A – Accountant notes

You had an accountant, who recorded all your expenses. Each day you sent her notes on expenses and she appended them to a single big summary file. Each of your notes consisted of several rows, like the following:

```
bed = 100
table = 150
furniture = bed + table
furniture = furniture + 10
```

which (probably) meant that you spent 250 on furniture and 10 on transport (probably). More generally, each row in your note was one of the following:

```
name = number
name = item + item
```

where `item` was either `number` or `name`, `number` was a natural number not exceeding 10^9 (with no leading zeroes) and `name` was an arbitrary sequence of small and capital letters.

While transferring your notes to the summary file, the accountant neither changed the order of rows of a note nor shuffled rows of different notes. But sometimes she changed names... She did it consistently though, changing all occurrences of the name in a single note to the same new one. Also, different names from the same note were rewritten as different names in the summary file. But there is no guarantee that it was consistent with your other notes — the name *beer* from one note could have been rewritten as *drink*, the name *tee* from the other note could have been rewritten as *drink* as well, while the *beer* from the next note could have been rewritten as *food*. Lets call her version of your note *a transcription* of it.

The problem is that your accountant has just quit (and, as it turned out, took most of your savings) and you are left with the summary file and a huge unsorted pile of notes. You want to check if it is possible that some of your notes were not recorded in the summary, so for each note you are searching for a piece of summary that can be its transcription.

Multiple Test Cases

The input contains several test cases. The first line of the input contains a positive integer $Z \leq 15$, denoting the number of test cases. Then Z test cases follow, each conforming to the format described in section *Single Instance Input*. For each test case, your program has to write an output conforming to the format described in section *Single Instance Output*.

Single Instance Input

The first line of an input instance contains an integer k, the number of notes ($1 \leq k \leq 50000$). The following lines first contain the descriptions of k notes, then the description of the summary file follows. The description of each note and the summary starts with a line consisting of a single integer d ($1 \leq d$). Then d lines follow, each containing a single row of a note or the summary. The format of each row is as described in the problem statement above. You can assume that names, numbers, signs "=" and "+" are separated by single spaces and each line is at most 100 characters long. Both the length of the text and the sum of all patterns lengths will be at most $3 * 10^6$.

Single Instance Output

The output for a single instance should consist of k lines. In the i-th line there should be the answer for the i-th note. It should be the number of *the first row* in the summary file, where a possible transcription of the i-th note starts (the first row of the summary file has number 1) or word `NONE` if no fragment of the summary file could be such possible transcription.

Example

Input	Output
2 3 4 bed = 100 table = 150 furniture = bed + table furniture = furniture + 10 1 beer = 100 2 a = a + a a = a + 10 10 furniture = 100 table = 150 furniture = furniture + table furniture = furniture + 10 sofa = 100 stol = 150 meble = sofa + stol meble = meble + 10 picie = 100 drink = 0100 2 1 a = 100 + 200 1 a = 100 2 x = 100 + 200 y = 100	5 1 NONE 1 2

B — Better and faster!

You probably know this story already. You wake up in the morning and your head feels twice the size. You have a vague memory of a program your boss asked you to write. After you have logged in, you see a main piece of code you wrote yesterday.

```
unsigned int checksum (char str[], int len) {
    unsigned int r = 0;
    for (int k=0; k<8*len; k++) {               // iterate over bits of str
        if ((r & (1<<31)) != 0) r = (r << 1) ^ 0x04c11db7;
                          else r = (r << 1);   // do some magic
        if (str[k/8] & 1<<(7-k%8))              // if the k-th bit of str is set,
            r ^= 1;                             //   then flip the last bit of r
    }
    return r;
}
```

"Good", you think, "I commented it well". Still, you have some issues with understanding the "do some magic" part. But well, the function is called `checksum`, and — lo and behold — it really computes a kind of a checksum of a given string.

