# Problem A. Aerodynamics

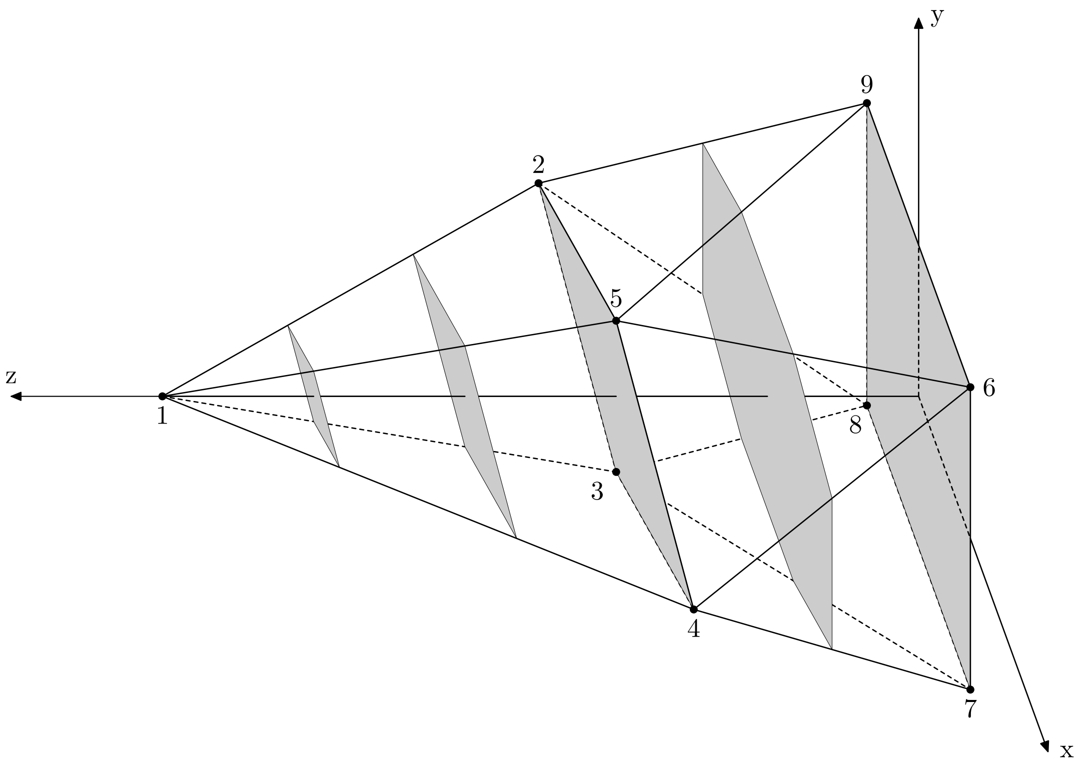
| | |
|---|---|
| Input file: | `aerodynamics.in` |
| Output file: | `aerodynamics.out` |

Bill is working in a secret laboratory. He is developing missiles for national security projects. Bill is the head of the aerodynamics department.

One surprising fact of aerodynamics is called Whitcomb area rule. An object flying at high-subsonic speeds develops local supersonic airflows and the resulting shock waves create the effect called wave drag. Wave drag does not depend on the exact form of the object, but rather on its *cross-sectional profile*.

Consider a coordinate system with $OZ$ axis pointing in the direction of object's motion. Denote the area of a section of the object by a plane $z = z_0$ as $S(z_0)$. Cross-sectional profile of the object is a function $S$ that maps $z_0$ to $S(z_0)$. There is a perfect aerodynamic shape called Sears-Haack body. The closer cross-sectional profile of an object to the cross-sectional profile of Sears-Haack body, the less wave drag it introduces. That is an essence of Whitcomb area rule.

Bill's department makes a lot of computer simulations to study missile's aerodynamic properties before it is even built. To approximate missile's cross-sectional profile one takes samples of $S(z_0)$ for integer arguments $z_0$ from $z_{min}$ to $z_{max}$.




Your task is to find the area $S(z_0)$ for each integer $z_0$ from $z_{min}$ to $z_{max}$, inclusive, given the description of the missile. The description of the missile is given to you as a set of points. The missile is the minimal convex solid containing all the given points. It is guaranteed that there are four points that do not belong to the same plane.

