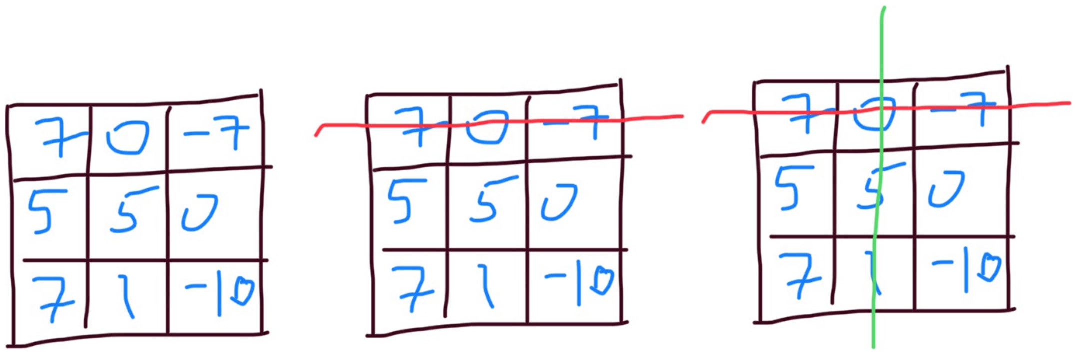
Board Game

Eto and Luna are playing a board game for money. The game is player on an NxN board filled with numbers. In each round Eto picks a row and crosses it off, then Luna picks a column and crosses it off. The value at the intersection of their row and column is Eto's *winnings*. If the value is positive, Luna has to pay that amount of money to Eto. If the value is negative, Eto has to pay that amount of money to Luna.



What is the largest amount of money that Eto can win (or, the least amount that she can lose, if she cannot win)?

Input

The first line of the input contains a single integer N (1 <= N <= 8), representing the size of the board. The following N lines describes the game board M. Each of those lines contains N integers $M_{i,j}$ (-1000 <= $M_{i,j}$ <= 1000), indicating the number at the i-th row and j-th column.

Output

Print one line containing the largest possible score after the game.

Example 1

```
Input:
2
10 10
-5 -5

Output:
5
```

Explanation:

Eto crosses out first row, and Luna crosses out first column. The value of $M_{i,j}$ is 10. So Eto gains $10.00.

Eto crosses out second row, and Luna crosses out second column (because there are no other options). The value of $M_{i,j}$ is -5. So Eto has to pay $5.00 to Luna.

The balance of Eto's winnings is $5.00.

Example 2

```
Input:
2
10 -5
10 -5

Output:
5
```

Example 3

```
Input:
2
10 -5
-5 10

Output:
-10
```

Desolate Number

Given two integers A and B, a desolate number N is defined as follows:

- N is an (A+B)-bit integer;
- the binary representation of N has exactly A 1's and B 0's (leading zeroes are ok);
- N has the maximum number of 1's adjacent to at least one 0 in its binary representation.

Your task is to find the smallest desolate number up to given A and B.

Input

The input consists of a single line, containing two non-negative integers A and B (1 <= A + B <= 50).

Output

Print one line, containing the smallest desolate number.

Example 1

```
Input:
4 3

Output:
45
```

Example 2

```
Input:
1 1

Output:
1
```

Example 3

```
Input:
3 2

Output:
13
```

Move Union Find

In this problem you are to implement a variant of Union Find: Move Union Find.

The data structure you need to write is also a collection of disjoint sets, supporting 3 operations:

operation	description
`1 a b`	Union the sets containing `a` and `b`. If `a` and `b` are already in the same set, ignore this command
`2 a b`	Move the element `a` to the set containing `b`. If `a` and `b` are already in the same set, ignore this command
`3 a`	Return the number of elements and the sum of elements in the set containing `a`

Initially, the collection contains n sets: {1}, {2}, {3}, …, {n}

Input

The first line contains two integer N and M. N is the total number of sets initially, M is the number of commands. 1 <= N, M <= 100,000

Each of the next m lines contain a command. For each operation, 1 <= a,b <= N.

Output

For each type-3 command, output 2 integers: the number of elements and the sum of elements.