You recall the rest of your task. You were supposed to compute this checksum for a given string and then for slightly modified versions of this string. Actually, the rest of your program also looks quite decent.

```
#include <stdio.h>

int main()
{
    char str[1000001],c;
    int TESTS,n,changes,p;
    for (scanf ("%d", &TESTS); TESTS>0; TESTS--) {
        scanf ("%d %s", &n, str);               // read the input
        printf ("%u\n", checksum(str, n));      // compute checksum for original string
        for (scanf ("%d", &changes); changes>0; changes--) {
            scanf ("%d %c", &p, &c);            // apply the change
            str[p-1] = c;
            printf ("%u\n", checksum(str, n));  // compute checksum for modified string
        }
    }
}
```

And then you recall the final issue. The program works perfectly well, but also terribly slow. You just have to make it work faster. Much faster. As you have heard that Java is a better and safer programming language, you even made an equivalent Java version (see the last page), which works even slower (strange, eh?).

Multiple Test Cases

The input contains several test cases. The first line of the input contains a positive integer $Z \leq 20$, denoting the number of test cases. Then Z test cases follow, each conforming to the format described in section *Single Instance Input*. For each test case, your program has to write an output conforming to the format described in section *Single Instance Output*.

Single Instance Input

Below by a *character*, we mean a single small or large letter, or a digit.

In the first line of an input instance, there is a natural number n ($1 \leq n \leq 10^6$) and a string s, separated by a single space. String s consists of n characters. The second line of the input contains

one integer t ($0 \leq t \leq 10^5$) denoting the number of changes to be applied to string s. Each of the next t lines consists of a natural number $p \in [1, n]$ and a character c, separated by a single space. It encodes a change of a string: the p-th character of s has to be replaced by c.

Single Instance Output

You have to produce the same output the program above would do. In other words, you have to output $t+1$ lines, each containing a natural number being a checksum. The first checksum has to be computed for an original string s, the remaining ones are to be computed after each change made to s.

Example

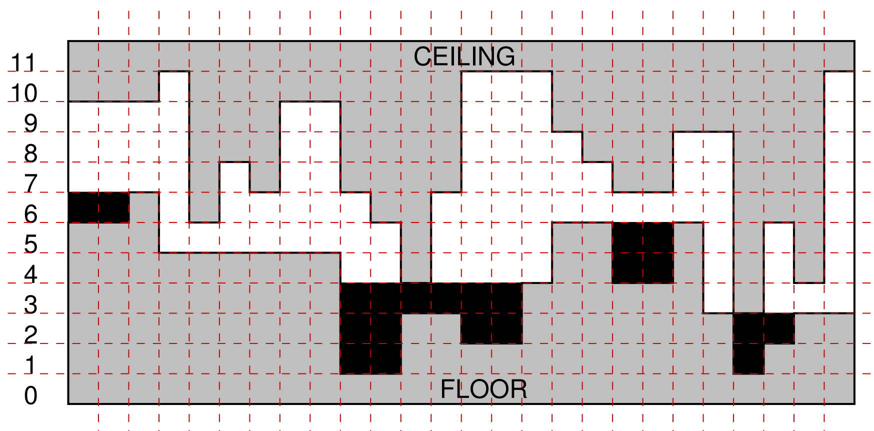
Input	Output
1 5 ABcd3 3 1 B 2 A 1 d	1914964467 2137468714 2087137066 4274181240

Java Version of The Program

```
import java.util.Scanner;

public class Compute {

static long checksum (byte[] str, int len) {
    int r = 0;
    for (int k=0; k<8*len; k++) {                    // iterate over bits of str
        if ((r & (1<<31)) != 0) r = (r << 1) ^ 0x04c11db7;
                           else r = (r << 1);        // do some magic
        if ((str[k/8] & 1<<(7-k%8)) != 0)            // if the k-th bit of str is set,
            r ^= 1;                                  //   then flip the last bit of r
    }
    long rr = (r<0 ? r+0x100000000L : r);            // Java does not have unsigned int
    return rr;
}

public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    for (int TESTS = in.nextInt(); TESTS>0; TESTS--) {
        int n = in.nextInt();                        // read the input
        byte[] str = in.next().getBytes();           // compute checksum for
        System.out.println (checksum(str,n));        //   original string
        for (int changes = in.nextInt(); changes>0; changes--) {
            int p = in.nextInt();                    // apply the change
            byte c = in.next().getBytes()[0];
            str[p-1] = c;
            System.out.println (checksum(str,n));    // compute checksum for
        }                                            //   modified string
    }
}
```

C — Cave

As an owner of a land with a cave you were delighted when you last heard that underground fuel tanks are great business. Of course, the more volume one can store, the better. In case of your cave, the effective volume is not easy to calculate, because the cave has a rather sophisticated shape (see figure). Thank heavens it is degenerate in one dimension!