## Input

The first line of the input file contains three integer numbers: $n$, $z_{min}$ and $z_{max}$ ($4 \le n \le 100$, $0 \le z_{min} \le z_{max} \le 100$). The following $n$ lines contain three integer numbers each: $x$, $y$, and $z$ coordinates of the given points. All coordinates do not exceed 100 by their absolute values. No two points coincide. There are four points that do not belong to the same plane.

## Output

For each integer $z_0$ from $z_{min}$ to $z_{max}$, inclusive, output one floating point number: the area $S(z_0)$. The area must be precise to at least 5 digits after decimal point.

## Sample input and output

| aerodynamics.in | aerodynamics.out |
|---|---|
| 9 0 5<br>0 0 5<br>-3 0 2<br>0 -1 2<br>3 0 2<br>0 1 2<br>2 2 0<br>2 -2 0<br>-2 -2 0<br>-2 2 0 | 16.00000<br>14.92000<br>10.08000<br>4.48000<br>1.12000<br>0.00000 |

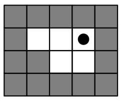
# Problem B. Blind Walk

Input file: `standard input`
Output file: `standard output`

This is an interactive problem.

Your task is to write a program that controls a robot which blindly walks through a maze. The maze is $n \times m$ ($1 \le n, m \le 30$) rectangular grid that consists of square cells. Each cell is either empty or blocked. All cells on the border of the maze are blocked. The robot starts in an empty cell. It can move south, west, north, or east to an adjacent empty cell. The robot is blind and has only bump sensors, so when it attempts to move it can either succeed or bump into blocked cell and fail.

The robot has to visit all empty cells in the maze. All cells are guaranteed to be reachable.



The picture shows sample maze where blocked cells are, filled and initial robot's location is designated with a circle.

## Interaction protocol

The program must write to the standard output one line with robot's action and wait for a line in the standard input with a response, then write next action and read next response, and so on until all empty cells in the maze had been visited. The program must exit only when all cells have been visited. Empty cells may be visited multiple times. It is acceptable to move even after all cells had been visited.

## Output

Each line of the standard output represents robot's action. It is one of the following five strings: `SOUTH`, `WEST`, `NORTH`, `EAST`, or `DONE`. `DONE` must be printed when the robot has visited all empty cells. After printing `DONE` your program must exit. You must flush standard output after printing each action.

## Input

Each line of the standard input represents response on robot's action. It is either a string `EMPTY` if robot has successfully moved in the specified direction to an adjacent cell or a string `BLOCKED` if robot's movement has failed because the corresponding adjacent cell was blocked.

## Sample input and output

| standard output | standard input |
|---|---|
| NORTH | BLOCKED |
| EAST | BLOCKED |
| SOUTH | EMPTY |
| EAST | BLOCKED |
| SOUTH | BLOCKED |
| WEST | EMPTY |
| SOUTH | BLOCKED |
| WEST | BLOCKED |
| NORTH | EMPTY |
| WEST | EMPTY |
| WEST | BLOCKED |
| NORTH | BLOCKED |
| EAST | EMPTY |
| NORTH | BLOCKED |
| DONE | |

# Problem C. Clock

Input file: `clock.in`
Output file: `clock.out`

One famous Russian architect plans to build a new monumental construction. It will be a huge clock that indicates the time from the beginning of the universe.

The face of this clock contains hands, moving at constant speeds. They are numbered from 1 to $n$ from the fastest to the slowest one. The fastest hand makes one revolution per minute (60 seconds). Each next hand moves slower than previous, the $(i+1)$-th hand makes one revolution when the $i$-th hand makes $d_i$ revolutions.

The setting mechanism of this clock is very simple. You can take a hand by the handle, located on its end, and move it in any direction. When you move the hand, slower hands are moving in proportion to their usual speeds, and faster hands are not moving. Remember that hands are huge, so setting this clock is a hard job.

Consider an example with three hands: a second hand, a minute hand, and an hour hand. Their lengths are 5, 15 and 10 meters respectively. You want to set the clock from 2:30 to 6:00 (fig. 1). The easiest way to do it is to rotate the minute hand 180° clockwise, and then move the hour hand 90° clockwise. The total distance you moved the handles of the hands is approximately 62.83 meters.

