Division expression (Poland)

Division expression is an arithmetic expression of the form

$$x_1/x_2/x_3/\dots/x_k$$

where x_i is a positive integer, for $i, (1 \le i \le k)$. Division expression is evaluated from the left to the right. For instance the value of the expression

$$1/2/1/2$$

is $1/4$. One can put parentheses into expression in order to change its value. For example the value of the expression

$$(1/2)/(1/2)$$

is 1. We are given a division expression E. Is it possible to put some parentheses into E to get an expression E' whose value is an integer number.

Task: Write a program that for each data set from a sequence of several data sets:

- reads an expression E from the text file `DIV.IN`,
- verifies whether it is possible to put some parentheses in E to get a new expression E' whose value is an integer number,
- writes the result to the text file `DIV.OUT`

Input data: The first line of the file `DIV.IN` contains one positive integer $d, (d \le 5)$. This is the number of data sets. The data sets follow. The first line of each data set contain an integer $n, (2 \le n \le 10000)$. This is the number of integers in the expression. Each of the following n lines contains exactly one positive integer not greater than $1\,000\,000\,000$. The ith number is the ith integer in the expression.

Output data: For each $i, (1 \le i \le d)$ your program should write to the ith line of the output file `DIV.OUT` one word `YES`, if the ith input expression can be transformed into an expression whose value is an integer number, and the word `NO` in the other case.

Example: For the input file `DIV.IN`:

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

the correct result is the output file `DIV.OUT`:

```
YES
NO
```

Stickers (LATVIA)

Charles is an auto races fan and he has decided to make his own model's collection. In the shop it is possible to buy models in closed and covered boxes. In each box there are parts for one model and a set of stickers with images of digits. In every box the set of stickers is the same. Charles decided to label models by consecutive integers starting from 1. For example, to label the 2070-th model four stickers are necessary: one sticker with "2", two with "0" and one with "7".

Charles completes every model in the following way: he opens a new box, builds the model and labels it using sticker(s). He can use stickers from current and previously opened boxes, but it is not allowed to open an additional new box to get at missing stickers.

Write a program which for a given set of stickers counts how many models Charles can label in the described way.

Input data: In the only line of text file `STI.IN` ten one-digit integers

$$i_0, i_1, i_2, i_3, i_4, i_5, i_6, i_7, i_8, i_9$$

are given, where i_j is the number of stickers with digit $j, (0 \leq j \leq 9)$ in the sticker set of every box. Each two neighbour digits are separated by one space symbol.

Output data: The only line of text file `STI.OUT` should contain one integer – number of labeled models.

Examples: Input data (file `STI.IN`)

```
1 1 1 1 1 1 1 1 1 1
```

Output data (file `STI.OUT`)

```
199990
```

Input data (file `STI.IN`)

```
3 4 5 4 3 4 5 4 3 4
```

Output data (file `STI.OUT`)

```
499999994999999999949999999973
```

MUTEXES (ESTONIA)

Modern programming languages allow writing programs that consist of several threads of execution. This is as if several programs are running in parallel in the same address space, accessing the same variables. Often the threads need to be synchronized with each other. For instance, one thread may need to wait for another to complete some computation and store the result into some variable.

The simplest tool for thread synchronization is *mutex*. A mutex is a special object that can be in *locked* or *unlocked* state. A locked mutex is always owned by exactly one thread. There are two operations that a thread can apply to a mutex: LOCK and UNLOCK.

When a thread applies LOCK to a mutex that is currently unlocked, the mutex becomes locked and the thread acquires ownership of the mutex. If a thread tries to apply LOCK to a mutex that is already locked by some other thread, the thread is blocked until the mutex is unlocked.

When a thread applies UNLOCK to a mutex owned by the thread, the mutex becomes unlocked. If there were other threads waiting to LOCK the mutex, one of them is granted ownership of the mutex. If there were several threads waiting, one is selected arbitrarily.

One of the common problems in multithreaded programs are deadlocks. A deadlock occurs when two or more threads are waiting for each other to release a mutex and none of them can continue. A deadlock occurs also when a thread is waiting for a mutex that was locked by another thread that has terminated without releasing the mutex.

Task: You are provided descriptions of some threads and your task is to decide whether deadlocks can occur.

Each of the threads is a sequence of instructions of the following form:

```
LOCK <mutex>
UNLOCK <mutex>
```

You may assume the following about the commands:

- names of *mutexes* are uppercase letters $A \ldots Z$;
- no thread attempts to lock a mutex it already owns;
- no thread attempts to unlock a mutex it does not own.

Input data: The first line of input file `MUT.IN` contains the number of threads $M, (1 \leq M \leq 5)$ and is followed by M blocks describing each thread. The first line of a block describing thread i contains the number of instructions in this thread $N_i, (1 \leq N_i \leq 10)$ and is followed by N_i lines with instructions. Instructions do not contain extraneous whitespace.

Output data: The first line of output file `MUT.OUT` must contain one number: D. D must be 1 if deadlocks are possible, or 0 if not.

If deadlocks are possible, the second line must describe a state of program in which a deadlock occurs. If there are several states with a deadlock, output any of them. In this case we are looking for complete deadlock, in which none of the threads can continue execution – a thread must be either terminated or blocked by a mutex. If deadlocks are not possible, the line must be empty.

State of program is described by specifying the zero-based index of current instruction for each thread in the order in which the threads are presented in input file. For a terminated thread, output -1 as the index. The indexes must be on a single line and separated by spaces.

Sample:

```
MUT.IN          MUT.OUT
2               1
1               -1 1
LOCK X
2
LOCK Y
LOCK X
```

The solution will not receive points for test cases, where there are no deadlocks, if it has not solved any test case, where deadlocks are possible.