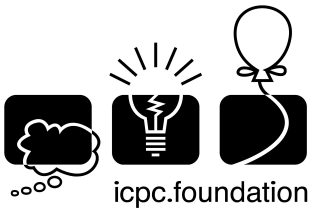


icpc International Collegiate Programming Contest

XXXIX Maratón Nacional de Programación
Colombia - ACIS / REDIS / CCPL 2025
ICPC

Problems

(This set contains 10 problems; problem pages are numbered from 1 to 17)

General Information. Unless otherwise stated, the conditions stated below hold for all the problems. However, since some problems may have specific requirements, it is important to read the problem statements carefully.

Program name. Each source file (your solution!) must be called

`<codename>.c`, `<codename>.cpp`, `<codename>.java`, or `<codename>.py`

as instructed below each problem title.

Input.

1. The input must be read from the standard input.
2. In most problems, the input can contain several test cases. Each test case is described using a number of lines specific to the problem.
3. In most cases, when a line of input contains several values, they are separated by single blanks. No other spaces appear in the input and there are no empty lines.
4. Every line, including the last one, has the usual end-of-line mark.
5. If no end condition is given, then the end of input is indicated by the end of the input stream. There is no extra data after the test cases in the input.

Output.

1. The output must be written to the standard output.
2. The result of each test case must appear in the output using a number of lines, which depends on the problem.
3. When a line of results contains several values, they must be separated by single spaces. No other spaces should appear in the output. There should be no empty lines.
4. Every line, including the last one, must have the usual end-of-line mark.
5. After the output of all test cases, no extra data must be written to the output.
6. To output real numbers, if no particular instructions are given, round them to the closest rational with the required number of digits after the decimal point. Ties are resolved rounding to the nearest upper value.

A: Account Qualifying

Source file name: `accountq.c`, `accountq.cpp`, `accountq.java`, *or* `accountq.py`
Author: Rodrigo Cardoso

The Bank of Linearonia (BoL) is rolling out a qualification process for customer accounts. For each account, the bank will compute a set of indices to rate its behavior over a given period, e.g., maximum, average, and minimum values, as well as the number and types of transactions.

Suppose an account X has N transactions $X[0], X[1], \ldots, X[N-1]$. A *transaction* is an integer: a *deposit* if it is positive, a *balance inquiry* if it is zero, and a *withdrawal* if it is negative. BoL wishes to compute the following indices:

d**:** the *maximum deposit*;

w**:** the *maximum withdrawal*; and

r**:** the *longest paired subperiod*, understood as the length of the longest subperiod that contains as many deposits as withdrawals.

If there are not deposits, the index d is 0. If there are not withdrawals, the index w is 0. However, if there are withdrawals, w is the most negative of them.

For instance, consider an account with transactions $23, -12, 0, 15, 0, 5$. In this case, the indices are:

$$d = 23, \qquad w = -12, \qquad r = 4.$$

The longest paired subperiod for this example is $r = 4$ because the subperiod $X[1..4]$ has length 4 and contains the same number of deposits and withdrawals (zeros do not affect the counts). No other subperiod has length greater than 4 with equal numbers of deposits and withdrawals. Note that care is required: for the sequence $(-3,\ 0)$ the longest paired subperiod is $r = 1$, since the subperiod consisting of the single element 0 has 0 deposits and 0 withdrawals.

Your task is to write a program that assists BoL in the account qualification process.

Input

The input consists of multiple test cases. Each test case begins with a line containing an integer N ($0 < N \leq 10\,000$), the length of the period. The next line contains exactly N integers $X[0], X[1], \ldots, X[N-1]$ separated by single spaces, where $-10\,000 \leq X[i] \leq 10\,000$ corresponding to a sequence of transactions X. A line containing a single `0` marks the end of input and is not a test case.

The input must be read from standard input.

Output

For each test case, output one line with the three blank-separated integers d, w, and r corresponding to X.

The output must be written to standard output.

Sample Input	**Sample Output**
6 23 -12 0 15 0 5 2 -3 0 3 100 200 300 3 0 0 0 0	23 -12 4 0 -3 1 300 0 0 0 0 3

B: Binary Dozens

Source file name: `bindozens.c`, `bindozens.cpp`, `bindozens.java`, *or* `bindozens.py`
Author: Rodrigo Cardoso

Egan recently founded a nano-egg company (microscopic eggs with promising pharmaceutical uses). Because eggs are typically sold by the dozen, Egan also developed *nano-crates*, each holding exactly 12 nano-eggs. The company sells only full dozens: an integer number of nano-crates, each completely filled.