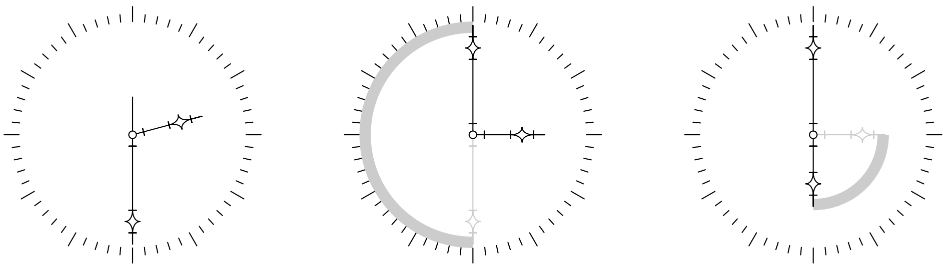


Fig. 1. Setting clock from 2:30 to 6:00.

Your task is to write a program that finds the way to set the clock that minimizes the total distance you have to move the handles.

## Input

The first line of the input file contains one integer $n$ — the number of hands ($0 < n \le 50$). The second line contains $n-1$ integer numbers $d_1, d_2, \ldots, d_{n-1}$ ($2 \le d_i \le 10^6$). The third line contains $n$ integer numbers $l_1, l_2, \ldots, l_n$ ($1 \le l_i \le 10^6$) — lengths of clock hands. Next two lines contain two non-negative integer numbers (one number per line): time indicated by the clock and the actual time that should be set. Both times are measured in seconds from the beginning of the universe and are less than $2^{63}$.

## Output

Print the minimal possible total distance you have to move the handles. The answer must be precise to at least 4 digits after decimal point.

## Sample input and output

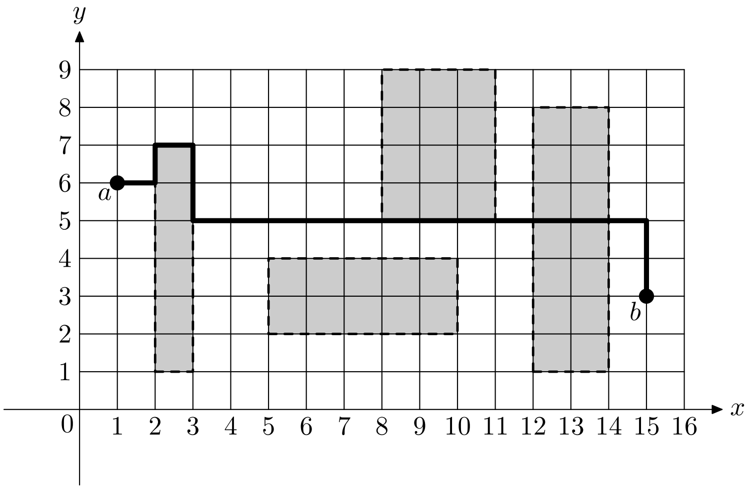
| clock.in | clock.out |
|---|---|
| 3<br>60 12<br>5 15 10<br>52200<br>453600 | 62.831853072 |

# Problem D. Drive through MegaCity

| | |
|---|---|
| Input file: | `drive.in` |
| Output file: | `drive.out` |

MegaCity of the future is a rectangular grid of streets. Each intersection has integer Cartesian coordinates $x$ and $y$. To get from intersection $a$ with coordinates $x_a$, $y_a$ to intersection $b$ with coordinates $x_b$, $y_b$ you need to drive exactly $|x_a - x_b| + |y_a - y_b|$ blocks. Usually, it takes 10 time units to drive one block, so one can easily compute the time it takes to get from $a$ to $b$. However, traffic jams that occur in MegaCity turn estimation of minimal driving time into a complex problem that you have to solve.

Traffic jams in MegaCity affect a rectangular area that is specified by coordinates of its bottom-left and top-right corners. The time to travel one block in the traffic jam is specified. All streets that are strictly inside the rectangular region are affected by the traffic jam. Sometimes, it is better to drive around the traffic jams to save time, but sometimes it is better to drive through some traffic jams as shown in the example — 17 blocks are driven outside of traffic jams, taking 10 time units per block, and 2 blocks in the light traffic jam with 11 time units per block.




