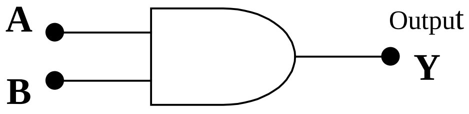
# National Collegiate Programming Contest 1998

## Notes for Contestants

1. There are 8 problems for the contest. You may work on the problems in any order.
2. You may use any algorithm/method to solve the problems. However, the execution time of your program for each problem must not exceed 30 seconds, otherwise the program will be considered **run-time exceeded**.
3. The judging system, **PC$^2$**, is provided by ACM. For detailed information of system environment and operational instructions, please refer to the additional *Contest Handbook*.
4. Input files for the problems are preinstalled in the system for automatic judging. The input files are named according to the problem ID, where "**px.dat**" is the input file for **Problem X**.
5. If you have any questions during the contest, you may communicate with the judges by sending message to the judges through the *Clarification System* in **PC$^2$**.
6. Since Visual Basic does not support console mode applications, VB programs should print output to the file "**output.txt**" instead of the standard output.

# Problem A Logic Circuit Simulation

**AND Gate (AND Operation)**

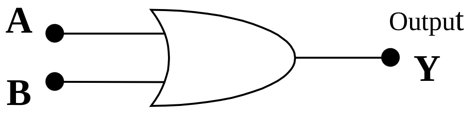


$\mathbf{Y} = \mathbf{A}\ \text{AND}\ \mathbf{B}$

| Input | | Output |
|---|---|---|
| A | B | Y |
| 0 | 0 | 0 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 1 |

If any input is 0, then the output is 0.

**OR Gate (OR Operation)**

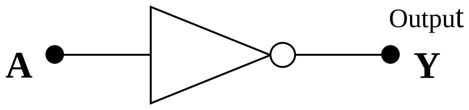


$\mathbf{Y} = \mathbf{A}\ \text{OR}\ \mathbf{B}$

| Input | | Output |
|---|---|---|
| A | B | Y |
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 1 |

If any input is 1, then the output is 1.

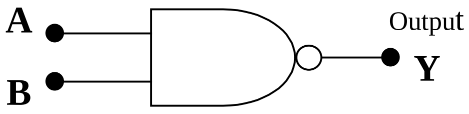
**Inverse Gate (NOT Operation)**



$\mathbf{Y} = \overline{\mathbf{A}}$

| Input | Output |
|---|---|
| A | Y |
| 0 | 1 |
| 1 | 0 |

**NAND Gate (AND+NOT Operation)**

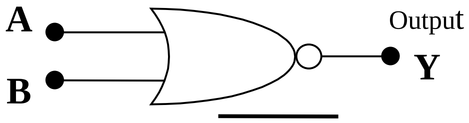


$\mathbf{Y} = \overline{\mathbf{A}\ \text{AND}\ \mathbf{B}}$

| Input | | Output |
|---|---|---|
| A | B | Y |
| 0 | 0 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 0 |

If any input is 0, then the output is 1.

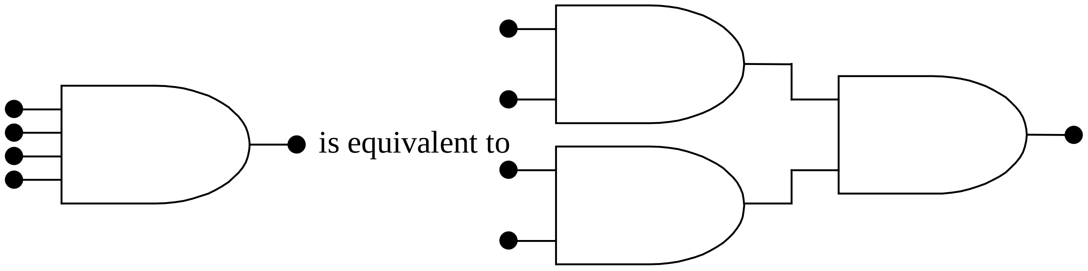
**NOR Gate (OR+NOT Operation)**

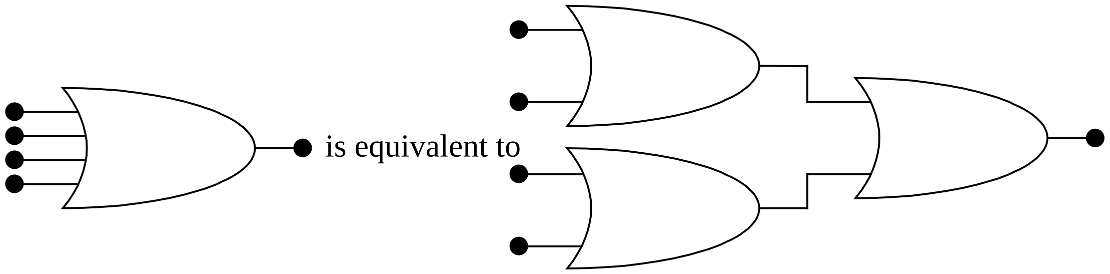


$\mathbf{Y} = \overline{\mathbf{A}\ \text{OR}\ \mathbf{B}}$

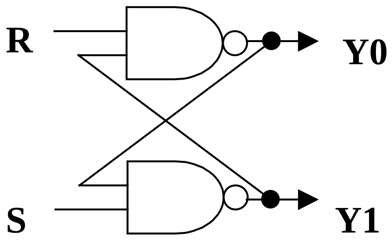
| Input | | Output |
|---|---|---|
| A | B | Y |
| 0 | 0 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 0 |

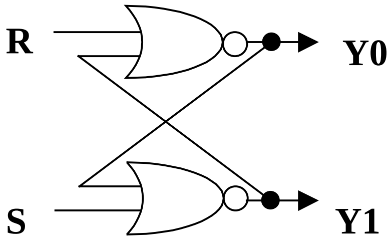
If any input is 1, then the output is 0.