The cave. All ponds that can be flooded with fuel are marked black.

Furthermore, there is some electrical wiring on the ceiling of the cave. You can never be sure if the insulation is intact, so you want to keep the fuel level just below the ceiling at every point. You can pump the fuel to whatever spots in the cave you choose, possibly creating several ponds. Bear in mind though that the fuel is a liquid, so it minimises its gravitational energy, e.g., it will run evenly in every direction on a flat horizontal surface, pour down whenever possible, obey the rule of communicating vessels, etc. As the cave is degenerate and you can make the space between the fuel level and the ceiling arbitrarily small, you actually want to calculate the maximum possible area of ponds that satisfy aforementioned rules.

Multiple Test Cases

The input contains several test cases. The first line of the input contains a positive integer $Z \leq 15$, denoting the number of test cases. Then Z test cases follow, each conforming to the format described in section *Single Instance Input*. For each test case, your program has to write an output conforming to the format described in section *Single Instance Output*.

Single Instance Input

In the first line of an input instance, there is an integer n ($1 \leq n \leq 10^6$) denoting the width of the cave. The second line of input consists of n integers $p_1, p_2, \ldots, p_n$ and the third line consists of n integers $s_1, s_2, \ldots, s_n$, separated by single spaces. The numbers p_i and s_i satisfy $0 \leq p_i < s_i \leq 1000$ and denote the floor and ceiling level at interval $[i, i+1)$, respectively.

Single Instance Output

Your program is to print out one integer: the maximum total area of admissible ponds in the cave.

Example

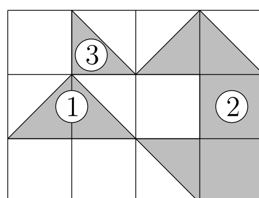
Input	Output
1 15 6 6 7 5 5 5 5 5 5 1 1 3 3 2 2 10 10 10 11 6 8 7 10 10 7 6 4 7 11 11	14

D — Decision

In a galaxy not so far away, in a time where men were real men, women were real women, and small furry creatures from Alpha Centauri were real small furry creatures from Alpha Centauri, an astronom called Mr. Gorsky discovered a small inhabited planet. After an initial enthusiasm (*yes, we are not alone!*), all the living Nobel Peace Prize winners gathered in one place, formed a committee, and discussed options of invading the planet. Long story short, in order to decide on this important topic, they need to know the number of the cities on the remote planet.

The quality of photos delivered by Mr. Gorsky were unfortunately quite bad: on a rectangular grid, each grid element was either blank (no city there) or (partially) dark, which meant a city or a part of it. If two dark parts share a common edge, they are a part of the same city. The committee then said plainly: "You have to count the number of the cities. Good luck, Mr. Gorsky".

An example map containing three cities is presented below.

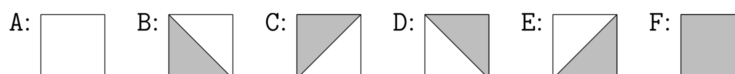


Multiple Test Cases

The input contains several test cases. The first line of the input contains a positive integer $Z \leq 20$, denoting the number of test cases. Then Z test cases follow, each conforming to the format described in section *Single Instance Input*. For each test case, your program has to write an output conforming to the format described in section *Single Instance Output*.

Single Instance Input

The first line of an input instance contains two integers n and m, the photo dimensions, such that $1 \leq n, m \leq 1000$. The following n lines contain the description of the photo. Each line contains m characters from the set $\{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}, \mathtt{E}, \mathtt{F}\}$ encoding the grid elements in the following way:



Single Instance Output

For each input instance, your program should output one line containing the number of cities on a given map.