## Input

The first line of the input file contains four integer numbers $x_a$, $y_a$, $x_b$, and $y_b$ — coordinates of the start and finish intersections. The second line of the input file contains a single integer number $n$ ($0 \le n \le 1000$) which specifies the number of traffic jams. The following $n$ lines describe traffic jams. Each traffic jam is described by five integer numbers $x_{1,i}$, $y_{1,i}$, $x_{2,i}$, $y_{2,i}$ and $t_i$, where first four numbers are coordinates of the bottom-left and top-right corners of the jammed area ($x_{1,i} < x_{2,i}$, $y_{1,i} < y_{2,i}$), and $t_i$ ($10 < t_i \le 10^8$) is the time it takes to travel one block inside this traffic jam. All coordinates in the input file are from 0 to $10^8$ inclusive. Areas of traffic jams neither intersect nor touch each other. Start and finish points are different and do not lie inside nor on the border of any traffic jam.

## Output

Write to the output file a single integer — the minimal driving time from intersection $a$ to intersection $b$.

## Sample input and output

| drive.in | drive.out |
|---|---|
| 1 6 15 3<br>4<br>2 1 3 7 44<br>5 2 10 4 33<br>8 5 11 9 22<br>12 1 14 8 11 | 192 |

# Problem E. Exclusive Access

Input file: `exclusive.in`
Output file: `exclusive.out`

One important problem in concurrent programming is to ensure exclusive access to shared resources by multiple threads. It is also known as Mutual Exclusion protocol. A code that needs to be protected from concurrent execution is called *critical section* (*CS*). In order to coordinate access to CS, application threads use a set of shared variables to send information to each other. These shared variables are distinct from all the variables that are used by application code. In practice, mutual exclusion protocol is implemented as two methods — *enterCS* and *exitCS*. When application needs to execute some code in CS, it calls enterCS, then executes CS, then calls exitCS.

For theoretical analysis of mutual exclusion protocol one must consider running application as a whole. Each thread of application is represented as an infinite loop that repeatedly performs some work unrelated to CS, which is called *non-critical section* (*NCS*), then calls enterCS, then executes CS, then calls exitCS, then the loop repeats. The code inside NCS and CS is not relevant; it is considered to perform no operations related to the protocol and does not modify shared variables used by the protocol.

We consider a system with two concurrently running threads. Threads use a set of shared one-bit variables to implement mutual exclusion protocol. Each variable can store a value of zero or one that can be read or written by a single instruction. Shared variables are initialized to zero. Each thread has a local pointer to the instruction (*IP*) that it is going to execute next. Execution starts from the top of the code. During each step of execution one of the threads is arbitrarily chosen, it executes one instruction, and then changes its IP to the next instruction to execute. This infinite sequence of steps is called *history*. A history is called *legal* if either both threads execute infinitely many steps or just one thread does, while the other thread, having taken a finite number of steps, stops with IP at NCS.

The table below contains several algorithms in pseudo-code that attempt to implement mutual exclusion protocol. In this pseudo-code *id* is 0 for the first thread and 1 for the second. Variables *want*[0], *want*[1], and *turn* are shared between threads to implement mutual exclusion protocol. Lines marked with "+" implement enterCS, lines marked with "-" implement exitCS. Lines NCS() and CS() are placeholders for some code that works inside non-critical and critical sections respectively and is not relevant for this problem.

<table>
<tr><th>Algorithm 1</th><th>Algorithm 2</th><th>Algorithm 3</th></tr>
<tr><td><pre>loop forever
  NCS()
+ loop while
+   (turn == 1 - id)
  CS()
- turn <- (1 - id)
end loop</pre></td><td><pre>loop forever
  NCS()
+ want[id] <- 1
+ loop while
+   (want[1 - id] == 1)
  CS()
- want[id] <- 0
end loop</pre></td><td><pre>loop forever
   NCS()
+  want[id] <- 1
+  turn <- (1 - id)
+  loop while
+    (want[1 - id] == 1 and
+     turn == 1 - id)
   CS()
-  want[id] <- 0
end loop</pre></td></tr>
</table>

The task is to figure out if the given algorithm satisfies three important properties:

- The algorithm satisfies *mutual exclusion* if in any legal history CS is not executed concurrently by two threads (that is, there is no step where IP of both threads is at CS).
- The algorithm satisfies *deadlock freedom* if any legal history has infinitely many executions of CS.
- The algorithm satisfies *starvation freedom* if in any legal history a thread that executes infinitely many steps has infinitely many executions of CS.

The property of mutual exclusion is trivial. The algorithm that simply loops forever doing nothing will satisfy it. The sample algorithms above all satisfy mutual exclusion, but the first two fail to achieve deadlock freedom. The algorithm 3 (originally created by Gary Peterson) satisfies all three properties.