Within the circuit, you will see the following circuit modules, which are called flip-flops. You may like to use the following truth tables to determine their output values in terms of the input data and their previous output status.

| | | Input | | Output | |
|---|---|---|---|---|---|
| | | R | S | Y0 | Y1 |
| <br> | Initial status→ | 1 | 1 | 1 | 0 |
| | | 0 | 1 | 1 | 0 |
| | | 1 | 0 | 0 | 1 |
| | | 1 | 1 | Y0- | Y1- |
| | | 0 | 0 | 1 | 1 |
| | Y0- and Y1-: If the previous outputs are not all 1's, the outputs remain unchanged. However, if the previous outputs are all 1's, they will go back to the initial status. | | | | |

| | | Input | | Output | |
|---|---|---|---|---|---|
| | | R | S | Y0 | Y1 |
| <br> | Initial status→ | 0 | 0 | 0 | 1 |
| | | 0 | 1 | 1 | 0 |
| | | 1 | 0 | 0 | 1 |
| | | 0 | 0 | Y0- | Y1- |
| | | 1 | 1 | 0 | 0 |
| | Y0- and Y1-: If the previous outputs are not all 0's, the outputs remain unchanged. However, if the previous outputs are all 0's, they will go back to the initial status. | | | | |

A logic circuit using the logic gates above is designed to implement a desired logic operation. You are asked to write a program to simulate the circuit shown in Figure 1. Please note that A0 is the least significant bit and A15 is the most significant bit.

