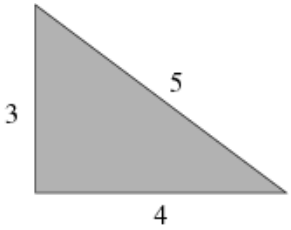
Problem A: Pythagoras's Revenge

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

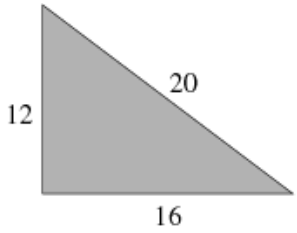
The famous Pythagorean theorem states that a right triangle, having side lengths *A* and *B* and hypotenuse length *C*, satisfies the formula

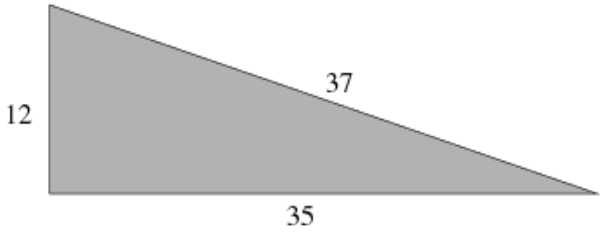
$$A^2 + B^2 = C^2$$

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 integral?

Input: Each line contains a single integer *A*, such that $2 \le A < 1048576 = 2^{20}$. The end of the input is designated by a line containing the value 0.

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 integral length.

Example input:	Example output:
3 12 2 1048574 1048575 0	1 2 0 1 175

A Hint and a Warning: Our hint is that you need not consider any value for *B* that is greater than $(A^2-1)/2$, because for any such right triangle, hypotenuse C satisfies $B < C < B + 1$, and thus cannot have integral length.

Our warning is that for values of $A \approx 2^{20}$, there could be solutions with $B \approx 2^{39}$, and thus values of $C^2 > B^2 \approx 2^{78}$.

You can guarantee yourself 64-bit integer calculations by using the type **`long long`** in C++ or **`long`** in Java. But neither of those types will allow you to accurately calculate the value of C^2 for such an extreme case. (Which is, after all, what makes this **Pythagoras's *revenge***!)

Problem B: Digit Solitaire

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

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 passed some time with the following 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: Each line contains a single integer $1 \le S \le 100000$, designating the starting value. The value *S* will not have any leading zeros. A value of 0 designates the end of the input.

Output: For each nonzero input value, a single line of output should express the ordered sequence of values that are considered during the game, starting with the original value.

Example input:	**Example output:**
95 396 28 4 40 0	95 45 20 0 396 162 12 2 28 16 6 4 40 0

Problem C: Any Way You Slice It

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

The Association for Cutting Machinery (ACM) has just announced a new portable laser capable of slicing through six-inch sheet metal like a hot knife through Jello (or butter, if you're a traditionalist). The laser is mounted on a small motorized vehicle which is programmed to drive over the surface being cut. The vehicle has two operations: it can move forward in a straight line, cutting the surface beneath it as it goes, or it can pivot in place to face a different direction.

Of course, trouble can arise if the laser cuts a hole in the surface, as the surface inside the hole will drop out, and the vehicle will fall in the hole. Your task is to take a set of instructions for the vehicle and decide whether they will result in cutting a hole—that is, if the path that it is cutting ever intersects itself. We will assume that we have an infinite surface, and that the laser makes a cut of zero width.

We assume that the starting location of the laser is (0, 0) and oriented to face in the positive Y direction. Sequences of instructions will always alternate between turn instructions and move instructions. All instructions are relative—e.g., turn a certain number of counterclockwise degrees relative to your current position. For example, suppose we gave the vehicle the following sequence of instructions:

```
TURN  -90
MOVE   10
TURN   90
MOVE    5
TURN  135
MOVE   10
TURN  -90
MOVE    5
```

The vehicle will take the following actions:

1. *Turn -90°.* The laser is still at (0, 0), but now the vehicle is facing in the positive X direction.
2. *Move forward 10.* This moves the laser to location (10, 0).
3. *Turn 90°.* The laser is still at (10, 0), but now the vehicle is facing in the positive Y direction.
4. *Move forward 5.* This moves the laser to location (10, 5). (See Figure 1)
5. *Turn 135°.* Now the vehicle is facing diagonally in the negative XY direction.
6. *Move forward 10.* If this instruction were completed, it would move the laser to approximately (2.93, -2.07). However, along the way the laser will intersect one of the previous cuts it made, making a hole and interfering with the mobility of the vehicle. (See Figure 2) Thus, this instruction (and all following instructions) cannot be completed.