Egg production is costly. Given a stock of H eggs, let D be the number of eggs that do not fit into complete nano-crates (12 eggs each). These surplus eggs must be discarded.

The following examples show some possible cases:

- If $H = 33$, two crates are used, discarding $D = 9$ eggs.
- If $H = 48$, four crates are used, discarding $D = 0$ eggs.

Production volumes can be very large and the line uses a binary counter to record batch size. Thus, H in decimal can be expressed in *binary* as B. Then, the previous examples should be interpreted as follows:

- If $B = 100001$, then $D = 9$.
- If $B = 110000$, then $D = 0$.

Help Egan by writing a program that reads B and outputs D in decimal.

Input

The input consists of multiple test cases, one per line. Each test case is a binary string B (characters '0' and '1' only) representing the number of nano-eggs H. You may assume $1 \leq |B| \leq 500$.

The end of the input is signaled with a line containing a single '*'.

The input must be read from standard input.

Output

For each test case B, output the number D of leftover nano-eggs when packing in nano-crates.

The output must be written to standard output.

Sample Input	Sample Output
100001 110000 1 111 1010110 *	9 0 1 7 2

C: Celestial Veins

Source file name: `celestial.c`, `celestial.cpp`, `celestial.java`, *or* `celestial.py`
Author: David Yepes

Eons ago, in a world bathed in perpetual night, a civilization of astronomers and poets --the Luminari-- flourished. Without instruments to plumb the depths of the cosmos, their understanding of the universe rested on what the eye could trace across the firmament. Thus they mapped the heavens as they appeared: a great celestial canvas projected onto a two-dimensional Cartesian plane. All their sacred theories, and every Euclidean distance they computed, were grounded in this planar projection.

They held that stars forged bonds of proximity and reciprocity, governed by a single rule: the *Law of Shared Skies*. According to this law, two distinct stars A and B form a true *bond* iff each regards the other as a close companion. Concretely, each star must lie within the other's *trusted circle*, defined as its K nearest distinct neighbors (by Euclidean distance in the planar projection). Ties at equal distance are broken by the *chronicle index*: the star that appears earlier in the ancient charts (input order) takes precedence. Any relationship that is not mutual is dismissed as a fleeting alignment, not a bond.

For the Luminari, a *constellation* is any group of at least two stars in which every pair is linked by an unbroken chain of bonds (i.e., a connected component of the mutual-bond graph with size greater than one). The pattern of bonds determines the constellation's essence, and each constellation is classified into exactly one of the following five sacred forms, evaluated in this order of priority:

Stellar Cluster: A group of stars where every star is bonded to all others, thus forming the maximum possible number of bonds for its size.

Cosmic Ouroboros: A closed ring of stars where the number of bonds is exactly equal to the number of stars.

Dragon's Claw: A formation where a single central star acts as a nexus, holding together the rest of the stars in the group, which possess no bonds among themselves.

Traveler's Path: A simple chain of stars that extends between two distinct endpoints, featuring no cycles or branches.

Amorphous Nebula: Any other formation of one or more interconnected stars that does not fit the patterns above. Additionally, any star that remains completely isolated is also counted in this category.

In a forgotten chronicle, the Luminari describe a celestial region they called ‘‘The Ring and the Swarm.’’ With observation parameter $K = 2$, their map listed eight stars in the order recorded by the chronicles.

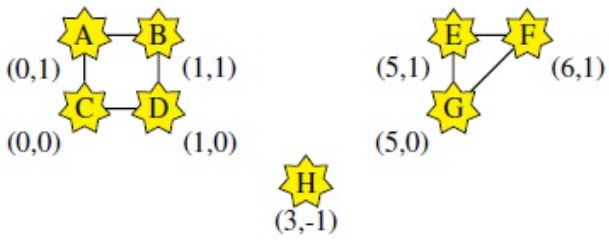


Figure 1: The Ring and the Swarm with adjusted labels

Their analysis, following the sacred classification rules, revealed three formations: one Cosmic Ouroboros (the Ring formed by stars A, B, C, and D), one Stellar Cluster (the Swarm formed by E, F, and G), and

one Amorphous Nebula (the isolated star H). No formations matched the Traveler's Path or Dragon's Claw patterns.