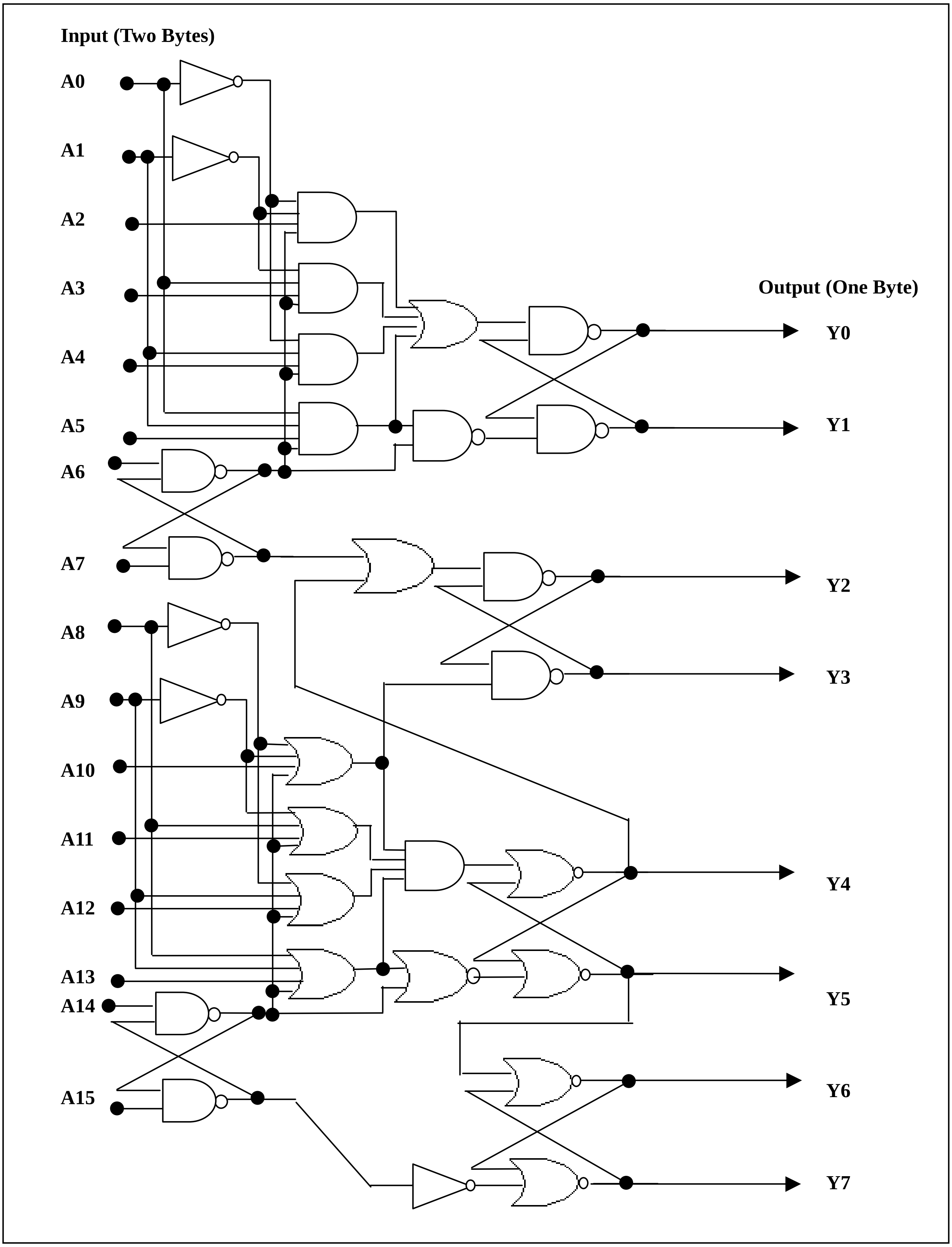


FIGURE 1. The circuit diagram to be simulated

# The Input

A set of two-byte data in hexadecimal will be read from a disk file with the format described as follows:

| Line Number | Data in the File | Explanation |
|---|---|---|
| 1 | *n* (a positive integer less than 512) | The length of the two-byte data sets |
| 2 | 0503 (two-byte data in hexadecimal) | The first data input to the circuit |
| 3 | F3B4(two-byte data in hexadecimal) | The second data input to the circuit |
| … | … | … |
| *n*+1 | F2B2(two-byte data in hexadecimal) | The last data input to the circuit |

☞ Please note that the **Line Number** and **Explanation** are not given in the file. They are shown here only to assist you in reading the data.

# The Output

Write your output Byte data (Y0 – Y7) sets to the standard output. The format should be the same as the input. The only difference is that the output has only one-byte data in a line. **When hex digits A~F are output**, **print out the upper case letters**. Please note that since there is feedback in the circuit, the outputs are often affected by the previous output status. This means that the same inputs may produce different outputs depending on the previous output data.

# Sample Input

```
10
7F80
569A
8D69
33F3
73B3
3344
66BC
8840
4000
73B1
```

☞ The first number of the input data is the most significant bits: A12 to A15, and the last number is the least significant bits: A0 to A3.

## Sample Output

```
10
A5
A5
25
A5
AE
A5
A5
25
95
99
```

☞ The first number is the most significant bits: Y4 to Y7, and the second number is for Y0 to Y3.

# Problem B Dropping Balls

A number of $K$ balls are dropped one by one from the root of a fully binary tree structure FBT. Each time the ball being dropped first visits a non-terminal node. It then keeps moving down, either follows the path of the left subtree, or follows the path of the right subtree, until it stops at one of the leaf nodes of FBT. To determine a ball's moving direction a flag is set up in every non-terminal node with two values, either **false** or **true**. Initially, all of the flags are **false**. When visiting a non-terminal node if the flag's current value at this node is **false**, then the ball will first switch this flag's value, i.e., from the **false** to the **true**, and then follow the left subtree of this node to keep moving down. Otherwise, it will also switch this flag's value, i.e., from the **true** to the **false**, but will follow the right subtree of this node to keep moving down. Furthermore, all nodes of FBT are sequentially numbered, starting at 1 with nodes on depth 1, and then those on depth 2, and so on. Nodes on any depth are numbered from left to right.

For example, FIGURE 1 represents a fully binary tree of maximum depth 4 with the node numbers 1, 2, 3, …, 15. Since all of the flags are initially set to be **false**, the first ball being dropped will switch flag's values at node 1, node 2, and node 4 before it finally stops at position 8. The second ball being dropped will switch flag's values at node 1, node 3, and node 6, and stop at position 12. Obviously, the third ball being dropped will switch flag's values at node 1, node 2, and node 5 before it stops at position 10.

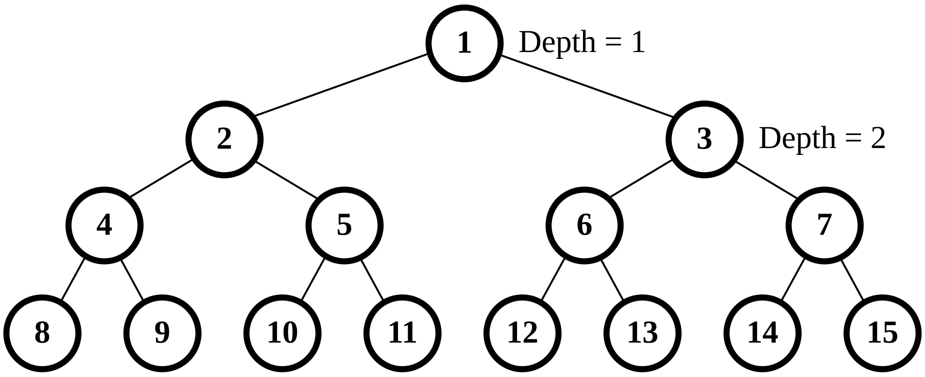


FIGURE 1. An example of FBT with the maximum depth 4 and sequential node numbers.

Now consider a number of test cases where two values will be given for each test. The first value is $D$, the maximum depth of FBT, and the second one is $I$, the $I^{th}$ ball being dropped. You may assume the value of $I$ will not exceed the total number of leaf nodes for the given FBT. Please write a program to determine the stop position *P* for each test case.