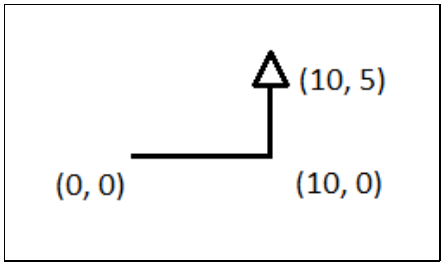


Figure 1:
after 2 turn/move instructions

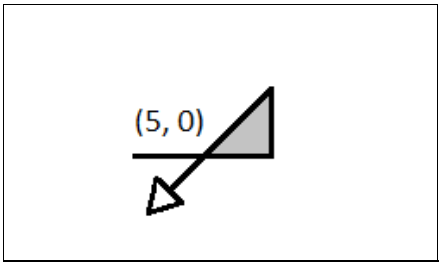


Figure 2:
after 3 turn/move instructions

Input: The input will be a series of at most 20 data sets. Each data set begins with a line containing an integer N representing the number of turn/move combinations that will be executed ($1 \le N \le 100$). A value of zero for N indicates the end of the input.

The next N lines contain the instructions. Each instruction contains two integers T and M, where T is the number of degrees to turn ($-179 \le T \le 179$) and M is the distance to move after that turn ($1 \le M \le 100$).

Note that, although T and M will be integers, you should not assume the position of the laser will always be integer coordinates; in fact, they will typically not be integral. We have taken care to choose data sets for which lines that intersect do so sufficiently away from an endpoint, and that lines that do not intersect remain sufficiently separated from each other.

Output: For each data set you should output the number of the *first* move instruction that will create a hole (e.g., output 3 if the third move creates the hole). Note that once you have detected a hole, the remaining instructions are irrelevant and may be ignored (but must still be read from the input).

If the entire set of instructions can be carried out without creating a hole, then print the word **SAFE**.

Example input:	Example output:
4 -90 10 90 5 135 10 -90 5 4 -90 10 90 5 135 5 -90 10 6 60 10 60 10 60 10 60 10 60 10 80 20 0	3 SAFE 6

Problem D: Is the Name of This Problem

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

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.

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.

You must decide whether each sentence is the result of a Quine operation. To be a Quine, 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 it matches this pattern, the sentence is a Quine of the phrase *A*. Note that phrase A must contain the exact same sequence of characters both times it appears.

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`**.

Example input:	Example 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

A review of quotation marks in strings: As a reminder, the quotation mark character is a regular character, and can be referred to in C, C++, and Java using the standard single-quote notation, like so:

```
'"'
```

However, to place a quotation mark inside a double-quoted string in C, C++, and Java, you must place a backslash (\) in front of it. If you do not it will be interpreted as the end of the string, causing syntax errors. For example:

```
"This quotation mark \" is inside the string"
"\""
"\"SAID SHE\" SAID SHE"
```

Problem E: ASCII Addition

Time limit: 1 s
Memory limit: 512 MiB

Nowadays, there are smartphone applications that instantly translate text and even solve math problems if you just point your phone's camera at them. Your job is to implement a much simpler functionality reminiscent of the past – add two integers written down as ASCII art.

An *ASCII art* is a matrix of characters, exactly 7 rows high, with each individual character either a dot or the lowercase letter x.

