

2010

SPOJ Solved Problems

This document is my attempt to list down solutions to some of the interesting problems and algorithms that are available on Sphere Online Judge (<http://www.spoj.pl/>). It has over 1200 problems and an ideal platform to sharpen one's understanding of algorithms and their application to problem solving. I work on these problems in my leisure. I have listed these solutions for my own reference in future. All the solutions that I have accumulated in this document are my own and need not be the most optimized or the best solutions. However, I have taken care of the fact that each of these solutions have been tested and "accepted" at SPOJ before incorporating them here.

I hope that going through these interesting problems would intrigue the readers and encourage them to participate in these competitive sites. I have provided both C++ and Java implementations of the solutions. Any suggestions, criticisms and better algorithms are more than welcome. Please note: it is futile to "Google" for solutions to SPOJ problems because they are rare; even we are not allowed to see the solutions submitted by others in SPOJ.



1. CANDY.....	5
2. ROTATING RINGS	10
3. ARRANGING AMPLIFIERS	18
4. HAPPY COINS.....	21
5. A CONCRETE SIMULATION	24
6. MUSICAL CHAIRS	28
7. ADJACENT BIT COUNTS	31
8. ROBBERY 2.....	34
9. PUBLISH OR PERISH.....	38
10. ZIG-ZAG RABBIT	41
11. DINOSTRATUS NUMBERS.....	48
12. ANOTHER GAME WITH NUMBERS.....	54
13. POLYNOMIAL	60
14. INSTRUCTION DECODER.....	65
15. SEINFELD.....	70
16. PICKING UP CHICKS	74
17. FEANOR THE ELF.....	78
18. USE OF FUNCTION ARCTAN	82
19. HEADSHOT	86
20. FRACTIONS ON TREE	90
21. TWO PROFESSORS	94
22. TRAVELLING SHOEMAKER.....	99
23. CROSS COUNTRY.....	113
24. THE CLOCKS	117
25. GARBAGE	121
25. SETNJA.....	124
26. HARRY POTTER.....	130
27. LISA.....	134
29. TEMPTATION ISLAND.....	138
30. BLACK OR WHITE	142
31. ROUND TABLE	148
32. RECTANGLES PERIMETER.....	153
33. COINS GAME	156

34.	ALIGNING STRINGS	159
35.	CARD SORTING.....	162

Problem Classification

Problem	AlgorithmType
CANDY	HARD DYNAMIC PROGRAMMING
ROTATING RINGS	EFFICIENT SIMULATION
ARRANGING AMPLIFIERS	ADHOC, SORTING
HAPPY COINS	PATTERN FINDING
A CONCRETE SIMULATION	ADHOC, EFFICIENT SIMULATION
MUSICAL CHAIRS	EASY DYNAMIC PROGRAMMING
ADJACENT BIT COUNTS	EASY DYNAMIC PROGRAMMING
ROBBERY 2	ADHOC, MATHS
PUBLISH OR PERISH	ADHOC, SORTING
ZIG-ZAG RABBIT	ADHOC, MATHS
DINOSTRATUS NUMBERS	MATHS, PRIME FACTORIZATION
ANOTHER GAME WITH NUMBERS	MATHS, COMBINATORICS
POLYNOMIAL	MATHS
INSTRUCTION DECODER	ADHOC
SEINFELD	GREEDY
PICKING UP CHICKS	GREEDY
FEANOR THE ELF	ADHOC, PHYSICS, PROJECTILE MOTION
USE OF FUNCTION ARCTAN	ADHOC, MATHS
HEADSHOT	ADHOC
FRACTIONS ON TREE	ADHOC
TWO PROFESSORS	SCHEDULING
TRAVELLING SHOEMAKER	GRAPH THEORY, EULERIAN & HAMILTONIAN PATH
CROSS COUNTRY	DYNAMIC PROGRAMMING
THE CLOCKS	ADHOC
GARBAGE	GRAPH THEORY, HAMILTONIAN CIRCUIT
SETNJA	ADHOC, BIG INTEGER MATHS
HARRY POTTER	EASY DYNAMIC PROGRAMMING
LISA	EASY DYNAMIC PROGRAMMING
TEMPTATION ISLAND	EASY DYNAMIC PROGRAMMING
BLACK OR WHITE	HARD DYNAMIC PROGRAMMING
ROUND TABLE	VERY HARD DYNAMIC PROGRAMMING
RECTANGLES PERIMETER	MODERATE DYNAMIC PROGRAMMING
COINS GAME	EASY DYNAMIC PROGRAMMING
ALIGNING STRINGS	MODERATE DYNAMIC PROGRAMMING
CARD SORTING	MODERATE DYNAMIC PROGRAMMING

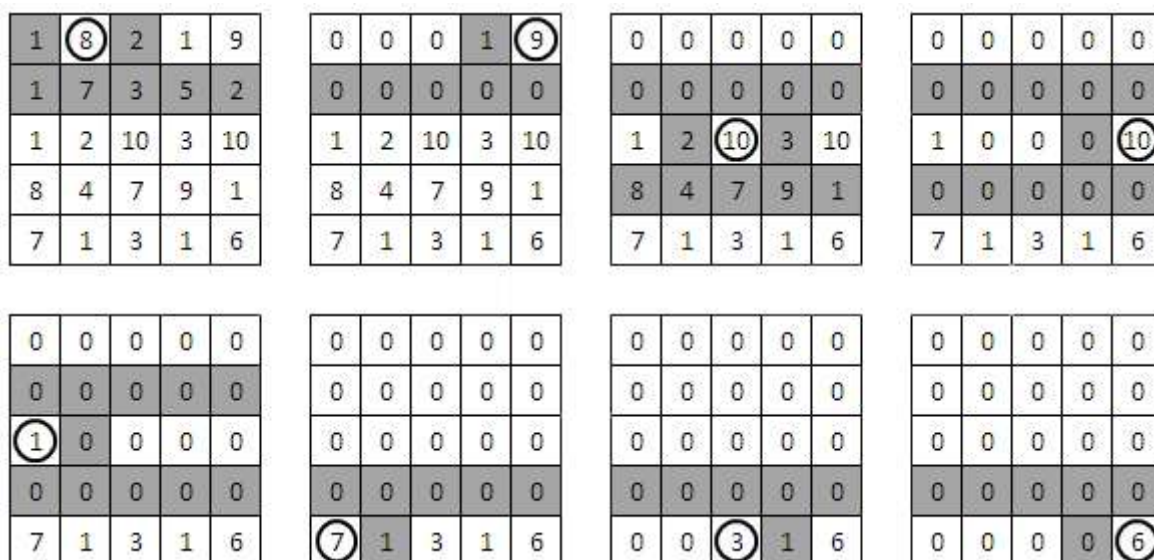
1. Candy

Little Charlie is a nice boy addicted to candies. He is even a subscriber to All Candies Magazine and was selected to participate in the International Candy Picking Contest.

In this contest a random number of boxes containing candies are disposed in M rows with N columns each (so, there are a total of $M \times N$ boxes). Each box has a number indicating how many candies it contains.

The contestant can pick a box (any one) and get all the candies it contains. But there is a catch (there is always a catch): when choosing a box, all the boxes from the rows immediately above and immediately below are emptied, as well as the box to the left and the box to the right of the chosen box. The contestant continues to pick a box until there are no candies left.

The figure bellow illustrates this, step by step. Each cell represents one box and the number of candies it contains. At each step, the chosen box is circled and the shaded cells represent the boxes that will be emptied. After eight steps the game is over and Charlie picked $10+9+8+3+7+6+10+1 = 54$ candies.



For small values of M and N , Charlie can easily find the maximum number of candies he can pick, but when the numbers are really large he gets completely lost. Can you help Charlie maximize the number of candies he can pick?

Input

The input contains several test cases. The first line of a test case contains two positive integers M and N ($1 \leq M \times N \leq 10^5$), separated by a single space, indicating the number of rows and columns respectively. Each of the following M lines contains N integers separated by single spaces, each representing the initial number of candies in the corresponding box. Each box will have initially at least 1 and at most 10^3 candies. The end of input is indicated by a line containing two zeroes separated by a single space.

Output

For each test case in the input, your program must print a single line, containing a single value, the integer indicating the maximum number of candies that Charlie can pick.