For each test cases the range of two parameters *D* and *I* is as below:

$2 \le D \le 20$, and $1 \le I \le 524288$.

# The Input

Contains *l*+2 lines.

| | | |
|---|---|---|
| Line 1 | *l* | the number of test cases |
| Line 2 | $D_1 I_1$ | test case #1, two decimal numbers that are separated by one blank |
| … | | |
| Line *k*+1 | $D_k I_k$ | test case #*k* |
| Line *l*+1 | $D_l I_l$ | test case #*l* |
| Line *l*+2 | -1 | a constant –1 representing the end of the input file |

# The Output

Contains *l* lines.

| | |
|---|---|
| Line 1 | the stop position *P* for the test case #1 |
| … | |
| Line *k* | the stop position *P* for the test case #*k* |
| … | |
| Line *l* | the stop position *P* for the test case #*l* |

# Sample Input

```
5
4 2
3 4
10 1
2 2
8 128
-1
```

# Sample Output

```
12
7
512
3
255
```

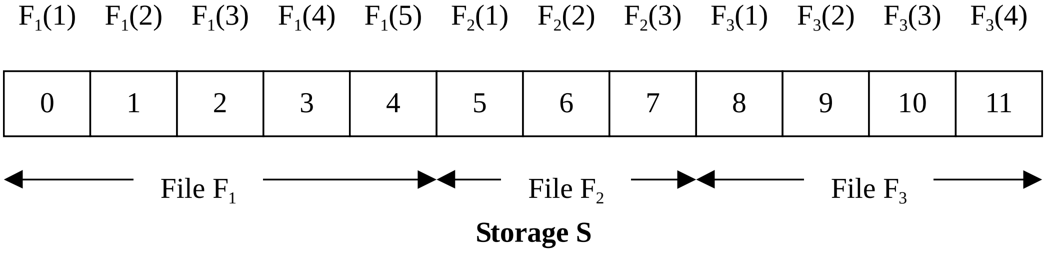
# Problem C Movement of Reading Head

Suppose we have $K$ files representing by $F_1$, $F_2$, $F_2$, …, $F_K$. The total length of these files, measured in block numbers, is $N$ blocks, and the length of each file is $L_i$ block(s) for $1 \le i \le K$. We denote the $b$th block of a file $F_i$ as $F_i(b)$ for $1 \le b \le L_i$; e.g., the 9th block of $F_2$ file is $F_2(9)$, and the 4th block of $F_3$ file is $F_3(4)$. Now consider a storage space S consisting of a single reading head and $N$ blocks with sequential number starting from 0 to $N$-1. These $K$ files are stored to the space S in a sequential order from $F_1$, $F_2$, $F_3$, …, $F_K$. We will assume that there is no spare blocks left for storing these K files. Apparently, this means that

$$\sum_{i=1}^{K} L_i = N$$

When reading from S, a profile array PF is used to indicate the starting block of the reading for every file, and the reading order is to read a block at $F_1$, then a block at $F_2$, …, a block at $F_K$ with one block being read for a file at one time. After $F_K$ is being read, we restart to read the next block at $F_1$, then the next block at $F_2$, …, and the process circulates in this fashion. Within a file when the previous reading has reached to the last block, the next block to be read is the first block of this file. Obviously, the reading head has to move through several blocks during each time of reading. Thus, we define a term TB($P$) to be the total number of blocks that the reading head needs to move for the $P$ consecutive times of reading. Apparently, we will be interested in finding the value of TB($P$). Given the profile array PF you may assume the reading head is initially rested on the starting block of the first file that is going to be read, and thus TB(1) = 0.

For example: let $K$ = 3, $N$ =12, $L_1$ = 5, $L_2$ = 3, $L_3$ = 4, PF be 2, 3, 3. These three files will be stored to S as shown in FIGURE 1.



| $F_1(1)$ | $F_1(2)$ | $F_1(3)$ | $F_1(4)$ | $F_1(5)$ | $F_2(1)$ | $F_2(2)$ | $F_2(3)$ | $F_3(1)$ | $F_3(2)$ | $F_3(3)$ | $F_3(4)$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

FIGURE 1. An example of three files being stored in the storage S.

According to the given PF the reading head is initially rested on the second block of the first file, i.e., $F_1(2)$. When reading from S, the first time of reading is to read the second block of $F_1$, i.e., $F_1(2)$, which is located at position 1. At this time TB(1) = 0. The second time of reading is to read the third block of $F_2$, i.e., $F_2(3)$, which is located at position 7. Thus, the total number of blocks that the reading head has to move for 2 consecutive times of reading, i.e., TB(2), is 6 blocks. Similarly, the third time of reading is to read the third block of $F_3$, i.e., $F_3(3)$, which is located at position 10. This means that the reading head has to move 3 blocks for the third time of reading. Thus, the total number of blocks that the reading head has to move for 3 consecutive times of reading is 9 blocks, i.e., TB(3) = 0 + 6 + 3 = 9 blocks. Similarly, the fourth time of reading is to read the third block of $F_1$, i.e., $F_1(3)$, which is located at position 2. This means that the reading head has to move 8 blocks for the fourth time of reading. Thus, the total number of blocks that the reading head has to move for 4 consecutive times of reading is 17 blocks, i.e., TB(4) = 0 + 6 + 3 + 8 = 17 blocks.

