STICKERS (LATVIA)

Charles is auto races fan and he decided to make his own model's collection. In shop it is possible to buy models in closed and covered boxes. In each box there are one model parts and 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 complete every model in the following way: he opens new box, builds a model and labels it using sticker(s). He can use stickers from current and previously opened boxes, but it is not allowed to open additional new box to get at missing stickers.

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

Input data: In the only line of text file `STICKERS.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 a 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 `STICKERS.OUT` should contain one integer – number of labeled models.

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

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

Output data (file `STICKERS.OUT`)

```
199990
```

Input data (file `STICKERS.IN`)

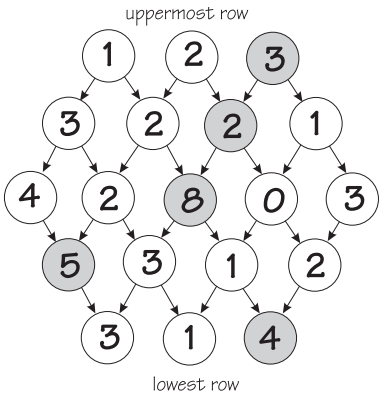
```
3 4 5 4 3 4 5 4 3 4
```

Output data (file `STICKERS.OUT`)

```
49999999499999999949999999973
```

HONEYCOMB PROBLEM (FINLAND)

Figure 1 shows a honeycomb of numbers (side length of the honeycomb is 3). A route starts from some node in the uppermost row and ends to some node in the lowest row. From a node, the route can continue only diagonally down to the left or diagonally down to the right. When creating a route through the honeycomb, you are allowed to make *at most* one swap of two numbers on *at most* one horizontal row of the honeycomb. (Swapping essentially means that in one chosen row you are allowed to place the greatest number of that row to any position on the same row.) Your task is to write a program that calculates the highest sum of numbers on any route using the ability of swapping two numbers on a chosen row.



FIGUR 1. *A honeycomb of side length* 3.

Restrictions:

- The *numbers* in the nodes are integers between 0 and 99.
- The *side length* of the honeycomb is an integer between 1 and 99.

Input data: The side length of the honeycomb is in the first row of the file `INPUT.TXT`. If the side length is n, the honeycomb consists of $2n-1$ rows. Numbers on the rows of the honeycomb are on the following $2n-1$ rows as follows:

```
3
1 2 3
3 2 2 1
4 2 8 0 3
5 3 1 2
3 1 4
```

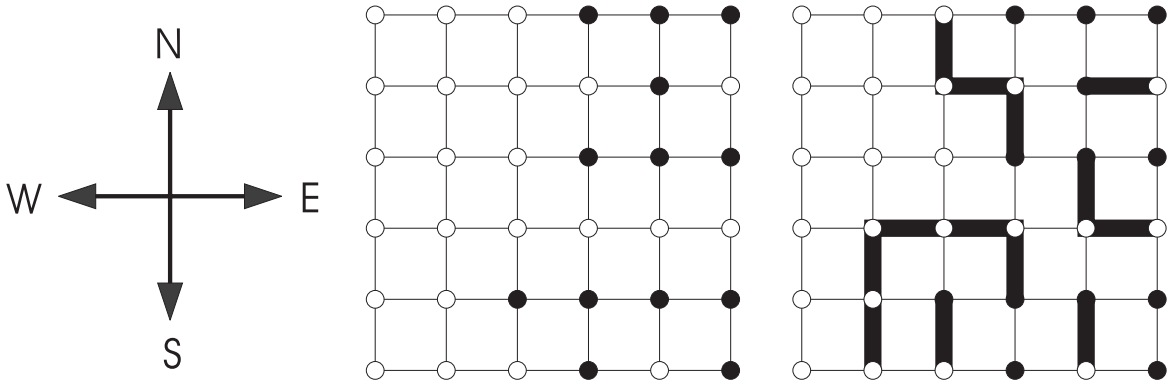
Output data: The highest sum is written as an integer in the file `OUTPUT.TXT`. In the example of Figure 1:

```
22
```

In the Figure 1 the correct solution ($3+2+8+5+4=22$) is gray shaded. Notice that number '5' on the 4th row (from the top) is swapped to the 3rd position (from the left) on that row.

ELECTRONICAL PLATE (LITHUANIA)

A square grid is carved on the top of a square plate. The place where two gridlines cross is called a node. There are $n \times n$ nodes in the grid.



FIGUR 2. *The problem (center) and the solution (right)*

Some nodes contain pins. The task is to connect those pins to the nodes on the boundary of the plate using electronic circuits. A circuit can be laid out only on the grid (e.g. it can't be laid out slantwise). Any two circuits can't have a common point, therefore any two circuits can't be laid out on the same gridline, nor on the same node. A circuit can't be laid out on the boundary grid (the circuit must be finished as soon as it reaches boundary) nor on a node, containing another pin.

An example of an electronic plate containing pins is given in *figure 2, center*. Black dots in the picture represent pins.

Problem. Write a program to connect as many pins as possible to the nodes on the boundary. The pins which are already on the boundary satisfy the requirements and there is no need to make any circuits for them.

If there exists more that one solution find any of them.

Input data: Input data are given in the text file `ELEKT.IN`. The first line of this file contains an integer $n (3 \leq n \leq 15)$.

