TASK 1: PRIME

A prime number is a natural number which has exactly two distinct natural number divisors: 1 and itself. The first prime number is 2. Can you write a program that computes the n^{th} prime number, given a number $n \leq 10000$?

Input File: `PRIME.IN`

The input file PRIME.IN contains just one number which is the number n as described above. The maximum value of n is 10000.

Example

```
30
```

Output File: `PRIME.OUT`

The output file consists of a single integer that is the n^{th} prime number.

Example

The output file for the example above contains the number

```
113
```

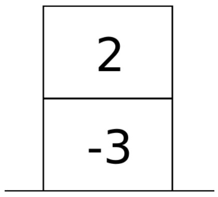
because 113 is the 30th prime number.

TASK 2: LVM

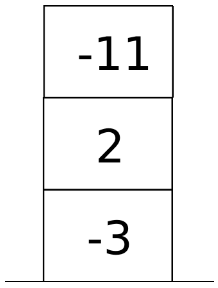
You are designing a virtual machine for a new programming language called Lombok. The Lombok Virtual Machine (LVM) runs an assembler-like machine code. It operates on a stack and a single register.

In detail, the instructions work as follows:

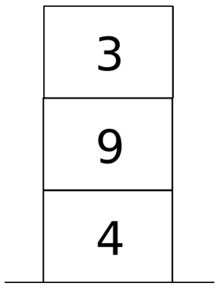
PUSH x: This instruction pushes a given integer onto the stack. If the stack for example looks like this



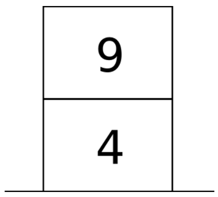
and the machine executes the instruction `PUSH -11`, the stack looks like this afterwards:



STORE: This instruction takes the topmost integer from the stack and stores it in the register. If the stack for example looks like this

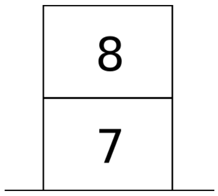


the register contains any integer, and the machine executes the instruction `STORE`, the stack looks like this afterwards:

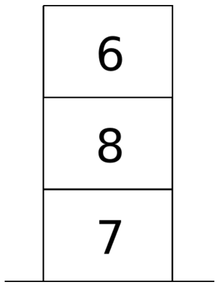


and the register contains the integer 3.

LOAD: This instruction copies the content of the register and pushes it onto the stack. If the register for example contains the integer 6, the stack looks like this

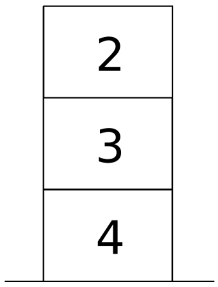


and the machine executes the instruction `LOAD`, the stack looks like this afterwards

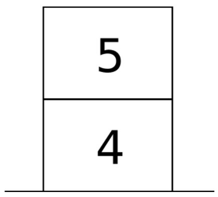


and the register still contains the integer 6.

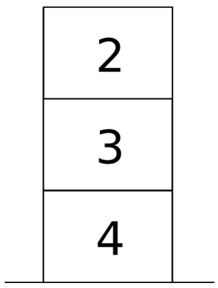
PLUS: This instruction takes the two topmost integers from the stack, adds them, and pushes the resulting integer back onto the stack. If the stack for example looks like this



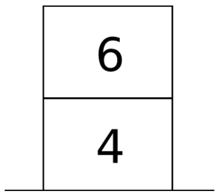
and the machine executes the instruction `PLUS`, the stack looks like this afterwards:



TIMES: This instruction takes the two topmost integers from the stack, multiplies them, and pushes the resulting integer back onto the stack. If the stack for example looks like this



and the machine executes the instruction `TIMES`, the stack looks like this afterwards:



`IFZERO n`: This instruction removes the topmost integer from the stack, and checks if it is equal to 0. If that is the case, it jumps to the n^{th} instruction (start counting at 0). If not, the machine continues as usual with the next instruction. See example below.

`DONE`: Finally, the `DONE` instruction prints out the integer on top of the stack, and terminates the program regardless of the following instructions.

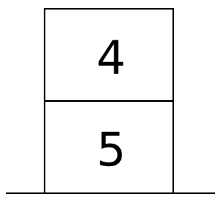
Computation starts with the first instruction. Initially, the stack is empty and the register contains the number 0.