Now given the parameters $K$, $N$, $L_i$, PF, $P$, please write a program to report the value of TB($P$), where

$K$: number of files,
$N$: number of blocks in the storage S,
$L_i$: the length of each file, where each value is separated by a blank,
PF: an array of $K$ integers representing the starting block of the reading for each file where each value is separated by a blank, and
$P$: number of the consecutive times of reading.

The range of each parameter is as below:

$1 \leq K \leq 10$
$1 \leq N \leq 200$
$1 \leq Li \leq 100$ for each $i$
$1 \leq$ entry in PF $\leq 100$ for each file, and
$1 < P \leq 1000$.

# The Input

Contains $l$ + 2 lines.

| | | |
|---|---|---|
| Line 1 | $l$ | the number of test cases |
| Line 2 | $K$ $N$ $L_1$ $L_2$ … $L_k$ PF $P$ | test case #1, 2$K$+3 decimal values each of which is separated by a blank |
| … | | |
| Line $k$+1 | $K$ $N$ $L_1$ $L_2$ … $L_k$ PF $P$ | test case #$k$ |
| … | | |
| Line $l$+1 | $K$ $N$ $L_1$ $L_2$ … $L_k$ PF $P$ | test case #$l$ |
| Line $l$+2 | -1 | a constant –1 representing the end of the input file |

## The Output

Contains *l* lines.

| | |
|---|---|
| Line 1 | output for the value of TB(*P*) at the test case #1 |
| … | |
| Line *k* | output for the value of TB(*P*) at the test case #*k* |
| … | |
| Line *l* | output for the value of TB(*P*) at the test case #*l* |

## Sample Input

```
5
3 12 5 3 4 2 3 3 3
3 12 5 3 4 2 3 3 4
3 12 5 3 4 1 1 1 4
2 10 5 5 1 1 2
2 10 5 5 1 2 2
-1
```

## Sample Output

```
9
17
15
5
6
```

# Problem D Convex Hull Finding

Given a single connected contour, which is either convex or non-convex (concave), use any algorithm to find its **Convex Hull**, i.e., the smallest convex contour enclosing the given shape. If the given contour is convex, then its convex hull is the original contour itself. The maximal size of the shape is 512×512, and the maximal number of the vertices of the shape is 512. Write a program to read the input data (the given shapes) from a disk file, implement your convex hull finding algorithm, and then output the shape data of the results to the standard output.

# The Input

The order of the vertices is counterclockwise in **X-Y** Cartesian Plane (if you consider the origin of the display window is on the upper-left corner, then the orientation of the vertices is clockwise), and none of the neighboring vertices are co-linear. Since all the shapes are closed contours, therefore, the last vertex should be identical to the first vertex. There are several sets of data within a given data file. The negative number –1 is used to separate the data set.

| Line Number | Data in the File | Explanation |
|---|---|---|
| 1 | $K$ | a positive integer showing how many sets of data in this file |
| 2 | $N$ | a positive integer showing the number of vertices for the shape |
| 3 | $X_1$ $Y_1$ | two positive integers for the first vertex ($X_1$, $Y_1$) |
| 4 | $X_2$ $Y_2$ | two positive integers for the next neighboring vertex ($X_2$, $Y_2$) |
| … | | |
| $N$+2 | $X_N$ $Y_N$ | two positive integers for the last vertex ($X_N$, $Y_N$) |
| $N$+3 | -1 | Delimiter |
| $N$+4 | $M$ | a positive integer showing the number of vertices for the next shape |
| $N$+5 | $XX_1$ $YY_1$ | two positive integers for the first vertex |
| … | | |

☞ Please note that the **Line Number** and **Explanation** are not given in the file. They are shown here only to assist you in reading the data.

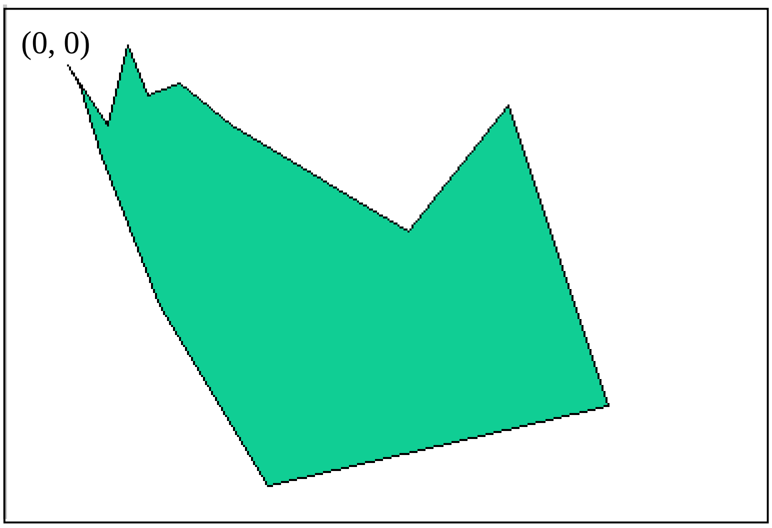
# The Output

Output the convex hull of all *K* input shapes to the standard output. The data format should be the same as the input file. In addition, the vertex with the smallest *Y* value should be the first point and if there are points with the same *Y* value, then the smallest *X* value within those points should be the first point.