Each of the following n lines consists of n digits separated by one space. The digits can be 1 or 0. One (1) means a pin, zero (0) - a node without a pin in the appropriate place of the grid.

The nodes are numbered from 1 to $n \times n$ first from left to right and then from the top to bottom. The number of the node the pin is on is the identifier of the pin.

Output data: Output the results to the text file `ELEKT.OUT`. Write k - the maximum number of pins connected to the boundary using electronic circuits - in the first line of the file. A circuit connecting an appropriate pin to the boundary should be described in each of the following k lines. First comes the identifier of the pin, then the sequence of letters, describing the directions of the circuit: *E* - to the East, *W* - to the West, *N* - to the North, *S* - to the South. One space should be left between the identifier and the sequence of letters, and no spaces should be left between the letters in the sequence.

The results should be presented in the increasing order of pin identifiers.

Examples:

INPUT DATA	OUTPUT DATA
6	6
0 0 0 1 1 1	11 E
0 0 0 0 1 0	16 NWN
0 0 0 1 1 1	17 SE
0 0 0 0 0 0	27 S
0 0 1 1 1 1	28 NWWSS
0 0 0 1 0 1	29 S

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.

If 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.

If 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.

There are two common kinds of problems in multithreaded programs: deadlocks and race conditions. 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.

A race condition occurs when two or more threads are accessing the same variable simultaneously. If one thread is writing a variable while another is reading it, the reading thread may receive an arbitrary value: the result does not even have to be either the old or the new value of the variable. If two or more threads are writing the same variable simultaneously, the result is also undefined: the variable may receive an arbitrary value. However, any number of threads can read a variable simultaneously without creating a race condition.

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

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

```
LOCK <mutex>
UNLOCK <mutex>
<variable>=<integer>
<variable>=<variable>
<variable>=<variable>+<variable>
<variable>=<variable>-<variable>
```

You may assume the following about the commands:

- names of *mutexes* are uppercase letters `A...Z`
- names of *variables* are lowercase letters `a...z`;
- all *integers* are in range `0...255`
- 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 `MUTEXES.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 $N_i, (1 \leq N_i \leq 30)$ and is followed by N_i lines with instructions. Instructions do not contain extraneous whitespace. Each thread contains at most 10 instructions operating on mutexes, the rest are assignments.

Output data: The first line of output file `MUTEXES.OUT` must contain two numbers: D and R. D must be 1 if deadlocks are possible, or 0 if not. Similarly, R must be 1 if race conditions 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.

The third line must contain description of a program state with a race condition if race condition can occur, or be empty if race condition is not possible. If there are several race conditions, output any of them.

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:

```
MUTEXES.IN      MUTEXES.OUT
2               0 1
5
a=5             1 3
b=a
LOCK X
c=a+b
UNLOCK X
4
LOCK X
c=2
UNLOCK X
b=3
```

DIVISION EXPRESSION (POLAND)

Division expression is an arithmetic expression of the form

$$x_1/x_2/x_2/.../x_k$$

where x_i is a positive integer, for $i, (1 \leq i \leq k)$ 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 $d, (d \leq 5)$ not larger than 5. This is the number of data sets. The data sets follow. The first line of each data set contain an integer $n, (2 \leq n \leq 10000)$. 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 $i, (1 \leq i \leq d)$ 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:

```
YES
NO
```

TIME ZONES (SWEDEN)

You're a businessman that has customers all over the world. During one day you get *exactly* one message from *each* time zone. The messages come with a time. Unfortunately due to a millennium bug their local time is given without anything that identifies their time zone.

Task: Your task is to identify from which time zones the messages originated.

In this task the number of hours in one day varies between $5 \ldots 60$. The number of time zones is always the same as the number of hours and each time zone has a time displacement which is an integral number of hours.

You are receiving the calls in "GMT", that is in time zone 0 without any time displacement. Your are also receiving one call from your own time zone.

The time zones are counted westward. That is, to get the time in your time zone you should add an integral number of hours to the local time in the other time zone. Note that this is not the common way to count the zones, normally zone 2 would be known as "-2:00".

For example, if the local time in time zone 2 is $03{:}15$ it is $05{:}15$ in your zone ("GMT").

No two messages arrive at the same minute. And all between $0{:}00$-$(n-1){:}59$ (in the case of a n-hour day). The date line goes between zone 0 and the last zone and can thus be ignored.

Input data: The first row contains the number of time zones, $n, 5 \leq n \leq 60$, which also is the number of hours in one day and the number of messages you received.

Each of the following n rows contains a time given as $hhmm$ (2 digits hours and 2 digits minutes where $0 \leq hours \leq n-1$ and $0 \leq minutes \leq 59$). The rows are ordered in chronological order, i.e. the first call that arrived is on the first row.

There is one unique solution in all given test examples.

Output data: One row containing n numbers from 0 to $n-1$ identifying in which time zones the n calls originated. The first number corresponds to the first time given in the input etc.

Example: `ZONES.IN`:

```
5
0017
0250
0400
0201
0002
```

`ZONES.OUT`:

```
3 1 0 2 4
```

Note that call 3 must come from time-zone 0 or it would have arrived at a time later than 4:59 which is the last minute in the 5-hour days in this example.