Example

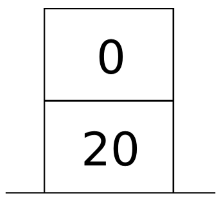
```
0: PUSH 5
1: STORE
2: LOAD
3: LOAD
4: PUSH -1
5: PLUS
6: STORE
7: LOAD
8: IFZERO 13
9: LOAD
10: TIMES
11: PUSH 0
12: IFZERO 3
13: DONE
```

Note that the line numbers are added for illustration; they are not part of the program.

After executing the first 8 instructions, the stack looks like this:



and the register contains the integer 4. The next instruction `IFZERO 12` removes the integer 4 from the stack. Since 4 is not equal to 0, the instructions `LOAD`, `TIMES` and `PUSH 0` are executed next, resulting in the stack:



The next instruction `IFZERO 3` removes 0 from the stack and jumps to second `LOAD` instruction (address `3`), since 0 is equal to 0. Eventually, this program computes $5 \times 4 \times 3 \times 2 \times 1$ and prints out the resulting integer 120.

You can assume that the given program is correct; there will be no infinite loops, and the program will never attempt to remove an integer from an empty stack. You can assume that the stack does not grow bigger than 100, that the argument of `PUSH` is an integer not smaller than -10000 and not larger than 10000, and that the argument of `ISZERO` is an integer not smaller than 0 and not larger than 1000.

Input File: `LVM.IN`

The first line of the input contains an integer n between 2 and 1000, indicating the number of instructions of the program. The following n lines contain the program instructions. Arguments are separated by a space character from the preceding instruction.

Example

```
14
PUSH 5
STORE
LOAD
LOAD
PUSH -1
PLUS
STORE
LOAD
IFZERO 13
LOAD
```

```
TIMES
PUSH 0
IFZERO 3
DONE
```

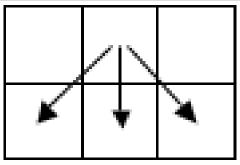
Output File: `LVM.OUT`

The output file consists of a single integer, which is the result of the first `DONE` instruction encountered during execution.

Example

```
120
```

Figure 1 When moving from one tile to a tile in the next lower row, the gecko can only move vertically down or diagonally to the left or right.



TASK 3: GECKO

During the rainy season, one of the walls in the house is infested with mosquitoes. The wall is covered by $h \times w$ square tiles, where there are h rows of tiles from top to bottom, and w columns of tiles from left to right. Each tile has 1 to 1000 mosquitoes resting on it.

A gecko wants to eat as many mosquitoes as possible, subject to the following restrictions. It starts by choosing any tile in the top row, and eats the mosquitoes in that tile. Then, it moves to a tile in the next lower row, eats the mosquitoes on the tile, and so on until it reaches the floor. When it moves from one tile to a tile in the next lower row, it can only move vertically down or diagonally to the left or right (see Figure 1).

Given the values of h and w, and the number of mosquitoes resting on each tile, write a program to compute the maximum possible number of mosquitoes the gecko can eat in one single trip from the top to the bottom of the wall.

Input File: `GECKO.IN`

The first line has two integers. The first integer $h \in \{1, 2, \ldots, 500\}$ is the number of rows of tiles on the wall. The second integer $w \in \{1, 2, \ldots, 500\}$ is the number of columns of tiles on the wall.

Next, there are h lines of inputs. The i^{th} line specifies the number of mosquitoes in each tile of the top i^{th} row. Each line has w integers, where each integer $m \in \{1, 2, \ldots, 1000\}$ is the number mosquitoes on that tile. The integers are separated by a space character.

Example

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

Output File: `GECKO.OUT`

The output file contains a single integer, which is the maximum possible number of mosquitoes the gecko can eat in one single trip from the top to the bottom of the wall.

Example

```
32
```

TASK 4: HOUSING

For the Youth Olympic Games in Singapore, the administration is considering to house each team in several units with at least 5 people per unit. A team can have from 5 to 100 members, depending on the sport they do. For example, if there are 16 team members, there are 6 ways to distribute the team members into units: (1) one unit with 16 team members; (2) two units with 5 and 11 team members, respectively; (3) two units with 6 and 10 team members, respectively; (4) two units with 7 and 9 team members, respectively; (5) two units with 8 team members each; (6) two units with 5 team members each plus a third unit with 6 team members. This list might become quite lengthy for a large team size.