Your task is to apply the Luminari's ancient laws to each star map and report how many formations of each sacred type are present.

Input

The input consists of multiple star maps. Each map begins with a line containing two integers N ($1 \leq N \leq 1\,500$) and K ($1 \leq K \leq 5$). Then follow N lines; each line contains two integers X and Y ($-10^9 \leq X, Y \leq 10^9$) giving the coordinates of a star. The order of appearance defines the chronicle index: the first star listed has the lowest index, the second the next, and so on. The input terminates with a case where $N = 0$ and $K = 0$.

The input must be read from standard input.

Output

For each map, print one line with five space-separated integers: the counts of Traveler's Path, Cosmic Ouroboros, Stellar Cluster, Dragon's Claw, and Amorphous Nebula, in this exact order.

The output must be written to standard output.

Sample Input	**Sample Output**
8 2 0 1 1 1 0 0 1 0 5 1 6 1 5 0 3 -1 7 2 0 0 1 0 0 1 100 0 101 0 102 0 103 0 0 0	0 1 1 0 1 1 0 1 0 0

D: Discrete Catalog

Source file name: `discrete.c`, `discrete.cpp`, `discrete.java`, *or* `discrete.py`
Author: Camilo Rocha

In a certain country, natural disasters strike often enough that the authorities must constantly evaluate their preparedness. Earthquakes, floods, and landslides regularly disrupt local communities by isolating sites and cutting off access to food, water, and other vital supplies. To plan for these crises, the national crisis center runs simulations on a map of the region.

The region under study has n sites labeled $1, 2, \ldots, n$. A *scenario* describes which sites are disrupted in a disaster. Formally, a (n, k)-scenario is a tuple $[a_1, a_2, \ldots, a_k]$ of pairwise distinct integers in the range $1..n$, written in *strictly increasing order*, i.e., $1 \leq a_1 < a_2 < \cdots < a_k \leq n$. This ensures that each group of k sites is represented exactly once in the catalog.

To manage these simulations, the crisis center maintains a *Crisis Catalog* that assigns each possible (n, k)-scenario a unique *ID*. All scenarios of size k are listed in *lexicographic order* and numbered starting from 0. In lexicographic order, one scenario comes before another if, at the first position where they differ, the site number in the first scenario is smaller. This is the same way words are ordered in a dictionary, but applied to integer sequences instead of letters.

For example, there are 10 different $(5, 3)$-scenarios, with IDs in the catalog as shown next:

$(5, 3)$-scenarios	ID
$[1, 2, 3]$	0
$[1, 2, 4]$	1
$[1, 2, 5]$	2
$[1, 3, 4]$	3
$[1, 3, 5]$	4
$[1, 4, 5]$	5
$[2, 3, 4]$	6
$[2, 3, 5]$	7
$[2, 4, 5]$	8
$[3, 4, 5]$	9

Given n, k, and a specific (n, k)-scenario, your task is to determine its ID in the Crisis Catalog.

Input

The input consists of two lines. The first line contains two blank-separated integers n and k, with $1 \leq n \leq 60$ and $1 \leq k \leq n$, indicating the number of sites and the size of the scenarios. The next line contains a list of k blank-separated integers $a_1, a_2, \ldots, a_k$ satisfying $1 \leq a_1 < a_2 < \cdots < a_k \leq n$.

The input must be read from standard input.

Output

For each test case, output a single line with the ID of the (n, k)-scenario $a_1, a_2, \ldots, a_k$.

The output must be written to standard output.

Sample Input	**Sample Output**
5 3 1 2 3 5 3 1 3 5 5 3 2 3 5 5 3 3 4 5 3 2 1 2 3 2 1 3 3 2 2 3 32 6 4 8 16 17 25 31 60 10 31 32 33 34 35 36 37 38 39 40	0 4 7 9 0 1 2 484650 75363982551

E: Efficient Encoding

Source file name: `encoding.c`, `encoding.cpp`, `encoding.java`, *or* `encoding.py`
Author: Rafael García

It is the year 3057. The Interplanetary Compression Priority Code (ICPC) is tasked with maintaining communications among human colonies scattered across the galaxy. To survive in a hostile environment with severely limited bandwidth, every message must be compressed as efficiently as possible before transmission.