Example

Input:

5 5

1 8 2 1 9
1 7 3 5 2
1 2 10 3 10
8 4 7 9 1
7 1 3 1 6
4 4
10 1 1 10
1 1 1 1
1 1 1 1
10 1 1 10
2 4
9 10 2 7
5 1 1 5
0 0

Output:

54
40
17

Time limit: 2s
Source limit: 50000B

Solution:

This is an elegant 'Dynamic Programing' problem. For the time being let us forget the matrix and concentrate on a single row. Let us take the example provided in the problem.

1	8	2	1	9
1	7	3	5	2
1	2	10	3	10
8	4	7	9	1
7	1	3	1	6

The matrix has 5 rows and 5 columns. Let us consider the first row and try to find out what is the maximum number of candies Little charlie collect from the first row.

Observing the first row, we see that if Little Charlie collects the candy from the last cell

1	8	2	1	9
---	---	---	---	---

 which is '9', he cannot collect candy from its immediate left cell which is '1'. Now, if we assume that Little Charlie has come to the last cell after collecting all the candies from the left, and the dimension of the row is 'N', then the maximum number of candies that he can collect from the entire row is = Max candies collected by Little Charlie till cell N - 2 + the candy in the Nth cell.Case (1a)

We also observe that if Little Charlie collects the candy from the last but one cell which is '1', he cannot collect candies from its immediate left and right cells which are '2' and '9' respectively. Hence, the maximum number of candies that he can collect from the entire row is = Max candies collected by Little Charlie till cell N - 1th cell, which include last but one cell ('1') also

.....Case(1b)

The base case is, when Little Charlie is at Row[0] cell he can collect only 1 candy or Row1[0]. When Little Charlie is at Row1[1] he can collect 1 or 8 i.e. Maximum(Row1[0], Row1[1]).

Hence, in the best case, the maximum number of candies collected by Little Charlie should be the maximum of above two cases:

Max Candies collected in a row of dimension N is,
Maximum of (candies collected till $N - 2^{\text{th}}$ cell+ the candy in the N^{th} cell , candies collected till $N - 1^{\text{th}}$ cell)

Row Total					
1	8	2	1	9	17
1	7	3	5	2	0
1	2	10	3	10	21
8	4	7	9	1	0
7	1	3	1	6	16
Total Candies =					54

Now, in this way if we calculate the max candies in each row as shown on the left, an interesting pattern to observe is that the totals for each row is also following the same pattern that we have observed for cells in each row. This is because, when Little Charlie reaches a cell, he cannot collect its immediate left and right cells. In addition to that he cannot also collect the immediate upper and lower row. Hence, that same pattern is also reflected in the totals of the rows.

Row Total					
1	8	2	1	9	17
1	7	3	5	2	0
1	2	10	3	10	21
8	4	7	9	1	0
7	1	3	1	6	16
Total Candies =					54

If Little Charlie collects the maximum candies from the last row which is '16', he cannot collect candies from its immediate upper row which has a total of '0' maximum candies.

Now, if we assume that Little Charlie has come to the last row after collecting all the candies from rows above, and the dimension of the entire matrix is ' $M \times N$ ', then the maximum number of candies that he can collect from the entire grid is = Max candies collected by Little Charlie till row $M - 2$ + maximum candies collected in the M^{th} row.

....Case(2a)

Row Total					
1	8	2	1	9	17
1	7	3	5	2	0
1	2	10	3	10	21
8	4	7	9	1	0
7	1	3	1	6	16
Total Candies =					54

We also observe that if Little Charlie collects the candy from the last but one row which is '0', he cannot collect candies from its immediate above and below rows which are '21' and

'16' respectively. Hence, the maximum number of candies that he can collect from the entire row is = Max candies collected by Little Charlie till cell $M - 1^{\text{th}}$ row, which include last but one row ('0') also.

....Case(2b)

Hence we can extend the recursive relation as

Max Candies collected in a matrix of dimension $M \times N$ is,
Maximum of (candies collected till $M - 2^{\text{th}}$ row+ candies collected in the M^{th} row , candies collected till $M - 1^{\text{th}}$ cell)

The base case is, when Little Charlie is at AllRows[0] cell he can collect only 17 candy or AllRows[0]. When Little Charlie is at AllRows[1] he can collect 17 or 0 i.e. Maximum(AllRows[0], AllRows[1])

Source Code(C++):

```
#include <iostream>
#define MAX 100000
// store the maximum amount of candy for each cell in a row
int rowWiseCandy[MAX];
// store the maximum amount of candy for each row
int maxCandyForRow[MAX];
using namespace std;

class Candy {
public:
    static void main() {
        int M, N;

        while(1){
            scanf("%d %d",&M,&N);

            if(M == 0 && N == 0)
                break;

            for(int i = 0; i < M; i++){
                for(int j = 0; j < N; j++){
                    scanf("%d",&rowWiseCandy[j]);
                }

                // the recursive relation is
                // rowWiseCandy[j] = max(rowWiseCandy[j - 2] + rowWiseCandy[j], rowWiseCandy[j - 1])
                rowWiseCandy[1] = max(rowWiseCandy[0], rowWiseCandy[1]);
                for(int j = 2; j < N; j++){
                    rowWiseCandy[j] = max(rowWiseCandy[j - 2] + rowWiseCandy[j], rowWiseCandy[j - 1]);
                }

                maxCandyForRow[i] = rowWiseCandy[N - 1];
            }

            // the recursive relation is
            // maxCandyForRow[j] = max(maxCandyForRow[j - 2] + rowWiseCandy[j], maxCandyForRow[j - 1])
            maxCandyForRow[1] = max(maxCandyForRow[0], maxCandyForRow[1]);
            for(int i = 2; i < M; i++){
                maxCandyForRow[i] = max(maxCandyForRow[i - 2] + maxCandyForRow[i], maxCandyForRow[i - 1]);
            }

            printf("%d\n",maxCandyForRow[M - 1]);
        }
    }
};

int main() {
    Candy::main();
    return 0;
}
```

Source Code(Java):

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.StringTokenizer;

public class Candy {
    private static int MAX = 100000;

    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        while (true) {

            StringTokenizer st = new StringTokenizer(br.readLine(), " ");

            int M = Integer.parseInt(st.nextToken());
            int N = Integer.parseInt(st.nextToken());
```



```

if (M == 0 && N == 0)
    break;

// store the maximum amount of candy for each cell in a row
int rowWiseCandy[] = new int[MAX + 1];
// store the maximum amount of candy for each row
int maxCandyForRow[] = new int[MAX + 1];

for (int i = 0; i < M; i++) {
    st = new StringTokenizer(br.readLine(), " ");

    for (int j = 0; j < N; j++){
        rowWiseCandy[j] = Integer.parseInt(st.nextToken());
    }

    // the recursive relation is
    // rowWiseCandy[j] = max(rowWiseCandy[j - 2] + rowWiseCandy[j], rowWiseCandy[j - 1])
    rowWiseCandy[1] = Math.max(rowWiseCandy[0], rowWiseCandy[1]);
    for (int j = 2; j < N; j++){
        rowWiseCandy[j] = Math.max(rowWiseCandy[j - 2] + rowWiseCandy[j], rowWiseCandy[j - 1]);
    }

    maxCandyForRow[i] = rowWiseCandy[N - 1];
}

// the recursive relation is
// maxCandyForRow[j] = max(maxCandyForRow[j - 2] + rowWiseCandy[j], maxCandyForRow[j - 1])
maxCandyForRow[1] = Math.max(maxCandyForRow[0], maxCandyForRow[1]);
for (int i = 2; i < M; i++){
    maxCandyForRow[i] = Math.max(maxCandyForRow[i - 2] + maxCandyForRow[i], maxCandyForRow[i - 1]);
}