In order to see how many choices to distribute the team members there are, the administration would like to have a computer program that computes for a number n the number $m(n)$ of possible ways to distribute the team members into the units allocated, with at least 5 people per unit. Note that equivalent distributions like $5+5+6$, $5+6+5$ and $6+5+5$ are counted only once. So $m(16) = 6$ (as seen above), $m(17) = 7$ (namely $17, 5+12, 6+11, 7+10, 8+9, 5+5+7, 5+6+6$) and $m(20) = 13$.

The computer program should read the number n and compute $m(n)$.

Input File: `HOUSING.IN`

The file contains just one number which is the number n as described above, where $5 \leq n \leq 100$. A sample file is the following.

Example

```
20
```

Output File: `HOUSING.OUT`

The output file consists of a single integer that is the number $m(n)$ as specified above. As n is at most 100, one can estimate that $m(n)$ has at most 7 decimal digits. A sample file (for the input $n = 20$) is the following.

Example

```
13
```

TASK 5: RANK

In a new sports discipline called Triball proposed for the Youth Olympic Games in Singapore, a set of k players $\{1, 2, \ldots, k\}$ are involved (where $k < 1000$). In every match, 3 players are selected and the result of the match is either 1 winner or 1 loser. Given the result of n matches (where $n < 10000$), our task is to give a ranking list of the players.

Precisely, the i^{th} match involves three distinct players $p_{i1}, p_{i2}, p_{i3} \in \{1, 2, \ldots, k\}$. The result of the i^{th} match is represented by $p_{i1} > p_{i2}, p_{i3}$ (or $p_{i2}, p_{i3} > p_{i1}$) where p_{i1} is the winner (or loser) of the match. The result $p_{i1} > p_{i2}, p_{i3}$ implies that p_{i1} is better than p_{i2} and that p_{i1} is better than p_{i3}. Similarly, the result $p_{i2}, p_{i3} > p_{i1}$ implies that p_{i2} is better than p_{i1} and that p_{i3} is better than p_{i1}. Our aim is to find the permutation of the k players so that a better player always comes before his opponent, considering all given matches. If we cannot reach our aim and such a permutation does not exist, we output 0.

Example 1

Consider 6 players $\{1, 2, 3, 4, 5, 6\}$. Suppose the results of 5 matches are: (1) $4 > 1, 3$, (2) $3 > 1, 2$, (3) $2, 5 > 6$, (4) $5 > 2, 3$, and (5) $1 > 2, 6$. Then, a valid ranking of the players is $4 > 5 > 3 > 1 > 2 > 6$.

Example 2

Consider 4 players $\{1, 2, 3, 4\}$. Suppose the results of 2 matches are: (1) $1 > 2, 4$ and (2) $2, 4 > 1$. Then, there is no valid ranking of the players.

Input File: `RANK.IN`

The input file `RANK.IN` is as follows. The first line contains two integers k and n separated by a space character, where $1 \leq k \leq 1000, 1 \leq n \leq 10000$. The number k is the number of players, denoted by numbers from 1 to k. The remaining n lines contain n results in the form of either $p_{i1} > p_{i2}, p_{i3}$ or $p_{i2}, p_{i3} > p_{i1}$.

For Example 1, the input file looks like this:

```
6 5
4>1,3
3>1,2
2,5>6
5>2,3
1>2,6
```

For Example 2, the input file looks like this:

```
4 2
1>2,4
2,4>1
```

Output File: `RANK.OUT`

If there is a valid ranking of the players, the output file `RANK.OUT` contains k integers, separated by a space character, representing the ordering of the players. Otherwise, the output file `RANK.OUT` contains one integer 0. For Example 1, the output file looks like this:

```
4 5 3 1 2 6
```

For Example 2, the output file looks like this:

```
0
```

TASK 6: 4SUM

In this task, you are provided with four sets of integers. Your task consists of selecting one integer from each set, such that their sum is 0. You can assume that exactly one such selection exists.

Input File: `4SUM.IN`

The first line of input file `4SUM.IN` contains four numbers, a, b, c, and d, separated by a space character, indicating the number of elements in each of the four sets. Each of these numbers is a positive integer $1 \leq a, b, c, d \leq 500$. The following $a + b + c + d$ lines contain the elements, each not smaller than -10000 and not larger than 10000. The elements of the first set are listed first, followed by the elements of the second set, etc.

Example

```
3 2 4 2
5
17
-8
-13
19
6
-9
10
0
-14
7
```

Output File: `4SUM.OUT`

The output file `4SUM.OUT` consists of the four integers, separated by a space character. The numbers must appear in the order in which they are listed in the input file.

Example

The output file for the input above consists of the numbers:

```
17 -13 10 -14
```