 3-ball juggling pattern in which you throw ball *A*, then ball *B*, then ball *C*, then ball *A*, then ball *B*, then ball *C*, and so on. Assuming we keep a regular rhythm of throws, a ball that is thrown higher into the air will take longer to return, and therefore will take longer before the next time it gets thrown. We say that a ball thrown to height *h* will not be thrown again until precisely *h* steps later in the pattern. For example, in the standard 3-ball pattern, we say that each ball is thrown to a height of 3, and therefore thrown again 3-steps later in the pattern. For example, ball *A* that we throw at time 1 of the process will be next thrown at time 4.

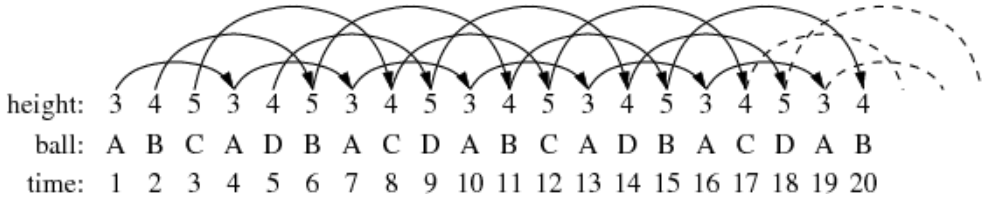


By convention, we label the first ball thrown as *A*, and each time we introduce a new ball into the pattern, we label it with the next consecutive uppercase letter (hence *B* and then *C* in the classic pattern).

There exist more complex juggling patterns. Within the community of jugglers, a standard way to describe a pattern is through a repeating sequence of numbers that describe the height of each successive throw. This is known as the **siteswap** notation.

To demonstrate the notation, we first consider the ''3 4 5'' siteswap pattern. This describes an infinite series of throws based on the repeating series ''3 4 5 3 4 5 3 4 5 . . .''. The first throw the juggler makes will be to a height of 3, the second throw will be to a height of 4, the third throw to a height of 5, the fourth throw to a height of 3 (as the pattern repeats), and so forth.

While the siteswap pattern describes the heights of the throws, the actual movement of individual balls does not follow as obvious a pattern. The following diagram illustrates the beginning of the ''3 4 5'' pattern.

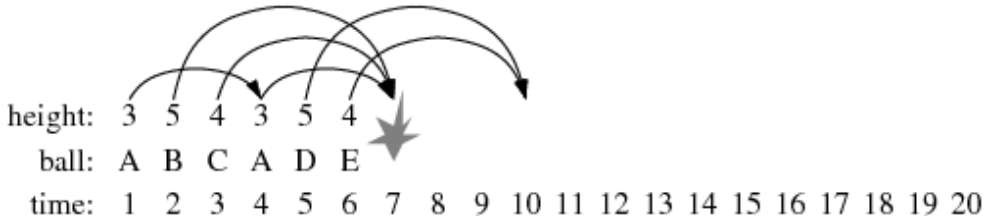


The first throw is ball *A*, thrown to a height of 3, and thus ball *A* is not thrown again until time 4. At time 2, we must make a throw with height 4; since ball *A* has not yet come back, we introduce a second ball, conventionally labeled *B*. Because ball *B* is thrown at time 2 with a height of 4, it will not be thrown again until time 6. At time 3, we introduce yet another ball, labeled *C*, and throw it to height 5 (thus it will next be thrown at time 8). Our next throw, at time 4, is to have height 3. However, since ball *A* has returned (from its throw at time 1), we do not introduce a new ball; we throw *A*. At time 5, we are to make a throw with height 4,

yet we must introduce a new ball, *D*, because balls *A*, *B*, and *C* are all still up in the air. (Ball *D* is the last ball to be introduced for this particular pattern.) The juggling continues with ball *B* being thrown to height 5 at time 6, and so on.

The ''3 4 5'' siteswap pattern works out nicely. It happens to be a 4-ball pattern, because after introducing ball *D*, the juggler can now continue until his or her arms get tired. Unfortunately, not all siteswap sequences are legitimate!

Consider an attempt to use a siteswap pattern ''3 5 4''. If we were only interested in making six throws, everything works well. But a problem arises at time 7, as shown in the following diagram.