Each message is a multiset of interplanetary symbols. The ICPC compresses messages written with an alphabet of n symbols, assigning to each symbol a codeword constructed using a binary prefix-free code (i.e., no codeword is a prefix of another). As usual, the ICPC relies on optimal coding strategies that minimize the total number of bits required-just as the legendary compression schemes of the 21st century once did.

However, the number of occurrences of each symbol is not constant over time (the ICPC actually works with their expected values). They fluctuate due to solar storms, the rotation of the quantum core, and other cosmic phenomena that only astrophysicists pretend to understand. To cope with this variability, between time 0 and a fixed time upper bound T, the ICPC models the expected value of the number of occurrences of each symbol as a piecewise-linear function specified by three points with coordinates in the non-negative real numbers (i.e., in $\mathbb{R}_{\geq 0}$): $(0, a_i)$, (m_i, b_i), and (T, c_i), with $0 < m_i < T$, and a_i, b_i, and c_i integers. Between these points, the expected value for the number of occurrences of i th symbol, $f_i(t)$, varies linearly over real values. At any time $t \in [0, T]$, the expected value of the number of occurrences of symbol i is:

$$f_i(t) = \begin{cases} a_i + \dfrac{b_i - a_i}{m_i} \cdot t, & 0 \leq t \leq m_i, \\ \\ b_i + \dfrac{c_i - b_i}{T - m_i} \cdot (t - m_i), & m_i \leq t \leq T. \end{cases}$$

To choose the best encoding time, the ICPC considers, for each $t \in [0, T]$, the optimal binary prefix-free code for the frequencies $\{f_i(t)\}_{i=1}^{n}$ and evaluates its total cost

$$C(t) \;=\; \sum_{i=1}^{n} f_i(t)\, \ell_i(t),$$

where $\ell_i(t)$ is the codeword length assigned to symbol i by an optimal code at time t.

Your task is to help the ICPC choose the ideal encoding time: find the minimum time $t \in [0, T]$ that minimizes the total encoding cost $C(t)$.

Input

The first line contains the number of test cases K ($1 \leq K \leq 10^3$). For each test case, the first line contains two integers n and T ($1 \leq n \leq 100$ and $1 < T \leq 10^6$), denoting the number of symbols and the maximum time value, respectively. Each of the next n lines contains four integers a_i, m_i, b_i, and c_i ($0 < m_i < T$ and $0 \leq a_i, b_i, c_i \leq 10^6$), describing the frecuency function $f_i(t)$.

The input must be read from standard input.

Output

For each test case, output two blank-separated numbers on one line: the smallest integer time $t \in [0, T]$ that minimizes $C(t)$, followed by the minimum cost $C(t)$ rounded to three digits after the decimal point.

The output must be written to standard output.

Sample Input	**Sample Output**
2 3 10 2 5 10 4 4 6 9 3 3 8 8 7 1 2 2 1 1 2	0 14.000 1 1.000

F: Fingerprints

Source file name: `fingerprints.c`, `fingerprints.cpp`, `fingerprints.java`, *or* `fingerprints.py`
Author: Camilo Rocha

A forensic database stores fingerprints encoded as strings of characters. These encodings are stored *circularly* so that the starting position in the string is arbitrary. For example, the encodings `ABCD`, `BCDA`, `CDAB`, and `DABC` all represent the same fingerprint. However, `DABC` and `ABDC` represent different fingerprints.

Lately, the database is running out of cloud storage space---apparently most of it is being consumed by the ever-growing needs of modern GPTs. To clean up the storage, the forensic team wants to keep only one copy of each distinct fingerprint. To do so, each group of equivalent encodings will be represented by is *canonical encoding*: the lexicographically smallest rotation.

Given a list of fingerprint encodings, output the set of distinct fingerprints that remain after deduplication, each represented by its canonical encoding.

Input

The input consists of several test cases, each being a line containing a number N ($1 \leq N \leq 100$) of fingerprints. Then, N lines follow, each containing a blank-separated pair M, F, where M is the length of the fingerprint F ($M = |F|$ and $1 \leq M \leq 5\,000$). Each fingerprint consists of uppercase English characters only. The end of input is signaled with a line containing a single 0.