An expression of the form $a + b$ is given, where both a and b are positive integers. The expression is converted into ASCII art by writing all the expression characters (the digits of a and b as well as the + sign) as 7×5 matrices, and concatenating the matrices together with a single column of dot characters between consecutive individual matrices. The exact matrices corresponding to the digits and the + sign are as folows:

```
xxxxx  ....x  xxxxx  xxxxx  x...x  xxxxx  xxxxx  xxxxx  xxxxx  xxxxx  .....
x...x  ....x  ....x  ....x  x...x  x....  x....  ....x  x...x  x...x  ..x..
x...x  ....x  ....x  ....x  x...x  x....  x....  ....x  x...x  x...x  ..x..
x...x  ....x  xxxxx  xxxxx  xxxxx  xxxxx  xxxxx  ....x  xxxxx  xxxxx  xxxxx
x...x  ....x  x....  ....x  ....x  ....x  x...x  ....x  x...x  ....x  ..x..
x...x  ....x  x....  ....x  ....x  ....x  x...x  ....x  x...x  ....x  ..x..
xxxxx  ....x  xxxxx  xxxxx  ....x  xxxxx  xxxxx  ....x  xxxxx  xxxxx  .....
```

Given an ASCII art for an expression of the form $a + b$, find the result of the addition and write it out in the ASCII art form.

Input

Input consists of exactly 7 lines and contains the ASCII art for an expression of the form $a + b$, where both a and b are positive integers consisting of at most 9 decimal digits and written without leading zeros.

Output

Output 7 lines containing ASCII art corresponding to the result of the addition, without leading zeros.

Example

input

```
....x.xxxxx.xxxxx.x...x.xxxxx.xxxxx.xxxxx.......xxxxx.xxxxx.xxxxx
....x.....x.....x.x...x.x.....x.........x...x...x...x.x...x.x...x
....x.....x.....x.x...x.x.....x.........x...x...x...x.x...x.x...x
....x.xxxxx.xxxxx.xxxxx.xxxxx.xxxxx.....x.xxxxx.xxxxx.xxxxx.x...x
....x.x.........x.....x.....x.x...x.....x...x...x...x.....x.x...x
....x.x.........x.....x.....x.x...x.....x...x...x...x.....x.x...x
....x.xxxxx.xxxxx.....x.xxxxx.xxxxx.....x.......xxxxx.xxxxx.xxxxx
```

output

```
....x.xxxxx.xxxxx.xxxxx.x...x.xxxxx.xxxxx
....x.....x.....x.x.....x...x.x.........x
....x.....x.....x.x.....x...x.x.........x
....x.xxxxx.xxxxx.xxxxx.xxxxx.xxxxx.....x
....x.x.........x.....x.....x.....x.....x
....x.x.........x.....x.....x.....x.....x
....x.xxxxx.xxxxx.xxxxx.....x.xxxxx.....x
```

Problem F: LRU Caching

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

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.