## Input

The input file starts with a line with two integer numbers — $m_1$ and $m_2$, where $m_i$ is the number of lines of code for $i$-th thread ($2 \le m_i \le 9$). It is followed by $m_1$ lines with the code for the first thread and $m_2$ lines with the code for the second thread.

The code for each thread contains one instruction per line. Instruction starts with an integer line number from 1 to $m_i$ (lines are numbered in ascending order and are included to aid readability), followed by instruction mnemonic, followed by a list of instruction arguments, all separated by spaces. The last arguments of instruction represent line numbers of the next instructions to execute (*NIP* — from 1 to $m_i$). There are three variables shared between threads — A, B, and C. Instruction mnemonics are:

- NCS — non-critical section placeholder. Its single argument is NIP.
- CS — critical section placeholder. Its single argument is NIP.
- SET — write value to the shared variable. It has three arguments $v$, $x$, and $g$, where $v$ is the variable to write (A, B, or C), $x$ is the value to write (0 or 1), and $g$ is NIP.
- TEST — read and test the value of the shared variable. It has three arguments $v$, $g_0$, and $g_1$ where $v$ is the variable to read (A, B, or C), $g_0$ is NIP if the value of the variable is zero, and $g_1$ is NIP if the value of the variable is one.

NCS and CS appear in the code for each thread exactly once. The code may or may not represent a simple loop, but is guaranteed to alternate executions of CS and NCS by one thread, that is, in every legal history two executions of CS by one thread always have NCS execution by the same thread in between and, vice versa, two executions of NCS by one thread have CS execution by the same thread in between.

## Output

Write to the output file a string of three letters. Letters represent properties of mutual exclusion, deadlock freedom, and starvation freedom. Write letter Y if the corresponding property is satisfied and N otherwise.

## Sample input and output

Three samples below represent algorithms 1–3 from the problem statement.

<table>
<tr><th>exclusive.in</th><th>exclusive.out</th></tr>
<tr><td><pre>4 4
1 NCS 2
2 TEST C 3 2
3 CS 4
4 SET C 1 1
1 NCS 2
2 TEST C 2 3
3 CS 4
4 SET C 0 1</pre></td><td><pre>YNN</pre></td></tr>
<tr><td><pre>5 5
1 NCS 2
2 SET A 1 3
3 TEST B 4 3
4 CS 5
5 SET A 0 1
1 NCS 2
2 SET B 1 3
3 TEST A 4 3
4 CS 5
5 SET B 0 1</pre></td><td><pre>YNN</pre></td></tr>
</table>

| exclusive.in | exclusive.out |
|---|---|
| 7 7<br>1 NCS 2<br>2 SET A 1 3<br>3 SET C 1 4<br>4 TEST B 6 5<br>5 TEST C 6 4<br>6 CS 7<br>7 SET A 0 1<br>1 NCS 2<br>2 SET B 1 3<br>3 SET C 0 4<br>4 TEST A 6 5<br>5 TEST C 4 6<br>6 CS 7<br>7 SET B 0 1 | YYY |

This is an algorithm (originally created by Leslie Lamport) that uses just two shared bits (A and B) and satisfies mutual exclusion and deadlock freedom, but is not free from starvation.

| exclusive.in | exclusive.out |
|---|---|
| 5 7<br>1 NCS 2<br>2 SET A 1 3<br>3 TEST B 4 3<br>4 CS 5<br>5 SET A 0 1<br>1 NCS 2<br>2 SET B 1 3<br>3 TEST A 6 4<br>4 SET B 0 5<br>5 TEST A 2 5<br>6 CS 7<br>7 SET B 0 1 | YYN |

There are two trivial algorithms. First one never executes CS nor NCS and thus guarantees mutual exclusion, but does not have deadlock freedom, nor starvation freedom properties. Second one loops between NCS and CS, thus fails to achieve mutual exclusion, but is free from deadlock and starvation.

| exclusive.in | exclusive.out |
|---|---|
| 3 3<br>1 SET A 0 1<br>2 CS 2<br>3 NCS 3<br>1 TEST A 1 1<br>2 CS 2<br>3 NCS 3 | YNN |
| 2 2<br>1 CS 2<br>2 NCS 1<br>1 NCS 2<br>2 CS 1 | NYY |

# Problem F. Fibonacci System

Input file: `fibonacci.in`
Output file: `fibonacci.out`

Little John studies numeral systems. After learning all about fixed-base systems, he became interested in more unusual cases. Among those cases he found a *Fibonacci system*, which represents all natural numbers in an unique way using only two digits: zero and one. But unlike usual binary scale of notation, in the Fibonacci system you are not allowed to place two `1`s in adjacent positions.