The input must be read from standard input.

Output

For each test case, output the canonical encodings of all distinct fingerprints, one per line, in the order in which they can be found in the input.

The output must be written to standard output.

Sample Input	**Sample Output**
5 4 ABCD 4 BCDA 3 XYZ 4 FFFF 3 ZXY 0	ABCD XYZ FFFF

G: Guard Deployment

Source file name: `guard.c`, `guard.cpp`, `guard.java`, *or* `guard.py`
Author: Rafael García

The Nlogonian Army military base is located in a remote and strategic area. The base is composed of a collection of rectangular buildings with flat roofs. The buildings are arranged in a way that creates an irregularly shaped base. Each building is defined by its base (a rectangle in the 2D plane) and its height (all in meters).

To increase the security of the base, the commander has decided to build a convex polygonal perimeter fence with *guard towers* at each vertex and surveillance cameras on their tops. The commander wants to minimize the length of the fence to save resources.

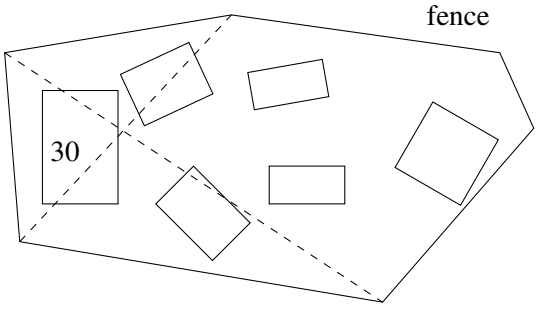
Additionally, the guard towers must have a line of sight to enable swift communication between the guards in an emergency. A line of sight between two guard towers is ensured if no part of the line intersects or touches the surface or edge of any building, including its roof. Since unexpected objects may be found on the roof of some buildings, if a line of sight passes above a building, it must do so at least one meter higher than the building's roof.

To facilitate communication, surveillance and manufacture, the commander also wants that all the designed guard towers has the same height. Furthermore, he wants this height to be minimal to avoid making the guard towers too visible and to save resources.

The commander has asked his team of engineers to determine the optimal lenght of the perimeter fence and the minimum height of the guard towers that guarantees a line of sight between them.

Can you help the team of engineers solving this problem?

The following figure shows the layout of a military base with six rectangular buildings. The building on the left is 30 meters high, and assume all the others are 10 meters high. Also shown is a polygonal perimeter fence (not necessarily optimal) surrounding the base. The two dotted lines show two lines of sight, justifying the placement of the cameras at a height of 31 meters.



Input

The first line contains a number K ($1 \leq K \leq 10^4$), the number of test cases.

For each case, the first line contains a natural number N ($1 \leq N \leq 10^3$) representing the number of buildings in the base. Each of the following N lines containins nine integers: $x_1, y_1, x_2, y_2, x_3, y_3, x_4, y_4, h$. The points (x_1, y_1), (x_2, y_2), (x_3, y_3) and (x_4, y_4) are the coordinates of the corners of the rectangular building's base ($0 \leq |x_1|, |y_1|, |x_2|, |y_2|, |x_3|, |y_3|, |x_4|, |y_4| \leq 10^6$), and, h ($1 \leq h \leq 10^3$) is the height of the building.

It is guaranteed that the input data correspond to rectangular bases and that no two of them overlap. More exactly, for each pair pairs of buildings, the intersection of their bases measures 0.

The input must be read from standard input.

Output

A single line containing two numbers separated by a single space: the optimal perimeter of the fence (a floating-point number with 6 decimal places) and the minimum height of the guard towers that ensures visibility between all pairs of towers.

The minimum height at which a watchtower must be located is 3 meters.

The output must be written to standard output.

Sample Input	**Sample Output**
1 5 3 0 5 2 2 5 0 3 10 2 6 5 6 5 8 2 8 5 9 8 8 9 7 8 8 7 30 6 5 8 5 8 6 6 6 31 6 3 8 3 8 4 6 4 20	28.054753 32

H: Holy Network

Source file name: `holy.c`, `holy.cpp`, `holy.java`, *or* `holy.py`
Author: David Yepes

Once a century, when the stars align in a sacred conjunction, the world's greatest alchemists leave their secret laboratories to gather at the Arcane Conclave. Their purpose is to perform the Great Concordance Ritual, an act of magic so immense it can rewrite the laws of nature.