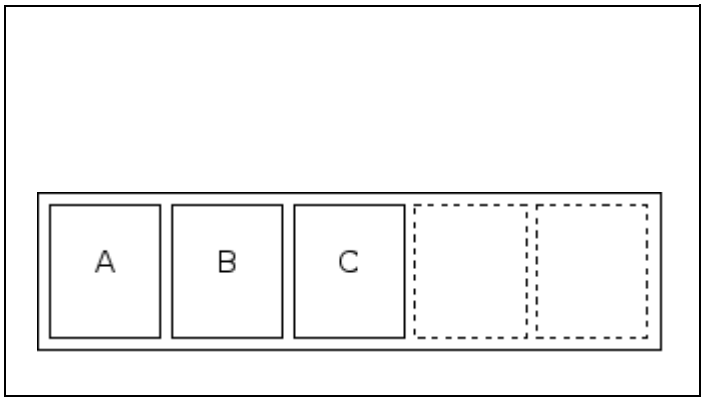


Figure 1: Cache after A, B, C

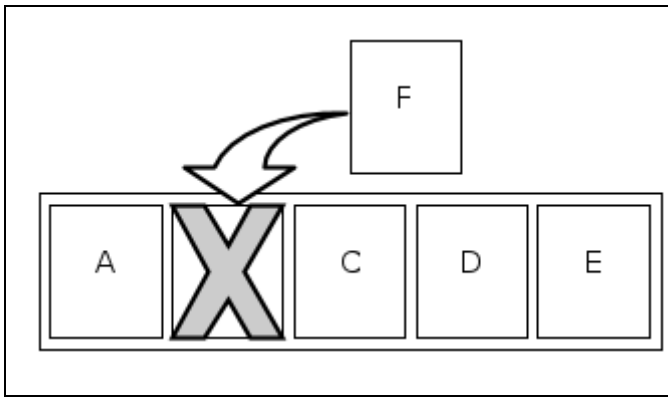


Figure 2: Cache after A, B, C, D, E, A, F

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.

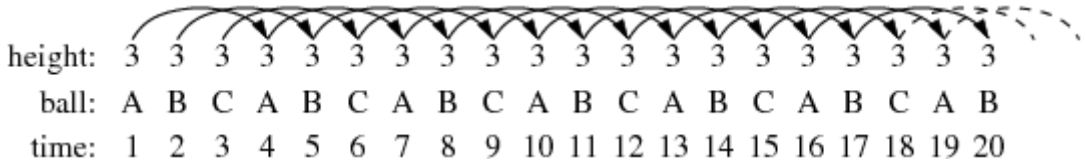
Output: For each data set you should output the line "Simulation *S*", where *S* is 1 for the first data set, 2 for the second data set, etc. Then for each exclamation mark in the data set you should output the contents of the cache on one line as a sequence of characters representing the pieces of data 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.

Example input:	**Example 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

Problem G: Jugglefest

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

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.

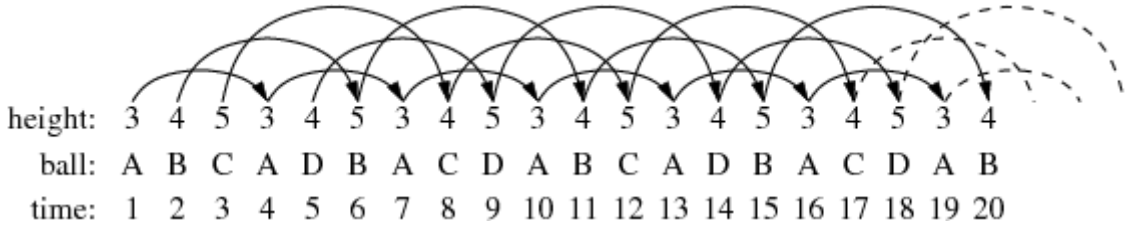


By convention, we label the first ball thrown as A, and each time we introduce a new ball into the pattern, we label it with the next consecutive uppercase letter (hence B and then C in the classic pattern).

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.

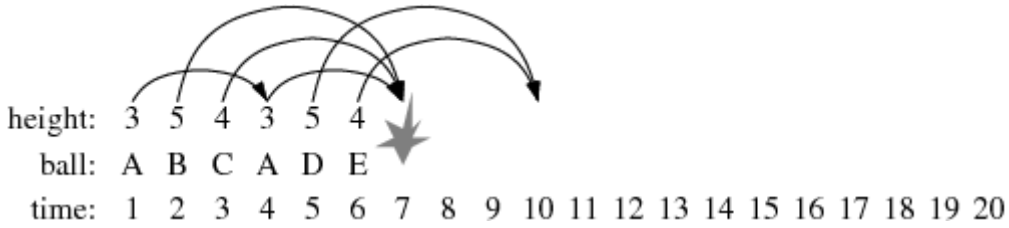
While the siteswap pattern describes the heights of the throws, the actual movement of individual balls does not follow as obvious a pattern. The following diagram illustrates the beginning of the "3 4 5" pattern.



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.

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.