Ball *B* was thrown at time 2 with a height of 5. Therefore, it should get its next turn to be thrown at time 7. However, ball *C* was thrown at time 3 with a height of 4, and so it too should get its next turn at time 7. (To add insult to injury, ball *A* gets thrown at time 4 with height of 3, also suggesting it get its next turn at time 7.) What we have here is a problem, resulting in a lot of balls crashing to the ground.

Input

Each line represents a separate trial. It starts with the number $1 \leq P \leq 7$ which represents the period of the repeating pattern, followed by *P* positive numbers that represent the throw heights in the pattern. An individual throw height will be at most 19. The input is terminated with a single line containing the value 0.

The input must be read from standard input.

Output

For each pattern, output a single line describing the first 20 throws for the given pattern, if 20 throws can be legally made. Otherwise, output the word `CRASH`. You need not be concerned with any crashes due to balls landing strictly after time 20.

The output must be written to standard output.

Sample Input	**Sample Output**
3 3 4 5 1 3 3 3 5 4 5 7 7 7 3 1 0	ABCADBACDABCADBACDAB ABCABCABCABCABCABCAB CRASH ABCDEEDABCCBEDAADCBE

G - Bounce

Source file name: `bounce.c`, `bounce.cpp` *or* `bounce.java`

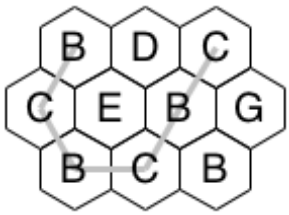


Figure 1

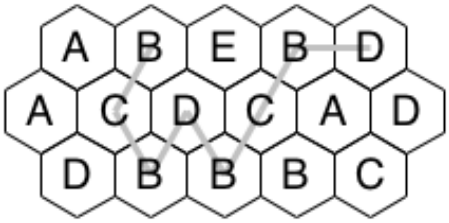


Figure 2

A puzzle adapted from a 2007 *Games Magazine* consists of a collection of hexagonal tiles packed together with each tile showing a letter. A bouncing path in the grid is a continuous path, using no tile more than once, starting in the top row, including at least one tile in the bottom row, and ending in the top row to the right of the starting tile. Continuous means that the next tile in a path always shares an edge with the previous tile.

Each bouncing path defines a sequence of letters. The sequence of letters for the path shown in Figure 1 is `BCBCBC`. Note that this is just `BC` repeated three times. We say a path has a *repetitive pattern of length n* if the whole sequence is composed of two or more copies of the first n letters concatenated together. Figure 2 shows a repetitive pattern of length 4: the pattern `BCBD` repeated twice. Your task is to find bouncing paths with a repetitive pattern of a given length.

In each grid the odd numbered rows will have the same number of tiles as the first row. The even numbered rows will each have one more tile, with the ends offset to extend past the odd rows on both the left and the right.

Input

The input will consist of one to twelve data sets, followed by a line containing only 0.

The first line of a data set contains blank separated integers r c n, where r is the number of rows in the hex pattern ($2 \le r \le 7$), c is the number of entries in the odd numbered rows, ($2 \le c \le 7$), and n is the required pattern length ($2 \le n \le 5$). The next r lines contain the capital letters on the hex tiles, one row per line. All hex tile characters for a row are blank separated. The lines for odd numbered rows also start with a blank, to better simulate the way the hexagons fit together.

The input must be read from standard input.

Output

There is one line of output for each data set. If there is a bouncing path with pattern length n, then output the pattern for the shortest possible path. If there is no such path, output the phrase: `no solution`. The data sets have been chosen such that the shortest solution path is unique, if one exists.

The output must be written to standard output.

Sample Input	**Sample Output**
3 3 2 B D C C E B G B C B 3 5 4 A B E B D A C D C A D D B B B C 3 3 4 B D C C E B G B C B 3 4 4 B D H C C E F G B B C B C 0	BCBCBC BCBDBCBD no solution BCBCBCBC

H - Grade School Multiplication

Source file name: `mult.c`, `mult.cpp` *or* `mult.java`

An educational software company, *All Computer Math* (ACM), has a section on multiplication of integers. They want to display the calculations in the traditional grade school format, like the following computation of 432×5678:

```
    432
   5678
-------
   3456
  3024
 2592
2160
-------
2452896
```

Note well that the final product is printed without any leading spaces, but that leading spaces are necessary on some of the other lines to maintain proper alignment. However, as per our regional rules, there should never be any lines with trailing white space. Note that the lines of dashes have length matching the final product.