Example

Input	Output
4 1 2 DD 2 2 FB DF 2 3 FAA AFB 4 4 AACB CAFD AFCE AACA	2 1 2 6

E — Everybody may get lost in space

As we all know, all essential systems of a space shuttle are "redundantly replicated" just in case. In the cold and empty space, the key question to a successful navigation is the centuries-old "were-may?" Luckily, in accordance to aforementioned rule, the shuttle's coordinates can be obtained from three independent sources. These systems provide not only x, y and z coordinates, but also the bound b on observational error. The error applies to the distance from point (x, y, z), meaning that whenever a system reports (x, y, z), the correct shuttle's coordinates might be any (x', y', z') with $\sqrt{(x-x')^2 + (y-y')^2 + (z-z')^2} \leq b$. Truth be told, it's not easy to determine the shuttle's position given as many as its three measures. Your task is to determine the volume of the (sub-)space in which the shuttle is possibly contained. At least one of the three systems is intact, but it might be the case that the others are broken.

Multiple Test Cases

The input contains several test cases. The first line of the input contains a positive integer $Z \leq 10000$, denoting the number of test cases. Then Z test cases follow, each conforming to the format described in section *Single Instance Input*. For each test case, your program has to write an output conforming to the format described in section *Single Instance Output*.

Single Instance Input

The input instance consists of three lines, each containing a single independent measure. Each measure consists of coordinates $x, y, z \in [-10^9, 10^9]$ and the observational error $b \in [1, 10^9]$ separated by single spaces.

Single Instance Output

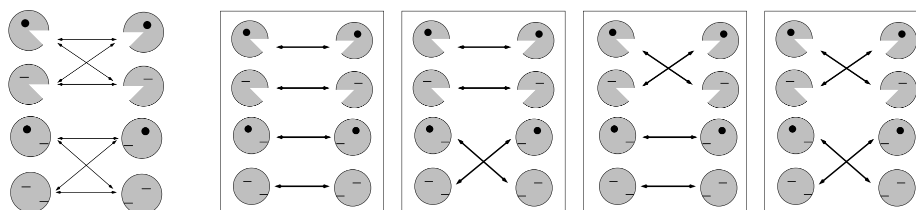
Your program is to print out the volume of the (sub-)space in which the shuttle is possibly contained. Your result is going to be accepted if and only if it is accurate to within a relative or absolute value of at most 10^{-6}.

Example

Input	Output
2 0 0 0 10 19 0 0 10 23 0 0 10 0 0 0 10 12 0 0 10 18 0 0 10	9602.0161463094 9334.7189713665

F – (False) faces

The Company is testing its brand new face recognition solution. The application is supposed to recognize people given their profiles. It is fed with a number of test cases to check if everything is working fine. Each test consists of $2n$ photos of n persons; there is a left profile and a right profile of each person in a single test case. The program matches left profiles with right profiles, but it is still far from perfect, so sometimes several right profiles are assigned to a single left profile (and vice versa). A *consistent reconstruction* is an assignment of *different* right profiles to *all* left profiles, such that all the pairs matched were proposed by the program.



The program output (on the left) and four possible consistent reconstructions.

In order to test the program, it is necessary to verify every possible consistent reconstruction. The verification has to be done by humans and the Company has a team of four experts who devoted their lives to face recognition. They are willing to do the job under one condition: their shares of work have to be equal, i.e., the number of consistent reconstructions has to be divisible by four. Your task is to check if it is so.

Multiple Test Cases

The input contains several test cases. The first line of the input contains a positive integer $Z \leq 100$, denoting the number of test cases. Then Z test cases follow, each conforming to the format described in section *Single Instance Input*. For each test case, your program has to write an output conforming to the format described in section *Single Instance Output*.

Single Instance Input

The first line contains the number n of persons in a test ($1 \leq n \leq 300$). Then n lines follows, each containing n characters, each of them being 0 or 1. The j-th character in the i-th line is 1 if and only if the program matches the i-th left profile with the j-th right profile.