## Sample Input

```
3
15
30 30
50 60
60 20
70 45
86 39
112 60
200 113
250 50
300 200
130 240
76 150
47 76
36 40
33 35
30 30
-1
12
50 60
60 20
70 45
100 70
125 90
200 113
250 140
180 170
105 140
79 140
60 85
50 60
-1
6
60 20
250 140
180 170
79 140
50 60
60 20
```

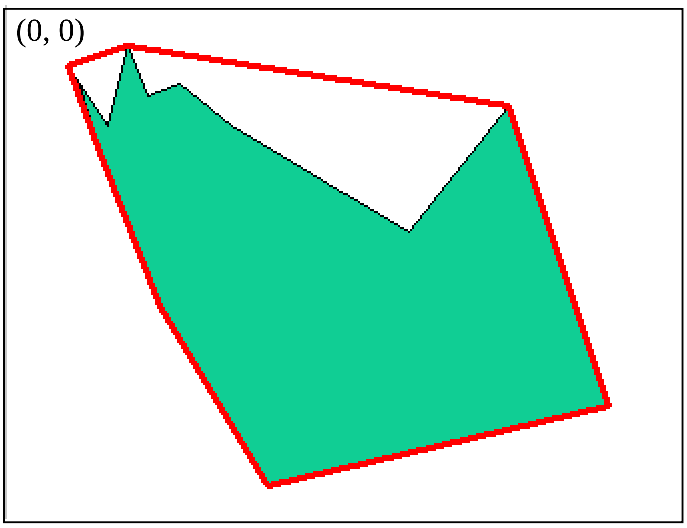
The contour shape of the first data set is shown in figure as follows:



## Sample Output

```
3
8
60 20
250 50
300 200
130 240
76 150
47 76
30 30
60 20
-1
6
60 20
250 140
180 170
79 140
50 60
60 20
-1
6
60 20
250 140
180 170
79 140
50 60
60 20
```

The convex hull of the above shape is shown in the following figure:



# Problem E Whoever-pick-the-last-one-lose

Consider the following **whoever-pick-the-last-one-lose** game. The game is played on a 5×5 board. Initially every array cell has a piece in it. Two players remove pieces alternatively from the board. The player can remove any number of consecutive pieces in a row or column. For example, in the configuration depicted below where one indicates a piece, the player can either remove one piece (**A1**, **A2**, or **B1**), or remove two pieces (**A1** and **A2**, or **A1** and **B1**) simultaneously. The game ends when one player is forced to take the last piece, and the other player wins the game.

| | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|
| **A** | 1 | 1 | 0 | 0 | 0 |
| **B** | 1 | 0 | 0 | 0 | 0 |
| **C** | 0 | 0 | 0 | 0 | 0 |
| **D** | 0 | 0 | 0 | 0 | 0 |
| **E** | 0 | 0 | 0 | 0 | 0 |

Write a program that evaluates board configurations from this game. The program must output “winning” when there exists a winning move that no matter how the opponent responds, it will force the opponent to take the last piece. Otherwise, the program must output “losing”. Note that during the game tree evaluation, if the current configuration has a winning move, then it is not necessary to search any further because the configuration is guaranteed to be winning. This can greatly reduce the game tree search time.

## The Input

The input file contains 5*c* + 1 lines.

| | | |
|---|---|---|
| Line 1 | *c* | the number of configurations |
| Lines 2-6 | … | configuration #1 |
| … | … | |
| Lines 5*c*–3 to 5*c*+1 | … | configuration #*c* |

## The Output

The output contains *c* lines.

| | |
|---|---|
| Line 1 | evaluation result of configuration #1 |
| … | |
| Line *c* | evaluation result of configuration #*c* |

# Sample Input

```
3
1 1 0 0 0
1 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
1 1 0 0 0
0 0 0 0 0
1 1 0 0 0
0 0 0 0 0
0 0 0 0 0
1 1 1 0 0
1 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
```

# Sample Output

```
winning
losing
winning
```

# Problem F Character Decoding

This is a test for decoding characters' values. Assume a numerical expression is encoded in English characters by replacing some digit numbers (from 0 to 9) with English characters. So this kind of numerical expressions can be expressed in new forms, such as

```
    2BAD = ABE + CD
```

Please write a program to decode the expressions in characters and output the numerical value of characters, according to the following rules and assumptions.

1. All character values are integers between 0 to 9 both inclusive.
2. An expression is represented as a set of items combined with operators. Only the operators **+**, **-** and **=** are used in each expression. And at most 5 items are used in one expression.
3. There is one and only one operator **=** in each expression. And only one item is in the left-hand side of the operator **=**.
4. Each item is represented by a combination of capital English characters and digital numbers. The value of the left-most character in each item is not 0.
5. The input data are represented as several rows of numerical expressions and are stored in a file. Each row is an independent expression with other rows. The end of the input file is a star symbol **(*)**.
6. Output the value of the left-most item in each expression row by row, in the same order as that in the input file.
7. If there are multiple solutions, print out the smallest values for each left-most item. If no possible solutions exist, print out a question mark **(?)** instead.