System.out.println(maxCandyForRow[M - 1]);
}
}
}

```

2. Rotating Rings

Any square grid can be viewed as one or more rings, one inside the other. For example, as shown in figure(a), a 5 x 5 grid is made of three rings, numbered 1,2 and 3 (from outside to inside.) A square grid of size N is said to be sorted, if it includes the values from 1 to N^2 in a row-major order, as shown in figure(b) for $N = 4$. We would like to determine if a given square grid can be sorted by only rotating its rings. For example, the grid in figure(c) can be sorted by rotating the first ring two places counter-clockwise, and rotating the second ring one place in the clockwise direction.

1	1	1	1	1
1	2	2	2	1
1	2	3	2	1
1	2	2	2	1
1	1	1	1	1

Figure (a)

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Figure (b)

9	5	1	2
13	7	11	3
14	6	10	4
15	16	12	8

Figure (c)

Input

Your program will be tested on one or more test cases. The first input line of a test case is an integer N which is the size of the grid. N input lines will follow, each line made of N integer values specifying the values in the grid in a row-major order. Note that $0 < N \leq 1,000$ and grid values are natural numbers less than or equal to 1,000,000.

The end of the test cases is identified with a dummy test case with $N = 0$.

Output

For each test case, output the result on a single line using the following format:

k. result

where k is the test case number (starting at 1,) and result is "YES" or "NO" (without the double quotes.) and a single space between "." and "result".

Sample

Input

```
4
9 5 1 2
13 7 11 3
14 6 10 4
15 16 12 8
3
1 2 3
5 6 7
8 9 4
2
1 2
4 3
0
```

Output

```
1. YES
2. NO
3. NO
```

Time Limit: 1s-2s
Source Limit: 50000B

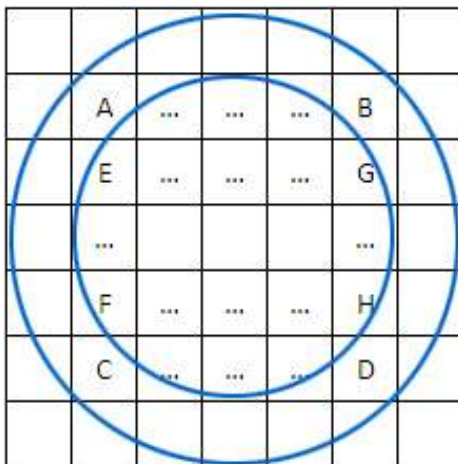
Solution:

A grid can be sorted if and only if the 2 conditions are satisfied

- The numbers that are destined for a particular ring remain in their rings and nowhere else. For example in test case 2, 5 is in outer ring at position grid[2][1] but it is destined for the inner ring at position grid[2][2] in the sorted sequence. By no means either by counter clockwise or clockwise rotation we can move 5 to grid[2][2]. So, this cannot be sorted. Test case 1 satisfies this condition so it can be sorted.
- The second condition is that the sequence starting from the first number of that ring must be preserved. For example for a grid of $N = 4$, the outer rings sequence is 1 2 3 4 8 12 16 15 14 13 9 5. If in any grid configuration if this sequence is not preserved then the grid cannot be sorted. In test case 3 the sequence is 1 2 3 4, but it should have been 1 2 4 3 in order to get sorted. So it cannot be sorted by any means.

So, to check for the first condition we calculate the sum of number of each ring of a grid and compare it with the sum for that particular ring(r) which is $2(n^2 + 1)(n - 2r + 1)$ [You need to derive this, where ring starts from 1].

For the second condition you need to expand the ring in a 1D array, search for the starting number for that ring which is $\{r(n + 1) - n\}$ [You need to derive this, where ring starts from 1] and traverse through the array to check whether the sequence is maintained e.g. for $N = 4$, check whether the common difference between the successive terms is 1, 1, 1, 1, 4, 4, 4, 4, -1, -1, -1, -1, -4, -4, -4, -4



Dimension of a ring(r):

Let us assume a matrix of dimension ' n '.

So, when $r = 1$, dimension of ring = n

when $r = 2$, dimension of ring = $n - 2 = n - 2(2 - 1)$

when $r = 3$, dimension of ring = $n - 4 = n - 2(3 - 1)$

Hence for any ' r ', dimension of ring $r = n - 2(r - 1)$ (1)

Value of first element of a ring(r):

when $r = 1$, value of first element of ring = 1

when $r = 2$, value of first element of ring = $(n + 1) + 1$

when $r = 3$, value of first element of ring = $(n + n + 2) + 1 = 2(n + 1) + 1$

Hence, for any ' r ', value of first element of ring = $(r - 1)(n + 1) + 1 = r(n + 1) - n$ (2)

Sum of all elements of a ring(r):

Refer to the figure above; let us consider a ring ' r ' which has 4 corner elements viz. A, B, C and D.

From (2), $A = r(n + 1) - n$

From (1), number of elements on the top row = $n - 2(r - 1)$

Hence, following AP series formula for any term $\{a + (n - 1)d\}$, we get

$$B = [r(n + 1) - n] + [n - 2(r - 1) - 1] \times 1$$

$$= rn - r + 1$$

Similarly, From (1) and following AP series formula for any term $\{a + (n - 1)d\}$, we get

$$\begin{aligned} C &= A + [n - 2(r - 1) - 1] \times n \\ &= [r(n + 1) - n] + [n - 2(r - 1) - 1]n \\ &= n^2 - rn + r \end{aligned}$$

and

$$\begin{aligned} D &= B + [n - 2(r - 1) - 1] \times n \\ &= [r(n + 1) - n] + [n - 2(r - 1) - 1]n \\ &= n^2 - rn + n - r + 1 \end{aligned}$$

$$\begin{aligned} \text{Hence, top row + bottom row} &= \frac{(A+B)(\text{No. of terms})}{2} + \frac{(C+D)(\text{No. of terms})}{2} = \frac{\text{No. of terms}}{2}(A + B + C + D) \\ &= \frac{(n - 2(r - 1))}{2}(rn + r - n + rn - r + 1 + n^2 - rn + r + n^2 - rn + n - r + 1) \\ &= \frac{(n - 2r + 2)}{2} 2(n^2 + 1) \\ &= (n - 2r + 2)(n^2 + 1) \end{aligned} \quad \text{..... (3)}$$

Similarly, by above logic,

$$\begin{aligned} E &= A + n = r(n + 1) = rn + r \\ F &= C - n = n^2 - rn + r - n \\ G &= B + n = rn - r + n + 1 \\ H &= D - n = n^2 - rn - r + 1 \end{aligned}$$

Hence, sum of left and right sides of the ring excluding 4 corner elements,

$$\begin{aligned} &\frac{(E + F)(\text{No. of terms})}{2} + \frac{(G + H)(\text{No. of terms})}{2} = \frac{\text{No. of terms}}{2}(E + F + G + H) \\ &= \frac{(n - 2(r - 1) - 2)}{2}(rn + r + n^2 - rn + r - n + rn - r + n + 1 + n^2 - rn - r + 1) \\ &= \frac{(n - 2r)}{2} 2(n^2 + 1) \\ &= (n - 2r)(n^2 + 1) \end{aligned} \quad \text{..... (4)}$$

Adding 3 and 4, we get

$$\begin{aligned} \text{Sum of all elements in a ring} &= (n - 2r + 2)(n^2 + 1) + (n - 2r)(n^2 + 1) \\ &= 2(n^2 + 1)(n - 2r + 1) \end{aligned} \quad \text{..... (5)}$$

Source Code (C++):

```
#include <iostream>
#include <conio.h>
using namespace std;

class RotatingRings {
public:
    static void main() {
        int testCase = 0;
        while (true) {
            // get the dimension of the square matrix
            int n;
            scanf("%d",&n);

            if (n == 0) {
                break;
            }
            testCase++;
            // define the matrix
            int **squareMatrix = new int*[n];
            for(int i = 0; i < n; i++){
                squareMatrix[i] = new int [n];
            }
            // initialize the matrix
            for(int row = 0; row < n; row++){
                for(int col = 0; col < n; col++){
                    squareMatrix[row][col] = 0;
                }
            }
            // get the data from console into the square matrix
            for(int row = 0; row < n; row++){
                for(int col = 0; col < n; col++){
                    scanf("%d",&squareMatrix[row][col]);
                }
            }
            // find the center coordinates of the square matrix
            int center = n / 2;
            // check for dimension if n is odd. The center element of a n(odd)
            // dimension square matrix arranged in a row-major order
            // is always (n ^ 2 + 1) / 2
            if (n % 2 != 0 && squareMatrix[center][center] != (n * n + 1) / 2) {
                printf("%d. NO\n", testCase);
            } else {
                /*
                 * A 5 x 5 matrix
                 * =====
                 * 9 5 1 2
                 * 13 7 11 3
                 * 14 6 10 4
                 * 15 16 12 8
                 */
                bool squareCanBeSorted = true;