Each master brings a unique ingredient, distilled over a century of work. Through arcane numerology and the study of the vibrational patterns of matter, each alchemist has managed to assign their ingredient an integer non-negative number that defines its essence: the *Harmonic Resonance*.

The ritual, however, is dangerously unstable. For the power to manifest safely, the group of alchemists channeling it must be in perfect harmony. The rule is strict and ancient: a group is harmonic only if the Resonance of every pair of ingredients within the group is compatible. Two ingredients are *compatible* if their numbers descend from the same numerical root (i.e., they share a factor greater than 1).

Forming the largest possible group is crucial, as the ritual's power increases exponentially with each participant. As the new alchemist and expert in numerology for the Conclave, you have been entrusted with the crucial task of analyzing all the ingredients and determining the maximum size of the harmonic group that can be formed, thus ensuring the ritual's success.

The chronicles of the Conclave of the Faded Star, the last to be held, detail a particularly complex challenge. Eight masters presented ingredients with Harmonic Resonances of **30, 42, 70, 105, 11, 13, 17 and 19**. The chronicles state that after a long deliberation, it was discovered that the largest possible harmonic group was of size 4, formed by the resonances **30, 42, 70 and 105**. An incomplete verification of that, as it was recorded, was as follows:

- The pair (30, 42) is compatible, as they descend from the numerical root 6.
- The pair (42, 70) is compatible, as they descend from the root 14.
- The pair (105, 42) is compatible, as they descend from the root 21.
- ...

This discovery was a milestone in the Conclave's history, as it proved that the addition of the other prime ingredients could not expand the core harmonic group, and that true harmony resided in a complex web of shared roots.

Input

The input consists of multiple test cases. Each test case begins with a line containing a single integer N ($1 \leq N \leq 50$), the number of ingredients. The next line contains N space-separated integers $p_1, p_2, ..., p_n$ ($1 \leq p_i \leq 10^{12}$), which are the Harmonic Resonances. The input terminates when a line with $N = 0$ is encountered, which should not be processed as a case.

The input must be read from standard input.

Output

For each test case, print a single line with a single integer: the size of the largest harmonic group that can be formed with the given Harmonic Resonances.

The output must be written to standard output.

Sample Input	**Sample Output**
8 30 42 70 105 11 13 17 19 5 30 42 70 105 154 5 2 3 5 7 11 0	4 5 1

I: Impossible Primebox

Source file name: `impossible.c`, `impossible.cpp`, `impossible.java`, *or* `impossible.py`
Author: Julián Badillo

> ''I got a challenge for you'', says Danny O'sea - famous career swindler and heistwoman, ''I need you to open this very secure safe, called the *Primebox*. I promise you, the pay is worth it.''

The rumor around town is that Danny O'sea is running the most awesome heist of her time: To steal the Ultra Luxurious McGuffin, currently displayed in the Extremely Securest Room of the Very Secure Gallery, and on top of that, locked inside the Primebox.

> ''I have acquired, at a great cost, crucial information about how the Primebox works'', she tells you.

The *Primebox* is composed of L *primelocks*, numbered $0, 1, \ldots, L-1$. The primelock i is a piece of programming code of the form:

```
prlck i:
x = x[ * A_i][ + B_i]
if P_i div x jumpto {prlck j_i | end} else jumpto {prlck m_i | end}
```

where A_i and B_i are positive integers, and P_i is a prime number ($0 \le i, j_i, m_i < L$). The expressions in square brackets are optional. For expressions in curly brackets separated by the character ''|'', exactly one of them is to be chosen. The notation s `div` t means ''*s divides t*''.

The primelock i has an *activation number* $K_i \ge 0$, whose meaning will become clear next. The locks are arranged sequentially and are activated (i.e., executed) starting from primelock 0 and an initial integer value of $x \ge 0$. The execution of a primelock is accomplished in a 'natural way' (please, do not ask what 'natural' means ...) and the whole process ends when any of the `jumpto end` instructions are executed.