Single Instance Output

The output should consist of one line containing `YES` if the number of reconstructions consistent with the program assignment is divisible by 4 and `NO` otherwise.

Example

Input	Output
2 4 1100 1100 0011 0011 3 111 011 001	YES NO

G — Garlands

Garland decorator is a profession which recently gained in importance, especially during Christmas time. Any kid can decorate a christmas tree, any parent can put gifts in sockets, and even anyone can start believing in Santa Claus, but hanging christmas garlands is a completely different story. As you will learn, it is an extremely important, responsible and tough job.

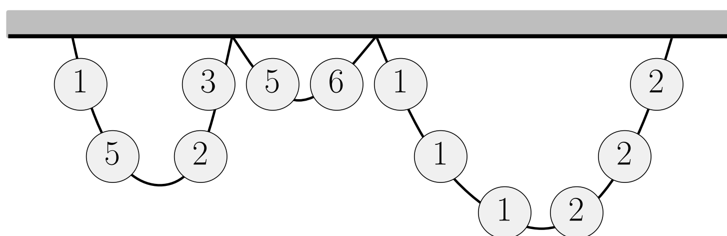
A garland consists of n pieces of equal length. Due to decorations like christmas balls attached to garlands, piece i has its own weight w_i. The garland has to be attached to the ceiling in m spots, where the very beginning of the garland should be attached to spot 1 and its end to spot m. The garland should also be hooked to the remaining spots, which divides it into segments, each consisting of several consecutive pieces. There are, however, several rules that every respectable garland decorator should keep in mind.

(i) Each segment should contain a positive even number of pieces. Due to this condition, we may divide a segment into two *half-segments*.

(ii) To minimize the chance that a guest hits your precious garland with their head (and tears it into pieces), the garland cannot hang too low: each half-segment can contain at most d pieces.

(iii) Finally, to keep the ceiling from falling on people heads, the decorator should minimize the weight of the heaviest half-segment.

An example of an optimally hanging garland (consisting of twelve pieces in three segments) is presented below; weights of respective pieces are given in circles.



Multiple Test Cases

The input contains several test cases. The first line of the input contains a positive integer $Z \leq 50$, denoting the number of test cases. Then Z test cases follow, each conforming to the format described in section *Single Instance Input*. For each test case, your program has to write an output conforming to the format described in section *Single Instance Output*.

Single Instance Input

The description of each garland consists of two lines. The first line describing a particular garland contains three positive integers n, m, and d ($1 \leq n \leq 40000$, $2 \leq m \leq 10000$, $1 \leq d \leq 10000$) separated by single spaces and described above. The second line contains n positive integers $w_1, w_2, \ldots, w_n$ ($1 \leq w_i \leq 10000$), being the weights of the corresponding pieces.

Single Instance Output

For each garland, your program should output a single line containing one integer, being the weight of the heaviest half-segment in an optimal attachment of the garland. If it is not possible to hang the garland satisfying conditions (i) and (ii), then your program should output word `BAD`.

Example

Input	Output
4 4 3 10 10 10 20 20 6 4 10 1 1 100 100 1 1 6 3 10 1 1 100 100 1 1 1 2 2 333	20 100 200 BAD

H. Hypervisor MacrOS

Bob is the leader of a team developing a hypervisor system *MacrOS*. The project is huge and comprises lots of packages. Since some of them may depend on others, the installation process of the whole system is rather complicated. MacrOS is in alpha stage now, so there are many customers testing the new product. The installation process is overwhelming for many of them, and unfortunately the technical support is Bob's responsibility as well.