                // traverse ring wise
                for (int ring = 1; ring <= center; ring++) {
                    // total sum of elements in a ring is {2(n^2 + 1)(n - 2r + 1)}
                    int expectedSum = 2 * (n * n + 1) * (n - 2 * ring + 1);

                    // total elements in a ring is 4(n - 2r + 1)
                    int noOfElementsInTheRing = 4 * (n - 2 * ring + 1);

                    // dimension of the ring is n - 2(r - 1)
                    int ringDimension = n - 2 * (ring - 1);

                    // find the corners of the ring
                    int leastRow = ring - 1, leastCol = ring - 1;
                    int maxRow = leastRow + (ringDimension - 1), maxCol = leastCol + (ringDimension - 1);

                    int *linearArrayOfRing = new int[noOfElementsInTheRing];
                    int counter = 0;

                    int sum = 0;
                    // traverse the top row of the ring
```

```

for (int row = leastRow, col = leastCol; col <= maxCol; col++) {
    sum = sum + squareMatrix[row][col];
    linearArrayOfRing[counter++] = squareMatrix[row][col];
}
// traverse the right side of the ring
for (int row = leastRow + 1, col = maxCol; row < maxRow; row++) {
    sum = sum + squareMatrix[row][col];
    linearArrayOfRing[counter++] = squareMatrix[row][col];
}
// traverse the bottom row of the ring
for (int col = maxCol, row = maxRow; col >= leastCol; col--) {
    sum = sum + squareMatrix[row][col];
    linearArrayOfRing[counter++] = squareMatrix[row][col];
}
// traverse the left side the ring
for (int row = maxRow - 1, col = leastCol; row > leastRow; row--) {
    sum = sum + squareMatrix[row][col];
    linearArrayOfRing[counter++] = squareMatrix[row][col];
}
// if the calculated sum matches expected sum, then check for sequence
if (sum == expectedSum) {
    // now match the sequence, the first element of the
    // ring is given by  $r(n + 1) - n$ 
    int firstElementValueOftheRing = ring * (n + 1) - n;

    // find the index of the first element in the linear array
    int ringElementIndex = -1;
    for (int i = 0; i < noOfElementsInTheRing; i++) {
        if (linearArrayOfRing[i] == firstElementValueOftheRing) {
            ringElementIndex = i;
            break;
        }
    }
    // no of sequence checks to be made per side of the ring
    // is  $4\{n - 2(r - 1) - 1\}$  i.e.  $4(\text{ringdimension} - 1)$ 
    int noOfChecksToMake = 4 * (ringDimension - 1);
    // check for sequence 1, 1, 1,..., n, n, n,..., -1, -1,
    // -1, -n, -n, -n...
    for (int i = 1, len = noOfElementsInTheRing, commonDiff = 1; i <= noOfChecksToMake; i++) {
        if ((linearArrayOfRing[(ringElementIndex + 1) % len] -
            linearArrayOfRing[ringElementIndex]
            % len) != commonDiff) {
            squareCanBeSorted = false;
            break;
        }
        // find when we need to switch from 1 to n to -1 to -n
        if (i % (ringDimension - 1) == 0) {
            int switchNo = i / (ringDimension - 1);
            switch (switchNo) {
                case 0:
                    commonDiff = 1;
                    break;
                case 1:
                    commonDiff = n;
                    break;
                case 2:
                    commonDiff = -1;
                    break;
                case 3:
                    commonDiff = -n;
                    break;
                default:
                    commonDiff = 1;
            }
        }
        ringElementIndex = (ringElementIndex + 1) % len;
    }
} else {
    squareCanBeSorted = false;
    break;
}
}

// display result

```

```

        if (squareCanBeSorted == false) {
            printf("%d. NO\n", testCase);
        } else {
            printf("%d. YES\n", testCase);
        }
    }
}
};

int main() {
    RotatingRings::main();
    getch();
    return 0;
}

```

Source Code (Java):

```

import java.io.BufferedReader;
import java.io.InputStreamReader;

public class RotatingRings {

    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        int testCase = 0;

        while (true) {
            // get the dimension of the square matrix
            int n = Integer.parseInt(br.readLine());

            if (n == 0) {
                break;
            }
            testCase++;

            // define the matrix
            int squareMatrix[][] = new int[n][n];

            // get the data from console into the square matrix
            for (int i = 0; i < n; i++) {
                String params[] = br.readLine().split("\\s");
                for (int j = 0; j < n; j++) {
                    squareMatrix[i][j] = Integer.parseInt(params[j]);
                }
            }

            // find the center coordinates of the square matrix
            int center = n / 2;

            // check for dimension if n is odd. The center element of a n(odd)
            // dimension square matrix arranged in a row-major order
            // is always (n ^ 2 + 1) / 2
            if (n % 2 != 0 && squareMatrix[center][center] != (n * n + 1) / 2) {
                System.out.printf("%d. NO\n", testCase);
            } else {
                /*
                 * A 5 x 5 matrix
                 * =====
                 * 9    5    1    2
                 * 13   7   11   3
                 * 14   6   10   4
                 * 15  16   12   8
                 */
                boolean squareCanBeSorted = true;

                // traverse ring wise
                for (int ring = 1; ring <= center; ring++) {
                    // total sum of elements in a ring is {2(n^2 + 1)(n - 2r + 1)}
                    int expectedSum = 2 * (n * n + 1) * (n - 2 * ring + 1);

                    // total elements in a ring is 4(n - 2r + 1)
                    int noOfElementsInTheRing = 4 * (n - 2 * ring + 1);

```

```
// dimension of the ring is n - 2(r - 1)
int ringDimension = n - 2 * (ring - 1);

// find the corners of the ring
int leastRow = ring - 1, leastCol = ring - 1;
int maxRow = leastRow + (ringDimension - 1), maxCol = leastCol + (ringDimension - 1);

int linearArrayOfRing[] = new int[noOfElementsInTheRing];
int counter = 0;

int sum = 0;
// traverse the top row of the ring
for (int row = leastRow, col = leastCol; col <= maxCol; col++) {
    sum = sum + squareMatrix[row][col];
    linearArrayOfRing[counter++] = squareMatrix[row][col];
}
// traverse the right side of the ring
for (int row = leastRow + 1, col = maxCol; row < maxRow; row++) {
    sum = sum + squareMatrix[row][col];
    linearArrayOfRing[counter++] = squareMatrix[row][col];
}
// traverse the bottom row of the ring
for (int col = maxCol, row = maxRow; col >= leastCol; col--) {
    sum = sum + squareMatrix[row][col];
    linearArrayOfRing[counter++] = squareMatrix[row][col];
}
// traverse the left side the ring
for (int row = maxRow - 1, col = leastCol; row > leastRow; row--) {
    sum = sum + squareMatrix[row][col];
    linearArrayOfRing[counter++] = squareMatrix[row][col];
}

// if the calculated sum matches expected sum, then check for sequence
if (sum == expectedSum) {
    // now match the sequence, the first element of the
    // ring is given by r(n + 1) - n
    int firstElementValueOftheRing = ring * (n + 1) - n;

    // find the index of the first element in the linear array
    int ringElementIndex = -1;
    for (int i = 0; i < linearArrayOfRing.length; i++) {
        if (linearArrayOfRing[i] == firstElementValueOftheRing) {
            ringElementIndex = i;
            break;
        }
    }
    // no of sequence checks to be made per side of the ring
    // is 4{n - 2(r - 1) - 1} i.e. 4(ringdimension - 1)
    int noOfChecksToMake = 4 * (ringDimension - 1);
    // check for sequence 1, 1, 1,..., n, n, n,..., -1, -1,
    // -1, -n, -n, -n...
    for (int i = 1, len = linearArrayOfRing.length, commonDiff = 1; i <= noOfChecksToMake; i++) {
        if ((linearArrayOfRing[(ringElementIndex + 1) % len] - linearArrayOfRing[(ringElementIndex)
            % len]) != commonDiff) {
            squareCanBeSorted = false;
            break;
        }
    }
    // find when we need to switch from 1 to n to -1 to -n
    if (i % (ringDimension - 1) == 0) {
        int switchNo = i / (ringDimension - 1);
        switch (switchNo) {
            case 0:
                commonDiff = 1;
                break;
            case 1:
                commonDiff = n;
                break;
            case 2:
                commonDiff = -1;
                break;
            case 3:
                commonDiff = -n;
                break;
        }
    }
}
```