To open the whole Primebox, the i-th primelock must be *activated* exactly K_i times. There might be several initial numbers that may activate the primelocks according to their corresponding activation numbers. The *key* to the Primebox is the minimum of them.

> ''Ah! Don't worry about it, I got one more piece of intelligence'', says Danny O'Sea, ''The key does exist and it's lower than 50 000''.

Your job is, given the primelocks' source code and their respective activation numbers, to crack the key. A juicy reward is waiting for you.

For instance, consider a Primebox defined with the following primelocks:

```
prlck 0:
x = x * 2 + 3
if 3 div x jumpto prlck 2 else jumpto prlck 1
prlck 1:
x = x * 4 + 1
if 5 div x jumpto prlck 0 else jumpto prlck 2
prlck 2:
x = x + 5
if 7 div x jumpto prlck 1 else jumpto end
```

and activation numbers $K_0 = 1$, $K_1 = 2$, and $K_2 = 2$. You may check that, starting with $x = 10$, the process will end after the primelocks are activated, as requested. However, this is not the case with any integer less than 10. Therefore, this Primebox is opened with the key 10.

Input

The input consists of several test cases. Notice that all variables, integers, and operators described here are separated by single blanks. Each test case defines a Primebox and starts with an integer L ($1 \leq L \leq 10$), the number of primelocks. The following $3 \times L$ lines describe the primelocks in sequential order, each one following the format described above. You may suppose that, for primelock i ($0 \leq i < L$), the involved parameters A_i, B_i, and P_i are positive integers ($0 \leq A_i, B_i, P_i \leq 1\,000$), P_i is a prime number, and every referenced primelock number is in the integer interval $0 \,..\, L-1$. The case ends with a line containing L numbers K_i ($0 \leq i < L$ and $0 \leq K_i \leq 1\,000$), the corresponding activation numbers for the given primelocks. The input ends with the line `0`, which should not be processed.

The input must be read from standard input.

Output

For each test case, output a single line with the corresponding cracked key.

The output must be written to standard output.

Sample Input	Sample Output
3 prlck 0: x = x * 2 + 3 if 3 div x jumpto prlck 2 else jumpto prlck 1 prlck 1: x = x * 4 + 1 if 5 div x jumpto prlck 0 else jumpto prlck 2 prlck 2: x = x + 5 if 7 div x jumpto prlck 1 else jumpto end 1 2 2 3 prlck 0: x = x + 3 if 71 div x jumpto end else jumpto prlck 1 prlck 1: x = x * 15 if 131 div x jumpto end else jumpto prlck 2 prlck 2: x = x * 24 + 1 if 3 div x jumpto prlck 1 else jumpto prlck 0 7 7 6 0	10 255

J: Just Palindromes!

Source file name: `just.c`, `just.cpp`, `just.java`, *or* `just.py`
Author: Rodrigo Cardoso

Ana is fascinated by the idea of finding *palindromes*, i.e., sentences that read the same backward and forward once you ignore spaces, punctuation, letter case, and letter characters outside the English alphabet. For example, the following four sentences are palindromic:

```
Go deliver a dare vile dog.

Cigar? Toss it in a can. It is so tragic.

Never odd or even!

A man, a plan, a canal: Panama!
```

Note that in a palindrome, only letter characters from the English alphabet are allowed, case does not matter, and all non-letter characters (punctuation, spaces, apostrophes, etc.) are ignored.

On the other hand, the following sentences are not palindromes, even if case and non-letter characters are ignored:

```
This is a palindrome.

Hello, world!
```

Help Ana by writing a program that reads a sentence and reports whether it is a palindrome.

Input

The input consists of multiple test cases. Each test case is a single line containing a sentence S of length N ($0 \leq N < 1\,000$) and made only of printable ASCII characters (i.e., characters with ASCII code between 32 and 126, inclusive). The input ends with a line containing a single asterisk '*', which must not be processed.

The input must be read from standard input.

Output

For each test case, output a single line with the letter character 'Y' if S is a palindrome and 'N' if not.

The output must be written to standard output.

Sample Input	Sample Output

`Go deliver a dare vile dog.`
`Cigar? Toss it in a can. It is so tragic.`
`A man, a plan, a canal: Panama!`
`This is a palindrome.`
`.---.-`
`*`
	`Y`
	`Y`
	`Y`
	`N`
	`Y`