Example 1

```
Input:
5 4
1 1 2
2 3 4
1 3 5
2 4 1
3 1

Output:
3 7
```

Explanation:

Initial sets: {1}, {2}, {3}, {4}, {5}

After `1 1 2` : {1, 2}, {3}, {4}, {5}

After `2 3 4` : {1, 2}, {3, 4}, {5}

After `1 3 5}` : {1, 2}, {3, 4, 5}

After `2 4 1` : {1, 2, 4}, {3, 5}

Example 2

```
Input:
5 6
1 1 2
2 3 2
3 3
2 1 4
2 2 4
3 3

Output:
3 6
1 3
```

Follow the Path

You are programming a new robot called Frank to navigate the path through a grid. Frank should be able to adjust to direction it is moving in rather than being pre-programmed. This particular robot needs to navigate through a 2D grid. At each grid location Frank receives the instructions for the instructions for its next move. The possible moves are:

- N - move North (or up)
- S - move South (or down)
- E - move East (or right)
- W - move West (or left)

Your task now is to calculate what path Frank should take and verify that it actually took that path (otherwise you have a bug in the code that directs Frank through the grid).

Example 1:

```
                F
    N   E   E   S<-W   E
                v
  <-W<-W   W   E->S   S
        ^           v
    S   N<-W<-W<-W   W
```

If Frank enters this grid at the grid location just below of the position of F, it will move west, then south, then east, then south, then west three times, then north and west two more times and finally it will exit the grid.

Example 2:

```
    F
    S   E   S<-W   E
    v       v   ^
    E->E->S   N<-W
            v       ^
    N   W   E->E->N

    N   W   S   W   N
```

If Frank enters this grid at the grid location just below of the position of F, it will move south, then east twice, and then it will enter a loop consisting of 8 moves. Frank will never leave this grid.

Input On the first line are three integers separated by blanks: the number of rows in the grid, the number of columns in the grid, and the number of the column in which the robot enters from the north. The possible entry columns are numbered starting with one at the left.

Then come the rows of the direction instructions. Each grid will have at least one and at most 10 rows and columns of instructions. The lines of instructions contain only the characters N, S, E, or W with no blanks.

Output

There should be one line of output. Either the robot follows a certain number of instructions and exits the grid on any one the four sides or else the robot follows the instructions on a certain number of locations once, and then the instructions on some number of locations repeatedly. The sample input below corresponds to the two grids above and illustrates the two forms of output. The word “step” is always immediately followed by “(s)” whether or not the number before it is 1.

Example 1

```
Input:

3 6 5
NEESWE
WWWESS
SNWWWW


Output:

10 step(s) to exit
```

Example 2

```
Input:

4 5 1
SESWE
EESNW
NWEEN
EWSEN

Output:

3 step(s) before a loop of 8 step(s)
```

Yet Another Sorting Problem (YASP)

"It is lovely day for pancakes." Kou thought.

As she said that, Kou found yet another sorting problem (yasp). In this problem, we are given a *stack* of integers and we are asked to sort that stack in ascending order from top to bottom. The only allowed operation, denoted by *flip(t)*, is to flip all elements from the *t*-th elements to the top upside down. Note that the 1st element is the bottom element while the N-th element refers to the top element if the size of the stack is N. For example, consider the following four stacks of integers.

```
          4       2       5       1       (top)
          3       3       4       2
          2       4       3       3
          5       5       2       4
          1       1       1       5
  (bottom)
         (a)     (b)     (c)     (d)
```

Performing *flip(3)* on stack (a) will result in stack (b). Similarly, by applying *flip(2)*, stack (b) will become stack (c). Finally, we can apply *flip(1)* on stack (c) and get the stack sorted as shown in stack (d).

We need to figure out a sequence of *flip* operations that transforms the given stack into a sorted one.

Input

The input consists of a single line. The stack of integers will be given in this line from top to bottom. The size of the stack is between 1 and 30, and those integers are between 1 and 100.

Output

Print one line containing (K+1) flip operations:

- Among the first K integers, the i-th integer specifies the *t* value of the i-th flip operation.
- The last, i.e., the (K+1)-th integer should always be `0` .

If there are multiple solutions, output any of them.

Example 1

```
Input:
1 2 3 4 5

Output:
0
```

Example 2

```
Input:
5 4 3 2 1

Output:
1 0
```

Example 3

```
Input:
5 1 2 3 4

Output:
1 2 0
```