```
        default:
            commonDiff = 1;
    }
}
ringElementIndex = (ringElementIndex + 1) % len;
} else {
    squareCanBeSorted = false;
    break;
}
}

// display result
if (squareCanBeSorted == false) {
    System.out.printf("%d. NO\n", testCase);
} else {
    System.out.printf("%d. YES\n", testCase);
}
}
}
}
```

3. Arranging Amplifiers

Scientists at the TIFR, Mumbai, are doing some cutting edge research on the Propagation of Signals. A young researcher comes up with a method of progressively amplifying signals, as they progress along a path. The method involves the placing of Amplifiers at regular distances along the line. Each amplifier is loaded with a number $a(i)$, which is called its amplification factor. The method of amplification is simple: an amplifier which receives a signal of strength X , and has Y loaded in it, results in a signal of strength Y^X [Y to the power X]. In course of his research, the young scientist tries to find out, that given a set of n amplifiers loaded with $a(0)$, $a(1)$, $a(2)$, ..., $a(n-1)$, which particular permutation of these amplifiers, when placed at successive nodes, with the initial node given a signal of strength 1, produces the strongest output signal.

This is better illustrated by the following example : 5 6 4

$4^{(5^{(6^1)})}$ is the strength of the strongest signal, which is generated by putting amplifier loaded with 6 in first place, 5 in second place and 4 in third place.

Given a list of integers specifying the set of amplifiers at hand, you must find out the order in which they must be placed, to get the highest signal strength. In case there exist multiple permutations with same output, you should print the one which has bigger amplifiers first.

Input

First line of input contains T , the number of test cases. For each test case first line contains a number n_i , which is equal to the number of amplifiers available. Next line contains n integers, separated by spaces which denote the values with which the amplifiers are loaded.

Output

Output contains T lines, one for each test case. Each line contains n_i integers, denoting the order in which the amplifiers should be kept such that the result is strongest.

Example

Input:

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

Output:

```
6 5 4
2 3
```

Constraints and Limits

$T \leq 20$, $N_i \leq 10^5$.

Each amplifier will be loaded with a positive integer p , $0 < p \leq 10^9$. No two amplifier > 1 shall be loaded with the same integer.

Time Limit: 5s

Source Limit: 50000B

Solution:

It's a very easy problem. It's just about recognizing the trick. For any two natural numbers a and b , a^b is always greater than b^a if $a < b$. For eg :- $4^5 > 5^4$ $8^{11} > 11^8$ and so on. so in order to achieve the maximum signal strength we need to sort the amplification factors of the amplifiers in descending order. That's it, job done. But what

if the sequence is 4 5 1 ? We sort it and the result is 5 4 1. So the result is $1^{4(5^1)} = 1$!! That's not desired as $4^5 > 1$. So we have to place all the ones at the beginning and then sort the numbers > 1 . But there is an exception. For only 2 and 3 and no numbers greater than 3, $2^3 < 3^2$. This is a case which needs to be handled separately.

Source Code (C++):

```
#include <stdio.h>
#include <conio.h>
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

class ArrangingAmplifiers {
public:
    static void main() {
        int T;
        // get the number of test cases
        scanf("%d",&T);

        while (T-- > 0) {
            int noOfOnes = 0;
            int noOfAmplifiers;
            scanf("%d",&noOfAmplifiers);

            vector<int> Ni;

            // get the amplifiers
            for(int i = 0, pi; i < noOfAmplifiers; i++){
                scanf("%d",&pi);
                // calculate the number of ones
                if(pi == 1) {
                    noOfOnes++;
                } else {
                    // store only the non ones
                    Ni.push_back(pi);
                }
            }

            // Display the amplifiers
            // 1. display the 1's
            for (int i = 1; i <= noOfOnes; i++) {
                printf("%d ", 1);
            }
            // 2. check if the no. of amplifiers is 2 and they are 2 and 3
            if ((Ni.size() == 2) && (Ni.at(0) + Ni.at(1) == 5)) {
                printf("2 3");
            } else {
                // 3. sort the remaining array
                sort(Ni.begin(), Ni.end());

                // 4. display the remaining elements
                for (int i = Ni.size() - 1; i >= 0; i--) {
                    printf("%d ", Ni.at(i));
                }
            }
            printf("\n");
        }
    }
};

int main() {
    ArrangingAmplifiers::main();
    getch();
    return 0;
}
```

Source Code (Java):

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.Arrays;

public class ArrangingAmplifiers {

    public static void main(String[] args) throws Exception {

        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        // get the number of test cases
        int T = Integer.parseInt(br.readLine());

        while (T-- > 0) {
            int noOfOnes = 0;

            int noOfAmplifiers = Integer.parseInt(br.readLine());
            String amplifiers[] = br.readLine().split("\\s");

            // calculate the number of ones
            // store only the non ones
            int Ni[] = new int[noOfAmplifiers];

            for (int i = 0, len = amplifiers.length, niIndex = 0; i < len; i++) {
                if (amplifiers[i].equals("1")) {
                    noOfOnes++;
                } else {
                    Ni[niIndex++] = Integer.parseInt(amplifiers[i]);
                }
            }

            // Display the amplifiers
            // 1. display the 1's
            for (int i = 1; i <= noOfOnes; i++) {
                System.out.printf("%d ", 1);
            }

            // 2. check if the no. of amplifiers is 2 and they are 2 and 3
            if ((Ni.length == 2) && (Ni[0] + Ni[1] == 5)) {
                System.out.printf("2 3");
            } else {
                // 3. sort the remaining array
                Arrays.sort(Ni);

                // 4. display the remaining elements
                for (int i = noOfAmplifiers - noOfOnes - 1; i >= 0; i--) {
                    System.out.printf("%d ", Ni[i]);
                }
            }
            System.out.printf("\n");
        }
    }
}
```

4. Happy Coins

A line of coins are given, some belongs to lxh while others belongs to hhb. Now the two guys are about to play a game. In every round of game, each player can select any two consecutive coins and change them into one coin, if the two coins belongs to the same person, the new one will be gained to hhb, or else it will belongs to lxh. The two guys take turns playing this game, lxh always play first. The game runs round by round. You can easily make sense that we will get only one coin at the end of game. Now the question is, suppose the two players perform optimally in the game, which person does the final coin belongs to?

Input

The first line of the input contains one integer T , which indicate the number of test cases.

Following each test, the first line contains an integer N ($N \leq 10^5$), the number of coins. Following N lines, each line contains a name, lxh or hhb, the name of whom this coins belongs to.

Output

For each case, output a line contains the name of whom the final coin belongs to.

Sample Input

```
1
2
lxh
hhb
```

Sample Output

```
Lxh
```

Time Limit: 1s

Source Limit: 50000B

Solution:

It's about finding the pattern in the placement of the coins

If we take

lxh --- 0

hhb --- 1

Then

when $n = 2$,

input1	input2	Ans
0	0	1(hhb)
0	1	0(lxh)
1	0	0(lxh)
1	1	1(hhb)

when $n = 3$,

input1	input2	input3	Ans
0	0	0	0(lxh)
0	0	1	1(hhb)
0	1	0	1(hhb)
0	1	1	0(lxh)
1	0	0	1(hhb)
1	0	1	0(lxh)
1	1	0	0(lxh)
1	1	1	1(hhb)

when n = 4,

input1	input2	input3	input4	Ans
0	0	0	0	1(hhb)
0	0	0	1	0(lxh)
0	0	1	0	0(lxh)
0	0	1	1	1(hhb)
0	1	0	0	0(lxh)
0	1	0	1	1(hhb)
0	1	1	0	0(lxh)
0	1	1	1	0(lxh)
1	0	0	0	0(lxh)
1	0	0	1	1(hhb)
1	0	1	0	1(hhb)
1	0	1	1	0(lxh)
1	1	0	0	1(hhb)
1	1	0	1	0(lxh)
1	1	1	0	0(lxh)
1	1	1	1	1(hhb)

So we can see that when n is even, if total number of hhb coins i.e., 0 is even, the result is hhb, else lxh. Again if n is odd, if number of hhb coins i.e., 0 is odd then result is hhb, else lxh.