# Input File

Contains *k* lines, with *k*-1 expressions.

| | |
|---|---|
| Line 1 | the first expression |
| … | |
| Line *k*-1 | the (*k*-1)th |
| Line *k* | A star symbol indicating end-of-file |

# Output File

Contains *k* - 1 lines. Each line is the smallest value that satisfies the corresponding expression.

| | |
|---|---|
| Line 1 | value |
| … | |
| Line *k*-1 | value |

# Sample Input

```
CA = AB + 6C
DDE5 = DEFG – EHI + DDH
A = 0
*
```

# Sample Output

```
81
1115
?
```

# Problem G Integral Determinant

Write a program to find the determinant of an integral square matrix. Note that the determinant of a square matrix can be defined recursively as follows: the determinant of a 1×1 matrix $\mathbf{M} = (a_{1,1})$ is just the value $|\mathbf{M}| = a_{1,1}$; further, the determinant of an $n\times n$ matrix is $|\mathbf{M}| = \sum_{i=1}^{n}(-1)^{i+1}\cdot|\mathbf{M}_{1,i}|$. Here the notation $\mathbf{M}_{1,i}$ is the $(n\text{-}1)\times(n\text{-}1)$ matrix by removing the first row and the $i^{\text{th}}$ column of the original $n\times n$ matrix $\mathbf{M}$.

A straightforward method of calculating the determinant of an n x n matrix by the recursive method will end up with $n!$ multiplications, a very time-consuming algorithm. To give you a feeling about this, note that 15! = 1,307,674,368,000. To reduce the time complexity, there are two ways of modifying the original matrix for easier computation.

1. Exchanging two columns (or rows) of a matrix will change the sign of the determinant; for example
$$\begin{vmatrix}1 & 2\\ 3 & 4\end{vmatrix} = -\begin{vmatrix}2 & 1\\ 4 & 3\end{vmatrix}$$
2. Multiplying one column by any constant, and add them to another column will not change the value of the determinant; for example:
$$\begin{vmatrix}2 & 1 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{vmatrix} = \begin{vmatrix}2 & 1 & 3+2\times2\\ 4 & 5 & 6+4\times2\\ 7 & 8 & 9+7\times2\end{vmatrix}$$

Using the above methods, you shall be able to write a program for computing the determinants of matrices, even for a size like 30×30, very efficiently. Below is an example to show how this can be done:

$$\begin{vmatrix}2 & 3 & 5\\ 1 & 6 & 7\\ 4 & 8 & 9\end{vmatrix} = \begin{vmatrix}2 & 3-2 & 5-2\times2\\ 1 & 6-1 & 7-1\times2\\ 4 & 8-4 & 9-4\times2\end{vmatrix} = \begin{vmatrix}2 & 1 & 1\\ 1 & 5 & 5\\ 4 & 4 & 1\end{vmatrix} = -\begin{vmatrix}1 & 1 & 2\\ 5 & 5 & 1\\ 1 & 4 & 4\end{vmatrix}$$

$$= -\begin{vmatrix}1 & 1-1 & 2-1\times2\\ 5 & 5-5 & 1-5\times2\\ 1 & 4-1 & 4-1\times2\end{vmatrix} = -\begin{vmatrix}1 & 0 & 0\\ 5 & 0 & -9\\ 1 & 3 & 2\end{vmatrix} = -\begin{vmatrix}0 & -9\\ 3 & 2\end{vmatrix} = -27$$

Note that the answer shall be an *integer*. That is, all the operations needed are just integer operations; by reducing to floating numbers would result in the round-off errors, which will be considered as the **wrong** answer. Do not worry about the problem of integral overflows problem. You can assume that the given data set will not cause the integer overflow problem. What is emphasized here is the required integer precision.

## The Input

*Several* sets of integral matrices. The inputs are just a list of integers. Within each set, the first integer (in a single line) represents the size of the matrix, *n*, which can be as large as 30, indication an *n*×*n* matrix. After *n*, there will be *n* lines representing the *n* rows of the matrix; each line (row) contains exactly *n* integers. Thus, there is totally $n^2$ integers for the particular matrix.

These matrices will occur repeatedly in the input as the pattern described above. An integer *n* = 0 (zero) signifies the end of input.

## The Output

For each matrix of the input, calculate its (integral) determinant and output them in a line. Output a single star (*****) to signify the end of outputs.

## Sample Input

```
2
5 2
3 4
3
2 3 5
1 6 7
4 8 9
0
```

## Sample Output

```
14
-27
*
```

# Problem H Least Path Cost

Given a positive integer Δ ($0 < \Delta < 10000$), which is called the *overhead*, and $M$ ($0 < M \leq 200$) straight line segments in a two-dimensional plane with the following properties:

1. each line segment has a height, which is a positive integer;
2. two line segments only intersect with each other on endpoints;
3. no two line segments are overlapped.