Bob had been given a list of dependencies by his team of programmers before the alpha tests started. Each dependency reads "package B depends on A", i.e., one has to install package A before B. This is denoted `A B` in short. Of course, if package C depends on B, and B in turn depends on A, then C depends on A as well, i.e., the dependencies are transitive. For example, the initial list of dependencies can be as follows:

As the product develops, the programmers sometimes call Bob to inform of new dependencies. Further, being in charge of technical support, Bob often receives phone calls from customers with questions which packages should be installed first. To no surprise, after several calls from programmers and customers, Bob realized how difficult it is to keep track of dependencies and answer queries on-line at the same time, and. For automatizing this process, he wrote a program which generated two log files. The first one contains the history of all phone calls. An entry `1 A B` denotes a new dependency introduced by the programmers, and `0 A B` denotes a query from a customer meaning "should I install A before B?". The second log is a history of all answers given to customers. An example is given in Table 1.

After a long period of testing, MacrOS is finally ready to enter beta stage. But Bob wants to be sure that there were no mistakes during the alpha testing. He wants to check if the answers given in the second log file are correct, and you are to help him. However, Bob does not give you the second log file. Moreover, he modified the first log in a tricky way: after each line corresponding to customer phone call that should have been answered with `NO`, he started/stopped reversing all lines corresponding to programmers calls; see the example below. Taking into account the Bob's modification of the first log file, you should give all the answers to the customers' questions.

First log file	Second log file (answers)
1 1 3	
0 1 3	YES
1 1 4	
0 5 2	NO
1 3 2	
1 6 5	
0 1 5	YES
1 4 5	
0 5 3	NO
1 1 2	

Table 1. The first answer is `YES` because package `1` should be installed before `3`.

Modified first log file
1 1 3
0 1 3
1 1 4
0 5 2 – start reversing
1 **2 3**
1 **5 6**
0 1 5
1 **5 4**
0 5 3 – stop reversing
1 1 2

Table 2. Reversed numbers of packages are written in bold.

Multiple Test Cases

The input contains several test cases. The first line of the input contains a positive integer $Z \leq 15$, denoting the number of test cases. Then Z test cases follow, each conforming to the format described

in section *Single Instance Input*. For each test case, your program has to write an output conforming to the format described in section *Single Instance Output*.

Single Instance Input

Two integers n and m ($1 \leq n \leq 10^5$, $0 \leq m \leq 10^5$) separated by a single space are given in the first line of an input instance. These denote the number of MacrOS' packages and the number of dependencies before alpha tests took place, respectively. The following m lines describe the initial list of dependencies. Each contains two numbers A and B ($1 \leq A, B \leq n$) separated by a single space, denoting that package B depends on A. Some dependencies may appear multiple times, but there will be no pair of mutually dependent packages. The next part of the input contains the Bob's modified version of the first log file, with the format described above. The total number of all phone calls is at most 10^5. The input instance ends with a line containing three zeros, `0 0 0`.

Single Instance Output

Your program should print the content of the second log file. This means that for each input line `0 A B`, you should print `YES` if package A should be installed first and `NO` if B should be installed first. **You can assume that all the queries have a unique answer, i.e., at the moment of such query there was already a dependency between packages A and B.**

Example

<table>
<tr><th>Input</th><th>Output</th></tr>
<tr><td>2
6 4
1 6
6 3
3 4
2 5
1 1 3
0 1 3
1 1 4
0 5 2
1 2 3
1 5 6
0 1 5
1 5 4
0 5 3
1 1 2
0 0 0
6 3
1 2
5 3
4 6
1 2 5
0 1 5
1 4 1
0 3 4
0 0 0</td><td>YES
NO
YES
NO
YES
NO</td></tr>
</table>

I — Islands

Deep in the Carribean, there is an island even stranger than the Monkey Island, dwelled by Horatio Torquemada Marley. Not only it has a rectangular shape, but is also divided into an $n \times m$ grid. Each grid field has a certain height. Unfortunately, the sea level started to raise and in year i, the level is i meters. Another strange feature of the island is that it is made of sponge, and the water can freely flow through it. Thus, a grid field whose height is at most the current sea level is considered *flooded*. Adjacent unflooded fields (i.e., sharing common edge) create unflooded areas. Sailors are interested in the number of unflooded areas in a given year.

An example of a 4×5 island is given below. Numbers denote the heights of respective fields in meters. Unflooded fields are darker; there are two unflooded areas in the first year and three areas in the second year.