Source Code (C++):

```
#include <stdio.h>
#include <conio.h>
#include <iostream>
using namespace std;

class HappyCoins {
public:
    static void main() {
        int T;
        // get the number of test cases
        scanf("%d",&T);

        while (T-- > 0) {

            int noOfCoins;
            scanf("%d",&noOfCoins);

            int hhbCount = 0;

            for (int i = 1; i <= noOfCoins; i++) {
                char curInputCoin[5] = {'\0'};
                scanf("%s",&curInputCoin);
                if(strcmp(curInputCoin,"hhb") == 0){
                    hhbCount++;
                }
            }

            // when noOfCoins is even, if total number of hhb coins is even,
            // then the result is hhb, else lxh.
            // Again if noOfCoins is odd, if number of hhb coins is odd
            // then result is hhb, else lxh.

            if (noOfCoins % 2 == 0) {
                if (hhbCount % 2 == 0) {
```

```

        printf("hxb\n");
    } else {
        printf("lxb\n");
    }
} else {
    if (hxbCount % 2 != 0) {
        printf("hxb\n");
    } else {
        printf("lxb\n");
    }
}
}
}
};

int main() {
    HappyCoins::main();
    getch();
    return 0;
}

```

Source Code (Java):

```

import java.io.BufferedReader;
import java.io.InputStreamReader;

public class HappyCoins {

    public static void main(String[] args) throws Exception {

        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        // get the number of test cases
        int T = Integer.parseInt(br.readLine());

        while (T-- > 0) {

            int noOfCoins = Integer.parseInt(br.readLine());
            int hxbCount = 0;

            for (int i = 1; i <= noOfCoins; i++) {
                String curInputCoin = br.readLine();
                if (curInputCoin.compareTo("hxb") == 0) {
                    hxbCount++;
                }
            }

            // when noOfCoins is even, if total number of hxb coins is even,
            // then the result is hxb, else lxb.
            // Again if noOfCoins is odd, if number of hxb coins is odd
            // then result is hxb, else lxb.

            if (noOfCoins % 2 == 0) {
                if (hxbCount % 2 == 0) {
                    System.out.println("hxb");
                } else {
                    System.out.println("lxb");
                }
            } else {
                if (hxbCount % 2 != 0) {
                    System.out.println("hxb");
                } else {
                    System.out.println("lxb");
                }
            }
        }
    }
}

```

5. A Concrete Simulation

You are given a matrix M of type 1234×5678 . It is initially filled with integers $1 \dots 1234 \times 5678$ in row major order. Your task is to process a list of commands manipulating M . There are 4 types of commands:

"R x y" swap the x-th and y-th row of M ;

"C x y" swap the x-th and y-th column of M ;

"Q x y" write out $M(x,y)$;

"W z" write out x and y where $z=M(x,y)$.

Input

A list of valid commands. Input terminated by EOF.

Output

For each "Q x y" write out one line with the current value of $M(x,y)$, for each "W z" write out one line with the value of x and y (interpreted as above) separated by a space.

Input:

```
R 1 2
Q 1 1
Q 2 1
W 1
W 5679
C 1 2
Q 1 1
Q 2 1
W 1
W 5679
```

Output:

```
5679
1
2 1
1 1
5680
2
2 2
1 2
```

Time Limit: 7s

Source Limit: 7777B

Solution:

In this problem the matrix M is like

1	2	3	4	5	...	5678
5679	5680	5681	5682	5683	...	11356
11357	17034
...
$1232 \times 5678 + 1$	1233×5678
$1233 \times 5678 + 1$	1234×5678

So, they are arranged in row major order i.e, if 'i' is the row number starting from 0 to 1233 and 'j' is the column number starting from 0 to 5677 then each element of matrix $M[i][j]$ can be represented as $i * 5678 + j + 1$.

Now, for query R i.e., to swap xth row with yth row if we swap each element of the xth row of matrix M with the yth row by running a for loop upto the total column length it would lead to TLE considering there can be many query of R type and each taking 5678 loop iterations. Same case for C type query.

So to avoid this we can maintain a 1D row array[1234] having elements from 0 to 1233 and a column array[5678] having elements 0 to 5677.

**** So instead of following the conventional approach of having a 2D array having $1234 * 5678$ memory locations we are restricting the memory locations to only $1234 + 5678$ locations.

So for Q type query each element $M[i][j] = \text{row}[i-1] * 5678 + \text{column}[j-1] + 1$

For R type query we would now just swap $\text{row}[x-1]$ and $\text{row}[y-1]$.

For C type query we would now just swap $\text{column}[x-1]$ and $\text{column}[y-1]$.

W type query would be just the reverse of Q type.

Source Code (C++):

```
#include <stdio.h>
#include <iostream>

class ConcreteSimulation {
public:
    static void main() {
        int row[1234], col[5678];
        int x, y, z, temp;
        char query;

        for (int i = 0; i < 1234; i++) {
            row[i] = i;
        }

        for (int j = 0; j < 5678; j++) {
            col[j] = j;
        }

        while (scanf("%c", &query) != EOF) {
            switch (query) {
                case 'R': {
                    scanf("%d %d", &x, &y);

                    temp = row[x - 1];
                    row[x - 1] = row[y - 1];
                    row[y - 1] = temp;
                    break;
                }
                case 'C': {
                    scanf("%d %d", &x, &y);

                    temp = col[x - 1];
                    col[x - 1] = col[y - 1];
                    col[y - 1] = temp;
                    break;
                }
                case 'Q': {
                    scanf("%d %d", &x, &y);

                    printf("%d\n", row[x - 1] * 5678 + col[y - 1] + 1);
                    break;
                }
                case 'W': {
                    scanf("%d", &z);
                }
            }
        }
    }
};
```

```

        x = (z - 1) / 5678;
        int i, j;
        for (i = 0; i < 1234; i++) {
            if (row[i] == x)
                break;
        }
        y = (z - 1) % 5678;
        for (j = 0; j < 5678; j++) {
            if (col[j] == y)
                break;
        }
        printf("%d %d\n", i + 1, j + 1);
        break;
    }
}
};

int main() {
    ConcreteSimulation::main();
}

```

Source Code (Java):

```

import java.io.BufferedReader;
import java.io.InputStreamReader;

public class ConcreteSimulation {

    public static void main(String[] args) throws Exception {
        int row[] = new int[1234], col[] = new int[5678];
        int x, y, z, temp;
        String query;

        for (int i = 0; i < 1234; i++) {
            row[i] = i;
        }

        for (int j = 0; j < 5678; j++) {
            col[j] = j;
        }

        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        while ((query = br.readLine()) != null) {
            char operation = query.charAt(0);

            switch (operation) {
                case 'R': {
                    String[] params = query.substring(2).split("\\s");
                    x = Integer.parseInt(params[0]);
                    y = Integer.parseInt(params[1]);

                    temp = row[x - 1];
                    row[x - 1] = row[y - 1];
                    row[y - 1] = temp;
                    break;
                }
                case 'C': {
                    String[] params = query.substring(2).split("\\s");
                    x = Integer.parseInt(params[0]);
                    y = Integer.parseInt(params[1]);

                    temp = col[x - 1];
                    col[x - 1] = col[y - 1];
                    col[y - 1] = temp;
                    break;
                }
                case 'Q': {
                    String[] params = query.substring(2).split("\\s");
                    x = Integer.parseInt(params[0]);
                    y = Integer.parseInt(params[1]);

```

```

        System.out.printf("%d\n", row[x - 1] * 5678 + col[y - 1] + 1);
        break;
    }
    case 'W': {
        String[] params = query.substring(2).split("\\s");
        z = Integer.parseInt(params[0]);

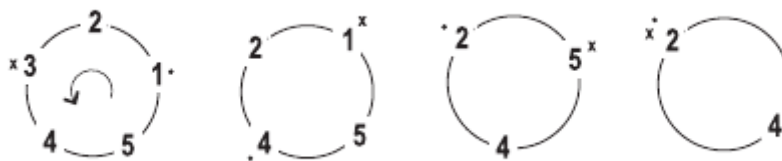
        x = (z - 1) / 5678;
        int i, j;
        for (i = 0; i < 1234; i++) {
            if (row[i] == x)
                break;
        }
        y = (z - 1) % 5678;
        for (j = 0; j < 5678; j++) {
            if (col[j] == y)
                break;
        }
        System.out.printf("%d %d\n", i + 1, j + 1);
        break;
    }
}
}
}
}
}

```

6. Musical Chairs

In the traditional game of Musical Chairs, $N + 1$ children run around N chairs (placed in a circle) as long as music is playing. The moment the music stops, children run and try to sit on an available chair. The child still standing leaves the game, a chair is removed, and the game continues with N children. The last child to sit is the winner.