Each line has a unique number between 1 and $M$. Each endpoint in the plane has a unique number between 1 and $N$ ($0 < N \leq 400$), where $N$ is the total number of endpoints. A line segment is represented by its two endpoints ($n_i$, $n_j$). Let *height*($L$) be the height of a line segment $L$.

A *path* is a sequence of line segments $L_{C_1}$, $L_{C_2}$, …, $L_{C_k}$, such that $k > 1$, $C_i \neq C_j \forall i \neq j$, $L_{C_i}$ intersects with $L_{C_{i+1}}$ for all $1 \leq i < k$, one endpoint of $L_{C_1}$ does not intersect with any other line segments, and one endpoint of $L_{C_k}$ does not intersect with any other line segments. The cost between two intersection line segments $L_{C_i}$ and $L_{C_{i+1}}$ is

$$\left| height(L_{C_i}) - height(L_{C_{i+1}}) \right|$$

That is, for example you can image, the number of stairs that one has to climb (up or down ) by walking from $L_{C_i}$ to $L_{C_{i+1}}$. The cost of a path $L_{C_1}$, $L_{C_2}$, …, $L_{C_k}$ is

$$k \cdot \Delta + \sum_{i=1}^{k-1} \cos t(L_{C_i}, L_{C_{i+1}}).$$

In the example shown in FIGURE 1, Δ = 25, $M$ = 8, and $N$ = 9. Then *cost*($L_2$, $L_3$) = 1 and *cost*($L_1$, $L_6$) = 8. $L_1$, $L_4$, $L_5$ is not a path. There are three paths in the plane. The cost for the path $L_1$, $L_6$, $L_7$, $L_8$ is 109. The cost for the path $L_1$, $L_4$, $L_5$, $L_8$ is 131. The cost for the path $L_2$, $L_3$ is 51. Hence $L_2$, $L_3$ is the path with the least cost.

You may also assume there is at least one path in the plane. Write a program to find the least cost among all paths.

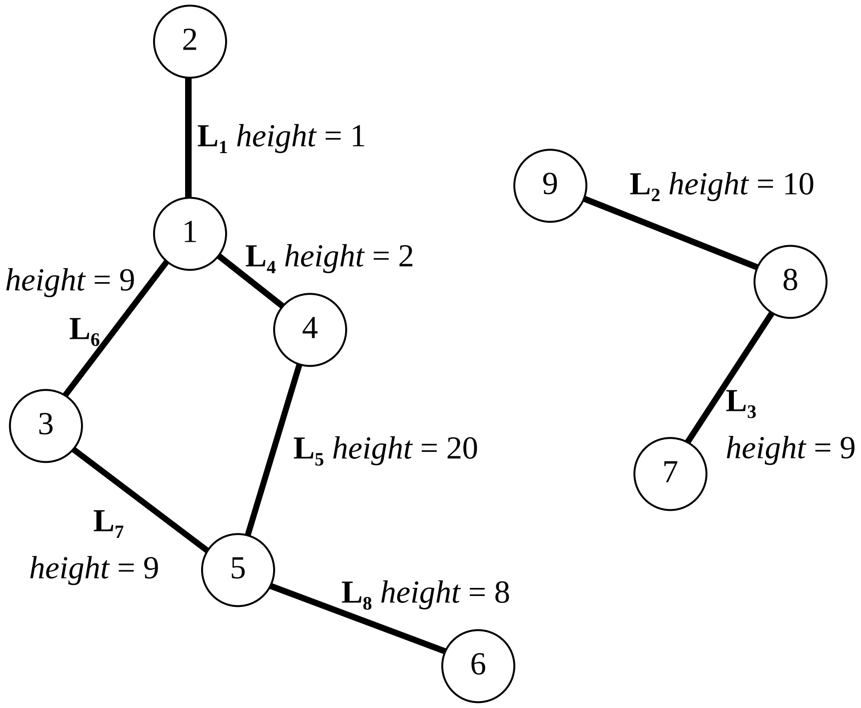

2
$\mathbf{L_1}$ *height* = 1
9
$\mathbf{L_2}$ *height* = 10
1
$\mathbf{L_4}$ *height* = 2
*height* = 9
8
$\mathbf{L_6}$
4
3
$\mathbf{L_3}$
$\mathbf{L_5}$ *height* = 20
*height* = 9
7
$\mathbf{L_7}$
*height* = 9
5
$\mathbf{L_8}$ *height* = 8
6

FIGURE 1. An example of 8 straight lines with 9 endpoints.

## The Input

The first line is *l*, the number of test cases. The first three lines of test case #*i* are $M_i$, $N_i$ and $\Delta_i$ which are the numbers of line segments and endpoints, and the overhead, respectively. The following $M_i$ lines each contains the two endpoints of each line segment, starting from $L_1$ to $L_{M_i}$, and its height. Each line segment is represented by three integers, separated by blanks.

## The Ouput

Contains *l* lines. The *i*th line contains the least cost of all paths in the *i*th test case.

## Sample Input

```
2
8
9
25
1 2 1
8 9 10
7 8 9
1 4 2
4 5 20
1 3 9
3 5 9
5 6 8
6
6
21
1 2 1
1 4 2
4 5 20
1 3 9
3 5 9
5 6 8
```

## Sample Output

```
51
93
```