One can prove that if you have number $N = \overline{a_n a_{n-1} \ldots a_1}_F$ in Fibonacci system, its value is equal to $N = a_n \cdot F_n + a_{n-1} \cdot F_{n-1} + \ldots + a_1 \cdot F_1$, where $F_k$ is a usual Fibonacci sequence defined by $F_0 = F_1 = 1$, $F_i = F_{i-1} + F_{i-2}$.

For example, first few natural numbers have the following unique representations in Fibonacci system:

$$\begin{aligned}
1 &= 1_F \\
2 &= 10_F \\
3 &= 100_F \\
4 &= 101_F \\
5 &= 1000_F \\
6 &= 1001_F \\
7 &= 1010_F
\end{aligned}$$

John wrote a very long string (consider it infinite) consisting of consecutive representations of natural numbers in Fibonacci system. For example, the first few digits of this string are `110100101100010011010`...

He is very interested, how many times the digit `1` occurs in the $N$-th prefix of the string. Remember that the *$N$-th prefix* of the string is just a string consisting of its first $N$ characters.

Write a program which determines how many times the digit `1` occurs in $N$-th prefix of John's string.

## Input

The input file contains a single integer $N$ ($0 \le N \le 10^{15}$).

## Output

Output a single integer — the number of `1`s in $N$-th prefix of John's string.

## Sample input and output

| fibonacci.in | fibonacci.out |
|---|---|
| 21 | 10 |

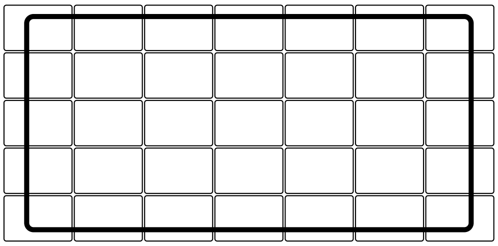
# Problem G. Giant Screen

Input file: `giant.in`
Output file: `giant.out`

You are working in Advanced Computer Monitors (ACM), Inc. The company is building and selling giant computer screens that are composed from multiple smaller screens. Your are responsible for design of the screens for your customers.

Customers order screens of the specified horizontal and vertical resolution in pixels and a specified horizontal and vertical size in millimeters. Your task is to design a screen that has a required resolution in each dimension or more, and has required size in each dimension or more, with a minimal possible price. The giant screen is always built as a grid of monitors of the same type. The total resolution, size, and price of the resulting screen is simply the sum of resolutions, sizes, and prices of the screens it is built from.

You have a choice of regular monitor types that you can order and you know their resolutions, sizes, and prices. The screens of each type can be mounted both vertically and horizontally, but the whole giant screen must be composed of the screens of the same type in the same orientation. You can use as many screens of the chosen type as you need.



## Input

The first line of the input file contains four integer numbers $r_h$, $r_v$, $s_h$, and $s_v$ (all from 100 to 10 000 inclusive) — horizontal and vertical resolution and horizontal and vertical size of the screen you have to build, respectively. The next line contains a single integer number $n$ ($1 \le n \le 100$) — the number of different screen types available to you. The next $n$ lines contain descriptions of the available screen types. Each description occupies one line and consists of five integer numbers — $r_{h,i}$, $r_{v,i}$, $s_{h,i}$, $s_{v,i}$, $p_i$ (all from 100 to 10 000 inclusive), where first four numbers are horizontal and vertical resolution and horizontal and vertical size of $i$-th screen type, and $p_i$ is the price.

## Output

Write to the output file a single integer — the minimal price of the specified giant screen.

## Sample input and output

<table>
<tr><th>giant.in</th><th>giant.out</th></tr>
<tr><td>1024 1024 300 300<br>3<br>1024 768 295 270 200<br>1280 1024 365 301 250<br>1280 800 350 270 210</td><td>250</td></tr>
<tr><td>2400 2000 800 700<br>3<br>1024 768 295 270 200<br>1280 1024 365 301 250<br>1280 800 350 270 210</td><td>1260</td></tr>
</table>

# Problem H. Hell on the Markets

Input file: `hell.in`
Output file: `hell.out`

Most financial institutions had become insolvent during financial crisis and went bankrupt or were bought by larger institutions, usually by banks. By the end of financial crisis of all the financial institutions only two banks still continue to operate. Financial markets had remained closed throughout the crisis and now regulators are gradually opening them. To prevent speculation and to gradually ramp up trading they will initially allow trading in only one financial instrument and the volume of trading will be limited to $i$ contracts for $i$-th minute of market operation.