Year 1:

1	2	3	3	1
1	3	2	2	1
2	1	3	4	3
1	2	2	2	2

Year 2:

1	2	3	3	1
1	3	2	2	1
2	1	3	4	3
1	2	2	2	2

Multiple Test Cases

The input contains several test cases. The first line of the input contains a positive integer $Z \leq 20$, denoting the number of test cases. Then Z test cases follow, each conforming to the format described in section *Single Instance Input*. For each test case, your program has to write an output conforming to the format described in section *Single Instance Output*.

Single Instance Input

The first line contains two numbers n and m separated by a single space, the dimensions of the island, where $1 \leq n, m \leq 1000$. Next n lines contain m integers from the range $[1, 10^9]$ separated by single spaces, denoting the heights of the respective fields. Next line contains an integer T $(1 \leq T \leq 10^5)$. The last line contains T integers t_j, separated by single spaces, such that $0 \leq t_1 \leq t_2 \leq \ldots \leq t_{T-1} \leq t_T \leq 10^9$.

Single Instance Output

Your program should output a single line consisting of T numbers r_j separated by single spaces, where r_j is the number of unflooded areas in year t_j.

Example

Input	Output
1 4 5 1 2 3 3 1 1 3 2 2 1 2 1 3 4 3 1 2 2 2 2 5 1 2 3 4 5	2 3 1 0 0

J — Jack's socks

Jack is a scientist. As you probably realize, this means he does not pay much attention to what he wears daily. He is also a man, which means that he knows the names of no more than six colors, cannot tell the difference between ecru and white and associates plum only with fruits. But today he is leaving for a conference. He has just pulled a set of single socks out of washing machine and must pair them. He can say which sock is similar to which one, and he may pair only these socks which seem similar. However, many socks are similar to many others. Worse than that, similarity relation is not necessarily transitive. For example, for Jack blue feels similar to seagreen and seagreen to green, but Jack can distinguish between blue and green and say that they are not similar.

Jack wonders if there is exactly one way to pair all his socks. Help him by writing an appropriate program. Do not worry: he might be a scientist, but he is not going to wear his socks with sandals.

Multiple Test Cases

The input contains several test cases. The first line of the input contains a positive integer $Z \leq 50$, denoting the number of test cases. Then Z test cases follow, each conforming to the format described in section *Single Instance Input*. For each test case, your program has to write an output conforming to the format described in section *Single Instance Output*.

Single Instance Input

The first line of input instance contains two integers n and m separated by a single space, where $1 \leq n \leq 1000$ and $0 \leq m \leq 10000$. Number n is even and denotes the number of socks; they are numbered from 1 to n. Each of the following m lines contains two numbers $a_i \neq b_i$ separated by a single space, which means that socks a_i and b_i are similar. Each similarity pair is listed exactly once, i.e., if (a_i, b_i) occurs, then neither (a_i, b_i) nor (b_i, a_i) appears later in this list.

Single Instance Output

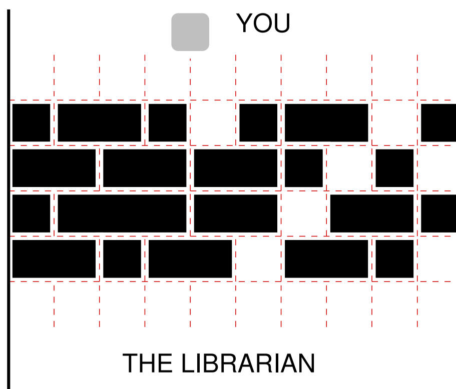
For each input instance your program should check whether there exists exactly one way of pairing all these socks. If not, it should output a line containing `NO`. Otherwise, it should output `YES` in the first line, followed by $n/2$ lines, each containing a sock pair from this pairing separated by a single space. Pairs should be output in sorted order, i.e., for each pair (c, d), it should hold that $c < d$ and for any two consecutive pairs (c, d) and (e, f), $c < e$.