In an attempt to create a similar game on these days' game consoles, you modify the game in the following manner: N Children are seated on N chairs arranged around a circle. The chairs are numbered from 1 to N . Your program pre-selects a positive number D . The program starts going in circles counting the children starting with the first chair. Once the count reaches D , that child leaves the game, removing his/her chair. The program starts counting again, beginning with the next chair in the circle. The last child remaining in the circle is the winner.



For example, consider the game illustrated in the figure above for $N = 5$ and $D = 3$. In the figure, the dot indicates where counting starts and x indicates the child leaving. So x denotes the D^{th} child that should leave. Starting off, child #3 leaves the game, and counting restarts with child #4. Child #1 is the second child to leave and counting restart with child #2 resulting in child #5 leaving. Child #2 is the last to leave, and child #4 is the winner. Write a program to determine the winning child given both N and D .

Input

Your program will be tested on one or more test cases. Each test case specifies two positive integers N and D on a single line, separated by one or more spaces, where $N, D < 1,000,000$.

The last line of the input file contains two 0's and is not part of the test cases.

Output

For each test case, write the winner using the following format:

$N \ D \ W$

Where N and D are as above, and W is the winner of that game.

Example

Input:

```
5 3
7 4
0 0
```

Output:

```
5 3 4
7 4 2
```

Time Limit: 15s

Source Limit: 50000B

Solution:

This apparently looks like a problem of circular linked list and can be solved easily by it by deleting the every d^{th} node among the n nodes. But the repeated breaking and joining of links in the worst case of $n=999999$ and $d=999999$ would not give the result in even 10mins. So we need to implement a faster way.

This can be done by a circular array approach. But even in that traversing the whole array repeatedly would lead to TLE. So to do it we can find a simple formula to implement the idea of circular array....

When a chair is removed its position is equal to the position of last removed chair + d

If we assume that $F(n, d)$ is position of the next to be removed, then

$$F(n, d) = F((n-1), d) + d$$

But this sequence goes around in a circle rather than linearly. So we need to wrap around from the end whenever we reach or cross the end of the array to the first. That is, if the array is from $\text{array}[0]$ to $\text{array}[n-1]$ where n is the number of chairs, then $\text{array}[n]$ would mean $\text{array}[0]$ as it is circular. We can do it by $a[n]=a[n\%n]=a[0]$. For $n+1$,

$n+2$, $n+3$ etc

$$a[(n+1) \% n] = a[1]$$

$$a[(n+2) \% n] = a[2]$$

$$a[(n+3) \% n] = a[3] \text{ and so on}$$

So our formula becomes

$F(n, d) = (F((n-1), d) + d) \% n$ with $F(1, d)=0$ when there is only one chair a is the winner(chair indexing starts from 0 to $n-1$)

So it boils down to a formula and no storage is required to implement it. Thus we have reduced the space complexity to $O(1)$ from $O(n)$ if had implemented circular linked list or circular array.

Source Code (C++):

```
#include <stdio.h>
#include <iostream>
using namespace std;

class MusicalChairs {
public:
    static void main() {
        while (true) {
            int N, D;
            scanf("%d %d", &N, &D);

            // if both the inputs are 0 then terminate
            if (N == 0 && D == 0)
                break;

            // calculate recursively last child
            int lastChild = 0;
            for (int i = 2; i <= N; i++) {
                lastChild = (lastChild + D) % i;
            }

            printf("%d %d %d\n", N, D, lastChild + 1);
        }
    }
};

int main() {
```

```

    MusicalChairs::main();
    return 0;
}

```

Source Code (Java):

```

import java.io.BufferedReader;
import java.io.InputStreamReader;

public class MusicalChairs {

    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        while (true) {

            String[] params = br.readLine().split("\\s");
            int N = Integer.parseInt(params[0]);
            int D = Integer.parseInt(params[1]);

            // if both the inputs are 0 then terminate
            if (N == 0 && D == 0)
                break;

            // calculate recursively last child
            int lastChild = 0;
            for (int i = 2; i <= N; i++) {
                lastChild = (lastChild + D) % i;
            }

            System.out.printf("%d %d %d\n", N, D, lastChild + 1);
        }
    }
}

```

7. Adjacent Bit Counts

For a string of n bits X , represented as $X_1, X_2, X_3, \dots, X_n$ the adjacent bit count of the string ($\text{AdjBC}(X)$) is given by

$$X_1 * X_2 + X_2 * X_3 + X_3 * X_4 + \dots + X_{n-1} * X_n$$

which counts the number of times a 1 bit is adjacent to another 1 bit i.e. how many times two 1 bits are side by side together. For example:

$\text{AdjBC}(011101101) = 3$

$\text{AdjBC}(111101101) = 4$

$\text{AdjBC}(010101010) = 0$

Note: a bit is represented by 0 & 1 only

Write a program which takes as input integers n and k and returns the number of bit strings X of n bits (out of 2^n) that satisfy $\text{AdjBC}(X) = k$. For example, for 5 bit strings, there are 6 ways of getting $\text{AdjBC}(X) = 2$:

11100, 01110, 00111, 10111, 11101, 11011

Input

The first line of input contains a single integer P , ($1 \leq P \leq 1000$), which is the number of data sets that follow. Each data set is a single line that contains the data set number, followed by a space, followed by a decimal integer giving the number (n) of bits in the bit strings, followed by a single space, followed by a decimal integer (k) giving the desired adjacent bit count. The number of bits (n) will not be greater than 100 and the parameters n and k will be chosen so that the result will fit in a signed 32-bit integer.

Output

For each data set there is one line of output. It contains the data set number followed by a single space, followed by the number of n -bit strings with adjacent bit count equal to k .

Example

Input:

```
10
1 5 2
2 20 8
3 30 17
4 40 24
5 50 37
6 60 52
7 70 59
8 80 73
9 90 84
10 100 90
```

Output:

```
1 6
2 63426
3 1861225
4 168212501
5 44874764
6 160916
7 22937308
8 99167
```

9 15476
10 23076518

Time Limit: 3s
Source Limit: 50000B

Solution:

Let $N(n, k)$ = the number of bit strings of length n and adjacent bit count k

We split that into

$N0(n, k)$ = number of bit strings ending in 0 with AdjBC = k

$N1(n, k)$ = number of bit strings ending in 1 with AdjBC = k

so $N(n, k) = N0(n, k) + N1(n, k)$

If the last bit is 0, we can get AdjBC = k if and only if

a) left $n-1$ bits has k adjacent 1 bits so

$$N0(n, k) = N(n-1, k) = N0(n-1, k) + N1(n-1, k)$$

If the last bit is 1, we can get AdjBC = k if

a) the bit before last bit is 0 and the first $n-1$ bits have AdjBC= k OR

b) the bit before last is 1 and the first $n-1$ bits have AdjBC= $k-1$ i.e

$$N1(n, k) = N0(n-1, k) + N1(n-1, k-1)$$

NOTE:-

a) $N(n, k) = N0(n, k) = N1(n, k) = 0$ if $k < 0$ or $k \geq n$, e.g :- $N(5, 7) = 0$ because we cannot find a string having AdjBC(5) = 7 when the total number of bits(i.e., 5) is less than 7(i.e., adjacent bit count).

b) $N1(1, 0) = 1$, $N0(1, 0) = 1$, $N(n, n-1) = 1$ (all 1 bits), e.g:- $N(5, 4) = 1$, i.e. a 5 bit string whose adjacent bit count is 4, which is possible if and only if all the bits of N are 1's.