Two banks had decided to cooperate with the government to kick-start the market operation. The boards of directors had agreed on trading volume for each minute of this first trading session. One bank will be buying $a_i$ contracts ($1 \le a_i \le i$) during $i$-th minute ($1 \le i \le n$), while the other one will be selling. They do not really care whether to buy or to sell, and the outside observer will only see the volume $a_i$ of contracts traded per minute. However, they do not want to take any extra risk and want to have no position in the contract by the end of the trading session. Thus, if we define $b_i = 1$ when the first bank is buying and $b_i = -1$ when the second one is buying (and the first one is selling), then the requirement for the trading session is that $\sum_{i=1}^{n} a_i b_i = 0$.

Your lucky team of three still works in the data center (due to the crisis, banks now share the data center and its personnel) and your task is to find such $b_i$ or to report that this is impossible.

## Input

The first line of the input file contains the single integer number $n$ ($1 \le n \le 100\,000$).

The second line of the input file contains $n$ integer numbers — $a_i$ ($1 \le a_i \le i$).

## Output

The first line of the output file must contain "`Yes`" if the trading session with specified volumes is possible and "`No`" otherwise. In the former case the second line must contain $n$ numbers — $b_i$.

## Sample input and output

| hell.in | hell.out |
|---|---|
| 4<br>1 2 3 4 | Yes<br>1 -1 -1 1 |
| 4<br>1 2 3 3 | No |

# Problem I. iSharp

Input file: `isharp.in`
Output file: `isharp.out`

You are developing a new fashionable language that is not quite unlike C, C++, and Java. Since your language should become an object of art and fashion, you call it $i^{\#}$ (spelled i-sharp). This name combines all the modern naming trends and also hints at how smart you are.

Your language caters for a wide auditory of programmers and its type system includes arrays (denoted with "`[]`"), references (denoted with "`&`"), and pointers (denoted with "`*`"). Those type constructors can be freely combined in any order, so that a pointer to an array of references of references of integers (denoted with "`int&&[]*`") is a valid type.

Multiple variables in $i^{\#}$ can be declared on a single line with a very convenient syntax where common type of variables is given first, followed by a list of variables, each optionally followed by additional variable-specific type constructors. For example, the following line:

```
int& a*[]&, b, c*;
```

declares variables a, b, and c with types "`int&&[]*`", "`int&`", and "`int&*`" correspondingly. Note, that type constructors on the right-hand sides of variables in $i^{\#}$ bind to variable and their order is reversed when they are moved to the left-hand side next to type. Thus "`int*& a`" is equivalent to "`int a&*`".

However, you discover that coding style with multiple variable declarations per line is confusing and is outlawed in many corporate coding standards. You decide to get rid of it and refactor all existing $i^{\#}$ code to a single variable declaration per line and always specify type constructor next to the type it refers to (instead of the right-hand side of variable). Your task it to write such refactoring tool.

## Input

The input file contains a single line with a declaration of multiple variables in $i^{\#}$. The line starts with a type name, followed by zero, one, or more type constructors, followed by a space, followed by one or more variable descriptors separated by "`,`" (comma) and space, and terminated by "`;`" (semicolon). Each variable descriptor contains variable name, followed by zero, one, or more type constructors.

Type name and variable names are distinct and consist of lowercase and uppercase English letters from "`a`" to "`z`" or "`A`" to "`Z`". The line contains at most 120 characters and does not contain any extra spaces.

## Output

Write to the output file a line for each variable declared in the input file. For each variable write its declaration on a single line in the same format as in the input file, but with all type constructors next to its type. Separate type with all type constructors from a variable name by a single space. Do not write any extra spaces.

## Sample input and output

| isharp.in | isharp.out |
|---|---|
| `int& a*[]&, b, c*;` | `int&&[]* a;`<br>`int& b;`<br>`int&* c;` |
| `Double[][] Array[];` | `Double[][][] Array;` |

# Problem J. Javanese Cryptoanalysis

| | |
|---|---|
| Input file: | `javanese.in` |
| Output file: | `javanese.out` |

Javanese is the language of the people in the Central and Eastern parts of the island of Java, Indonesia.