As a special case, when one of the digits of the second operand is a zero, it generates a single 0 in the partial answers, and the next partial result should be on the same line rather than the next line down. For example, consider the following product of 200001×90040:

```
     200001
      90040
-----------
    8000040
180000900
-----------
18008090040
```

The rightmost digit of the second operand is a 0, causing a 0 to be placed in the rightmost column of the first partial product. However, rather than continue to a new line, the partial product of 4×200001 is placed on the same line as that 0. The third and fourth least-significant digits of the second operand are zeros, each resulting in a 0 in the second partial product on the same line as the result of 9×200001.

As a final special case, if there is only one line in the partial answer, it constitutes a full answer, and so there is no need for computing a sum. For example, a computation of 246×70 would be formatted as

```
   246
    70
 -----
 17220
```

Your job is to generate the solution displays.

Input

The input contains one or more data sets. Each data set consists of two positive integers on a line, designating the operands in the desired order. Neither number will have more than 6 digits, and neither will have leading zeros. After the last data set is a line containing only 0 0.

The input must be read from standard input.

Output

For each data set, output a label line containing "`Problem `" with the number of the problem, followed by the complete multiplication problem in accordance with the format rules described above.

The output must be written to standard output.

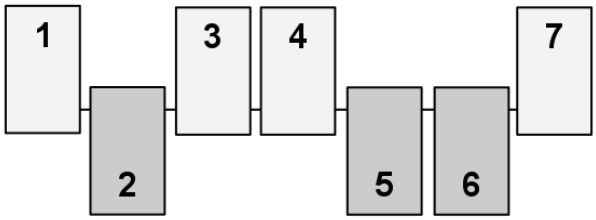
Sample Input	**Sample Output**

432 5678
200001 90040
246 70
0 0
	Problem 1
	  432
	  5678
	------- 	
	  3456
	  3024
	  2592
	2160
	------- 	
	2452896
	Problem 2
	  200001
	  90040
	----------- 	
	  8000040
	180000900
	----------- 	
	18008090040
	Problem 3
	  246
	  70
	----- 	
	17220

I - Shut the Box

Source file name: `shut.c`, `shut.cpp` *or* `shut.java`

Shut the Box is a one-player game that begins with a set of N pieces labeled from 1 to N. All pieces are initially ''unmarked'' (in the picture below, the unmarked pieces are those in an upward position). In the version we consider, a player is allowed up to T turns, with each turn defined by an independently chosen value V (typically determined by rolling one or more dice). During a turn, the player must designate a set of currently unmarked pieces whose numeric labels add precisely to V, and mark them. The game continues either until the player runs out of turns, or until a single turn when it becomes impossible to find a set of unmarked pieces summing to the designated value V (in which case it and all further turns are forfeited). The goal is to mark as many pieces as possible; marking all pieces is known as ''shutting the box.'' Your goal is to determine the maximum number of pieces that can be marked by a fixed sequence of turns.



As an example, consider a game with 6 pieces and the following sequence of turns: 10, 3, 4, 2. The best outcome for that sequence is to mark a total of four pieces. This can be achieved by using the value 10 to mark the pieces 1 + 4 + 5, and then using the value of 3 to mark piece 3. At that point, the game would end as there is no way to precisely use the turn with value 4 (the final turn of value 2 must be forfeited as well). An alternate strategy for achieving the same number of marked pieces would be to use the value 10 to mark four pieces 1 + 2 + 3 + 4, with the game ending on the turn with value 3. But there does not exist any way to mark five or more pieces with that sequence.

Input

Each game begins with a line containing two integers, N, T where $1 \leq N \leq 22$ represents the number of pieces, and $1 \leq T \leq N$ represents the maximum number of turns that will be allowed. The following line contains T integers designating the sequence of turn values for the game; each such value V will satisfy $1 \leq V \leq 22$. You must read that entire sequence from the input, even though a particular game might end on an unsuccessful turn prior to the end of the sequence. The data set ends with a line containing 0 0.

The input must be read from standard input.

Output

You should output a single line for each game, as shown below, reporting the ordinal for the game and the maximum number of pieces that can be marked during that game.

The output must be written to standard output.