Example

Input	Output
2 4 4 1 2 2 3 3 4 4 1 4 3 1 2 2 3 3 4	NO YES 1 2 3 4

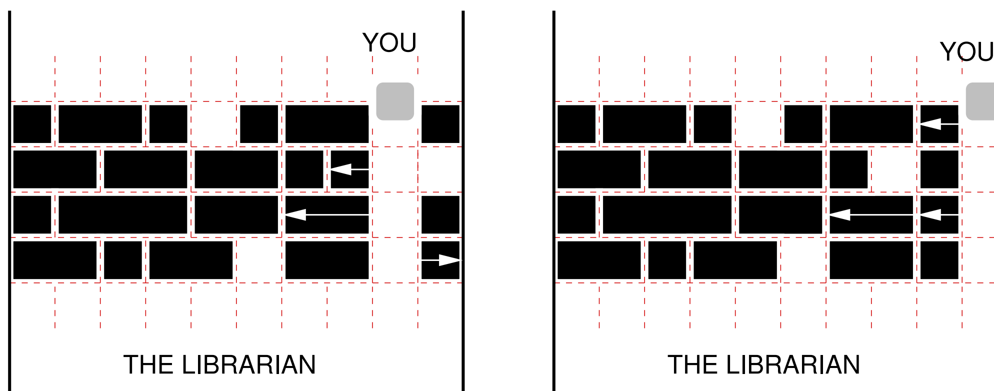
K - Knowledge for the masses

You are in a library equipped with bookracks that move on rails. There are many parallel rails, i.e., the bookracks are organized in several rows, see figure:



The boockracks in the library. There is no passage to the librarian at the moment.

To borrow a book, you have to find the librarian, who seems to hide on the opposite side of the bookracks. Your task then is to move the racks along the rails so that a passage forms. Each rack has a certain integer width, and can be safely positioned at any integer point along the rail. (A rack does not block in a non-integer position and could accidentally move in either direction). The racks in a single row need not be contiguous — there can be arbitrary (though integer) space between two successive bookracks. A passage is formed at position k if there is no bookrack in the interval $(k, k+1)$ in any row (somehow you don't like the idea of trying to find a more sophisticated passage in this maze.)



The passages formed in the library: at position 8 (the left figure) and at position 9 (the right figure). Both attained at cost 3 by moving the bookracks marked with arrows.

Moving a rack requires a certain amount of effort on your part: moving it in either direction costs 1. This cost does not depend on the distance of the shift, which can be explained by a well known fact that static friction is considerably higher than kinetic friction. Still, you are here to borrow a book, not to work out, so you would like to form a passage (at any position) with as little effort as possible.

Multiple Test Cases

The input contains several test cases. The first line of the input contains a positive integer $Z \leq 15$, denoting the number of test cases. Then Z test cases follow, each conforming to the format described in section *Single Instance Input*. For each test case, your program has to write an output conforming to the format described in section *Single Instance Output*.

Single Instance Input

Two space separated integers R and L ($1 \leq R$, $1 \leq L \leq 10^6$) are given in the first line of an input instance. They denote the number of rows and the width of each and every row, respectively. Then R lines with rows descriptions follow. Each such line starts with an integer n_i, followed by n_i integers $a_{i,1}, a_{i,2}, \ldots a_{i,n_i}$, all separated by single spaces. Number $a_{i,j}$ denotes either the width of a bookrack when $a_{i,j} > 0$ or a unit of empty space when $a_{i,j} = 0$. Note that for any row i, $\sum_j a_{i,j}$ equals L minus the number of $a_{i,j}$ that are equal to zero. You may assume that $n_1 + n_2 + \cdots + n_R \leq 2 * 10^7$. Moreover, there will be at least one 0 in the description of each row, which means that creating a passage is always possible.

Single Instance Output

In the first line, your program should output the minimum cost of making a passage through the bookracks. In the second line, it should print out the increasing sequence of all the positions at which a minimum cost passage can be formed.

Example

Input	Output
1 4 10 8 1 2 1 0 1 2 0 1 7 2 2 2 1 0 1 0 6 1 3 2 0 2 1 7 2 1 2 0 2 1 0	3 8 9