

BITWISE EXPRESSIONS

In signal processing, one is sometimes interested in the maximum value of a certain expression, containing bitwise AND and OR operators, when the variables are integers in certain ranges. You are to write a program that takes such an expression and the range of each variable as input and determines the maximum value that the expression can take.

For simplicity, the expression is of a specific form, namely a number of subexpressions in parentheses separated by the bitwise AND operator (denoted &). Each subexpression consists of one or more variables separated by the bitwise OR operator (denoted |). Using this convention, it is possible to completely specify the expression by giving the number of subexpressions and, for each subexpression, the number of variables in the subexpression. The variables are simply numbered according to their occurrence in the expression.

An example will clarify this. If the number of subexpressions is 4 and the number of variables in each subexpression is 3, 1, 2, and 2, then the expression will be

$$E = (v_1 | v_2 | v_3) \& (v_4) \& (v_5 | v_6) \& (v_7 | v_8)$$

The bitwise operators are defined in the common way. For example, to perform the operation $21 \& 6$, we first write down the binary form of the numbers (operands): 10101 and 110 (since $21 = 2^4 + 2^2 + 2^0$ and $6 = 2^2 + 2^1$). Every binary digit in the result now depends on the corresponding digit in the operands: if it is 1 in *both* operands, the resulting digit will be one, otherwise it will be zero. As is illustrated below to the left, the resulting value is 4. If instead we want to calculate $21 | 6$, the procedure is the same except that the resulting digit will be one if the corresponding digit is one in *any* of the operands, and thus it will be zero only in the case that the digit is zero in both operands. As is illustrated below in the center, the result is 23. The generalization to more than two operands is straightforward. The rightmost example below illustrates that $30 \& 11 \& 7 = 2$.

		11110
10101	10101	01011
<u>& 00110</u>	<u> 00110</u>	<u>& 00111</u>
00100	10111	00010

INPUT

The input is read from a ~~text file named `bitwise.in`~~ **standard input**. In the first line, two integers N and P are given, where N is the total number of variables ($1 \leq N \leq 100$) and P is the number of subexpressions ($1 \leq P \leq N$). In the next line, P integers (K_1, K_2, \dots, K_P) are given, where K_i is the number of variables in the i -th subexpression. The K_i are all greater than or equal to 1 and their sum equals N . Each of the following N lines contains two integers A_j and B_j ($0 \leq A_j \leq B_j \leq 2\,000\,000\,000$), specifying the range of the j -th variable in the expression according to $A_j \leq v_j \leq B_j$.

OUTPUT

standard output

The output is written into a ~~text file named bitwise.out~~. The content should be one row with a single integer: the maximum value that the expression can take.

EXAMPLE

Assume that we want to limit the values of the eight variables in the expression above according to $2 \leq v_1 \leq 4$, $1 \leq v_2 \leq 4$, $v_3 = 0$, $1 \leq v_4 \leq 7$, $1 \leq v_5 \leq 4$, $1 \leq v_6 \leq 2$, $3 \leq v_7 \leq 4$, and $2 \leq v_8 \leq 3$. This corresponds to the following content of ~~bitwise.in~~:

standard input

```
8 4
3 1 2 2
2 4
1 4
0 0
1 7
1 4
1 2
3 4
2 3
```

If writing in binary notation, one of the best assignments gives the expression $(100 | 011 | 000) \& (111) \& (100 | 010) \& (100 | 011)$, from which we note that all subexpressions may become equal to 7 except the third. Thus, the program should write a ~~file bitwise.out~~ with the content:

```
standard
output
6
```

GRADING

In 30% of the test cases, the number of possible assignments is less than one million.