Example input:	**Example output:**
3 3 4 5 1 3 3 3 5 4 5 7 7 7 3 1 0	ABCADBACDABCADBACDAB ABCABCABCABCABCABCAB CRASH ABCDEEDABCCBEDAADCBE

Problem H: Bounce

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

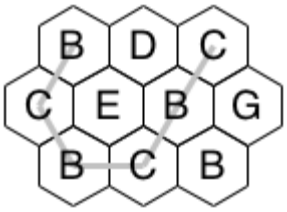


Figure 1

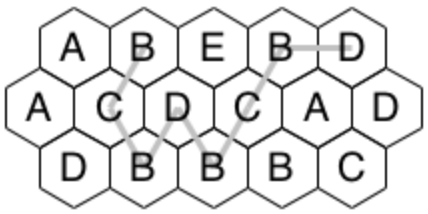


Figure 2

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 four: 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.

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.

Example input:	Example 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

Last modified on October 18, 2012.

Problem I: Book Borders

Time limit: 2 s
Memory limit: 512 MiB

A book is being typeset using a fixed width font and a simple greedy algorithm to fill each line. The book contents is just a sequence of words, where each word contains one or more characters.

Before typesetting, we choose a *maximum line length* and denote this value with m. Each line can be at most m characters long including the space characters between the words. The typesetting algorithm simply processes words one by one and prints each word with exactly one space character between two consecutive words on the same line. If printing the word on the current line would exceed the maximum line length m, a new line is started instead.

```
|its.a.long...|          |its.a.long.way|
|way.to.the...|          |to.the.top.if.|
|top.if.you...|          |you.wanna.rock|
|wanna.rock.n.|          |n.roll........|
|roll.........|
```

Text from the example input with maximum line lengths 13 and 14

You are given a text to be typeset and are experimenting with different values of the maximum line length m. For a fixed m, the *leading sentence* is a sentence (a sequence of words separated with a single space character) formed by the first words of lines top to bottom. In the example above, when the sample text is typeset with the maximum line length 14, the leading sentence is "`its to you n`".

Given a text and two integers a and b, find the length of the leading sentence for every candidate maximum line length between a and b inclusive. The length of a sentence is the total number of characters it contains including the space characters.

Input

The first line contains the text to be typeset – a sequence of words separated by exactly one space character. Each word is a string consisting of one or more lowercase letters from the English alphabet.

The second line contains two integers a and b – the edges of the interval we are interested in, as described above.

It is guaranteed that $1 \leq w \leq a \leq b \leq z \leq 500\,000$, where w is the length of the longest word in the text and z is the total number of characters in the text including the space characters.

Output

Output $b - a + 1$ lines – the k-th of those lines should contain a single integer – the total length of the leading sentence when the maximum line length is equal to $a - 1 + k$.

Example

input	output

`its a long way to the top if you wanna rock n roll`
`13 16`
	`22`
	`12`
	`12`
	`15`

Problem J: Kernel Knights

Time limit: 2 s
Memory limit: 512 MiB

Jousting is a medieval contest that involves people on horseback trying to strike each other with wooden lances while riding at high speed. A total of $2n$ knights have entered a jousting tournament – n knights from each of the two great rival houses. Upon arrival, each knight has challenged a single knight from the other house to a duel.

A *kernel* is defined as some subset S of knights with the following two properties:

- No knight in S was challenged by another knight in S.
- Every knight not in S was challenged by some knight in S.

Given the set of the challenges issued, find one kernel. It is guaranteed that a kernel always exists.

Input

The first line contains an integer n ($1 \leq n \leq 100\,000$) – the number of knights of each house. The knights from the first house are denoted with integers 1 through n, knights from the second house with integers $n+1$ through $2n$.

The following line contains integers $f_1, f_2, \ldots, f_n$ – the k-th integer f_k is the index of the knight challenged by knight k ($n+1 \leq f_k \leq 2n$).

The following line contains integers $s_1, s_2, \ldots, s_n$ – the k-th integer s_k is the index of the knight challenged by knight $n+k$ ($1 \leq s_k \leq n$).

Output

Output the indices of the knights in the kernel on a single line. If there is more than one solution, you may output any one.

Example

input

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

output

```
1 2 4 8
```