Sample Input	**Sample Output**
6 4 10 3 4 2 6 5 10 2 4 5 3 10 10 1 1 3 4 5 6 7 8 9 10 22 22 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 0	Game 1: 4 Game 2: 6 Game 3: 1 Game 4: 22

J - Sokoban

Source file name: `sokoban.c`, `sokoban.cpp` *or* `sokoban.java`

Soko-ban is a Japanese word for a warehouse worker, and the name of a classic computer game created in the 1980s. It is a one-player game with the following premise. A single worker is in an enclosed warehouse with one or more boxes. The goal is to move those boxes to a set of target locations, with the number of target locations equalling the number of boxes. The player indicates a direction of motion for the worker using the arrow keys (up, down, left, right), according to the following rules.

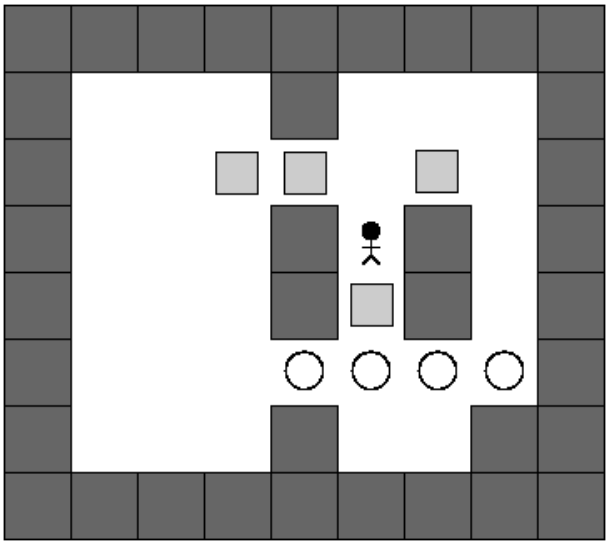
- If the indicated direction of motion for the worker leads to an empty location (i.e., one that does not have a box or wall), the worker advances by one step in that direction.
- If the indicated direction of motion would cause the worker to move into a box, and the location on the other side of the box is empty, then both the worker and the box move one spot in that direction (i.e., the worker pushes the box).
- If the indicated direction of motion for a move would cause the worker to move into a wall, or to move into a box that has another box or a wall on its opposite side, then no motion takes place for that keystroke.

The goal is to simultaneously have all boxes on the target locations. In that case, the player is successful (and as a formality, all further keystrokes will be ignored).

The game has been studied by computer scientists (in fact, one graduate student wrote his entire Ph.D. dissertation about the analysis of sokoban). Unfortunately, it turns out that finding a solution is very difficult in general, as it is both NP-hard and PSPACE-complete. Therefore, your goal will be a simpler task: simulating the progress of a game based upon a player's sequence of keystrokes. For the sake of input and output, we describe the state of a game using the following symbols:

Symbol	Meaning
`.`	empty space
`#`	wall
`+`	empty target location
`b`	box
`B`	box on target location
`w`	worker
`W`	worker on target location

For example, the initial configuration depicted below appears as the first sample input case.



Input

Each game begins with a line containing integers R and C, where $4 \leq R \leq 15$ represents the number of rows, and $4 \leq C \leq 15$ represents the number of columns. Next will be R lines representing the R rows from top to bottom, with each line having precisely C characters, from left-to-right. Finally, there is a line containing at most 50 characters describing the player's sequence of keystrokes, using the symbols U, D, L, and R respectively for up, down, left, and right. You must read that entire sequence from the input, even though a particular game might end successfully prior to the end of the sequence. The data set ends with the line 0 0.

We will guarantee that each game has precisely one worker, an equal number of boxes and locations, at least one initially misplaced box, and an outermost boundary consisting entirely of walls.

The input must be read from standard input.

Output

For each game, you should first output a line identifying the game number, beginning at 1, and either the word `complete` or `incomplete`, designating whether or not the player successfully completed that game. Following that should be a representation of the final board configuration.

The output must be written to standard output.

Sample Input	**Sample Output**
8 9 ######### #...#...# #..bb.b.# #...#w#.# #...#b#.# #...++++# #...#..## ######### ULRURDDDUULLDDD 6 7 ####### #..#### #.+.+.# #.bb#w# ##....# ####### DLLUDLULUURDRDDLUDRR 0 0	Game 1: incomplete ######### #...#...# #..bb...# #...#.#.# #...#.#.# #...+W+B# #...#b.## ######### Game 2: complete ####### #..#### #.B.B.# #.w.#.# ##....# #######