In 1926, a standard orthography using the English Alphabet was created for the Javanese language. This writing system uses all letters from A to Z. The five letters A, E, I, O, and U are vowels, while all other letters are consonants. In Javanese words vowels and consonants always alternate. This property is quite useful when deciphering encrypted Javanese texts.

A text $s$ consists of words, each word contains only capital letters. Let's call text $s$ *legitimate* if in each word of $s$ vowels and consonants alternate (no two vowels and no two consonants are located next to each other).

A *simple substitution cipher* is applied to a text $s$. That is, a bijection $f : A \to A$ is chosen, where $A$ is the set of capital letters. The encoded text $t$ is obtained from $s$ by substituting each letter $c$ with $f(c)$.

You're given the encoded text $t$. Find any legitimate text $s$ that can be encoded as $t$, or detect that there is no such legitimate $s$.

## Input

The input file contains the encoded text $t$, a list of words separated by spaces and/or line breaks. Each word consists only of capital letters (A to Z).

The input file contains no more than 100 000 characters.

## Output

If the text $t$ cannot be an encoded legitimate text, output only one word `impossible`.

Otherwise, output any legitimate text $s$ that can be encoded into $t$. Separate words of $s$ with spaces and/or line breaks. All letters in $s$ should be capital.

## Sample input and output

| javanese.in | javanese.out |
|---|---|
| `O RISK LIP FOCUS LUCKY` | `A CODE FOR VALID FILES` |
| `NEERC` | `impossible` |

# Problem K. KINA Is Not Abbreviation

| | |
|---|---|
| Input file: | `kina.in` |
| Output file: | `kina.out` |

When introducing new terms consisting of several words, it is useful to use abbreviations. An *abbreviation* is a word that consists of the first letters of several consecutive words.

An abbreviation is called *unambiguous* if the following two conditions are satisfied:

- It corresponds to exactly one sequence of words in a given text (although this sequence can appear in the text more than once);
- It does not appear in the text by itself.

For example, in the text "`A recursive acronym KINA means "KINA is not abbreviation"`", strings "`ARA`" and "`K`" are unambiguous abbreviations, strings "`A`" and "`KINA`" are ambiguous abbreviations, and strings "`RAA`" and "`KNA`" are not abbreviations.

To introduce an abbreviation in a text, it is placed in parentheses right after the sequence of words it corresponds to. Future occurrences of this sequence of words can be replaced by the abbreviation. In the example text above, introduction of the abbreviation "`K`" produces the following text: "`A recursive acronym KINA (K) means "K is not abbreviation"`".

If two occurrences of a sequence of words overlap, only one of them can be replaced by the abbreviation. Words in a sequence are separated by one or more non-alphabetic characters. Comparison of words is case-insensitive. For example, "`i18n`" is an occurrence of the word sequence "`I n`".

The *effectiveness* of an abbreviation is the decrease in the number of letters after introduction of this abbreviation. Only letters are taken into account; spaces, parentheses and all other non-alphabetical characters do not count.

Given a text, find an unambiguous abbreviation with the maximum effectiveness.

## Input

The input file contains a text with at most 4 000 characters. The text contains only characters with ASCII codes 32 (space) to 126 ("~"), 13 (carriage return), and 10 (line feed).

## Output

If there is no unambiguous abbreviation with positive effectiveness, then the output file should contain the single number 0.

Otherwise, the first line of the output file should contain the effectiveness of the optimal abbreviation. The second line should contain the unambiguous abbreviation itself. If there are multiple unambiguous abbreviations with the maximum effectiveness, output any one of them.

## Sample input and output

| kina.in | kina.out |
|---|---|
| `This problem name is "KINA is not abbreviation".`<br>`Once again: KINA is not abbreviation.` | `11`<br>`NA` |
| `To be or not to be: that is the question.` | `0` |
| `Here is the chorus of the song "Jingle Bells":`<br>`Jingle bells, jingle bells,`<br>`Jingle all the way;`<br>`Oh what fun it is to ride`<br>`In a one-horse open sleigh.` | `16`<br>`JB` |

In the first example, the optimal abbreviations are "`NA`" and "`INA`".
In the third example, the optimal abbreviations are "`JB`" and "`BJ`".