Source Code (C++):

```
#include <iostream>
#include <conio.h>
using namespace std;

static const int MAX = 100;
static int N0[MAX + 1][MAX + 1];
static int N1[MAX + 1][MAX + 1];

class AdjacentBitCounts {
public:
    static void adjbc() {
        N0[1][0] = N1[1][0] = 1;
        for (int i = 2; i <= MAX; i++) {
            N0[i][0] = N0[i - 1][0] + N1[i - 1][0];
            N1[i][0] = N0[i - 1][0];
            for (int j = 1; j < i; j++) {
                N0[i][j] = N0[i - 1][j] + N1[i - 1][j];
                N1[i][j] = N0[i - 1][j] + N1[i - 1][j - 1];
            }
        }
    }
}
```



```
static void main() {
    adjbc();

    // get the number of test cases
    int T;
    cin >> T;

    int tcno, n, k;
    while (T-- > 0) {
        cin >> tcno >> n >> k;
        if (k < 0 || k >= n) {
            printf("%d 0\n", tcno);
        } else {
            printf("%d %d\n", tcno, N0[n][k] + N1[n][k]);
        }
    }
};

int main() {
    AdjacentBitCounts::main();
    getch();
    return 0;
}
```

Source Code (Java):

```
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class AdjacentBitCounts {
    private static final int MAX = 100;
    private static int N0[][] = new int[MAX + 1][MAX + 1];
    private static int N1[][] = new int[MAX + 1][MAX + 1];

    public static void main(String[] args) throws Exception {
        adjbc();

        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        // get the number of test cases
        int T = Integer.parseInt(br.readLine());

        while (T-- > 0) {
            String[] params = br.readLine().split("\\s");
            int tcno = Integer.parseInt(params[0]);
            int n = Integer.parseInt(params[1]);
            int k = Integer.parseInt(params[2]);

            if (k < 0 || k >= n) {
                System.out.printf("%d 0\n", tcno);
            } else {
                System.out.printf("%d %d\n", tcno, N0[n][k] + N1[n][k]);
            }
        }

        private static void adjbc() {
            N0[1][0] = N1[1][0] = 1;
            for (int i = 2; i <= MAX; i++) {
                N0[i][0] = N0[i - 1][0] + N1[i - 1][0];
                N1[i][0] = N0[i - 1][0];
                for (int j = 1; j < i; j++) {
                    N0[i][j] = N0[i - 1][j] + N1[i - 1][j];
                    N1[i][j] = N0[i - 1][j] + N1[i - 1][j - 1];
                }
            }
        }
    }
}
```

8. Robbery 2

k bandits robbed a bank. They took away n gold coins. Being a progressive group of robbers they decided to use the following procedure to divide the coins. First the most respected bandit takes 1 coin, then the second respected takes 2 coins, ..., the least respected takes k coins, then again the most respected takes k+1 coins, and so on, until one of the bandits takes the remaining coins. Calculate how much gold each of the bandits gets.

Input

The first line of the input contains number t – the amount of tests. Then t test descriptions follow. Each test consists of two integers, n and k - the amount of coins and bandits respectively.

Constraints

$1 \leq t \leq 500$

$106 \leq n \leq 1015$

$2 \leq k \leq 100$

Output

For each test print the amounts of coins each bandit gets separated by spaces.

Example

Input:

```
3
1000000 2
1234567 3
123456789 4
```

Output:

```
499849 500151
411602 411887 411078
30869901 30858368 30862296 30866224
```

Time Limit: 1s

Source Limit: 50000B

Solution:

This is an AP series problem. No specific algorithm is applied. Only brute force solution is applied. Let us take an example of n coins and k bandits. So the distribution of coins would be

1	2	3	...	k
k + 1	k + 2	k + 3	...	k + k
...	...			
$k*(r-1) + 1$	$k*(r-1) + 2$	$k*(r-1) + 3$...	$k*(r-1) + k$

where, r is the row number.

Let us assume that there are R rows which will be completely filled up, and some elements will be left over in the subsequent row.

Elements in the 1st row: 1 2 3 ... k

So the total number of elements in the 1st row = $\frac{k(k+1)}{2}$, this comes from the sum formula for AP series

Elements in the 1st and 2nd row: 1 2 ... k k + 1 k + 2 ... 2k

So the total number of elements in the 1st and 2nd rows = $\frac{2k(2k+1)}{2}$. So the total number of elements in all the R rows = $\frac{kR(kR+1)}{2}$

Since the total no. of coins is n, so $\frac{kR(kR+1)}{2} \leq n$. Solving this eqn. gives us the quadratic equation

$k^2R^2 + kR - 2n \leq 0$, to solve for R, we use the Shridhar Acharya's formula for roots of quadratic equation

$$R = \frac{-k \pm \sqrt{k^2 + 8kn}}{2k^2}, \text{ we ignore the -ve root because R must be +ve}$$

To get the coins received by each bandit we have to add column wise. Let us take the first column i.e. for the first bandit. So the coins received by him till now are 1 k + 1 ... k(R - 1) + 1. The total number of coins received by the first bandit till now = (first term + last term) * (no of terms) / 2, by the AP series formula. So it boils down to $\frac{(1 + (k(R-1) + 1))R}{2}$ i.e. $R + \frac{kR(kR-1)}{2}$. So for any column (j), replacing 1 with j in the first and last terms renders the equation as $\frac{(j + (k(R-1) + j))R}{2}$ which results in $jR + \frac{kR(kR-1)}{2}$

Now, there will be more 1 row which contains the left over. For all the leftover cases, the coins collected by individual bandits are to be calculated individually.

Source Code (C++):

```
#include <iostream>
#include <conio.h>
#include <math.h>
using namespace std;

class Robbery2 {
public:
    static void main() {
        // get the number of test cases
        int T;
        scanf("%d", &T);

        for (int i = 0; i < T; i++) {

            long long int n, k;
            scanf("%lld %lld", &n, &k);

            // array to store the coins received by bandits
            long long int *coinsBandits = new long long int [k];

            // Let us assume that there are R rows which will be completely
            // filled up, and some elements will the left over in the subsequent row.
            long long int R = (-k + sqrt((double)k * k * (1 + 8 * n)))/(2 * k * k);

            // So the total number of elements in all the R rows
            long long int elementsInRows = (k * R * (k * R + 1)) / 2;

            // the number of elements left over
            long long int leftOverElements = n - elementsInRows;

            // Since in the R rows all the bandits have received at-least something.
            // To get the coins received by each bandit we have to add column wise
            for (int j = 1; j <= k; j++) {
                coinsBandits[j - 1] = j * R + (k * R * (R - 1)) / 2;
            }

            // Now, there will be more 1 row which contains the left over.
            // For all the leftover cases, the coins collected by individual
```

```

// bandits are to collected individually
int j = 1;
while (leftOverElements > 0) {
    if (leftOverElements > (R * k + j)) {
        coinsBandits[j - 1] = coinsBandits[j - 1] + (R * k + j);
        leftOverElements = leftOverElements - (R * k + j);
        j++;
    } else {
        coinsBandits[j - 1] = coinsBandits[j - 1] + leftOverElements;
        leftOverElements = 0;
    }
}

// Display the coins received by each bandit
for (int c = 1; c <= k; c++) {
    printf("%lld ", coinsBandits[c - 1]);
}
printf("\n");
}
}
};

int main() {
    Robbery2::main();
    getch();
    return 0;
}

```

Source Code (Java):

```

import java.io.BufferedReader;
import java.io.InputStreamReader;

public class Robbery2 {

    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        // get the number of test cases
        int T = Integer.parseInt(br.readLine());

        for (int i = 0; i < T; i++) {
            String[] params = br.readLine().split("\\s");
            long n = Integer.parseInt(params[0]);
            int k = Integer.parseInt(params[1]);

            // array to store the coins received by bandits
            long coinsBandits[] = new long[k];

            // Let us assume that there are R rows which will be completely
            // filled up, and some elements will be left over in the subsequent row.
            long R = (-k + (long) (Math.sqrt(k * k * (1 + 8 * n)))) / (2 * k * k);

            // So the total number of elements in all the R rows
            long elementsInRows = (k * R * (k * R + 1)) / 2;

            // the number of elements left over
            long leftOverElements = n - elementsInRows;

            // Since in the R rows all the bandits have received at-least something.
            // To get the coins received by each bandit we have to add column wise
            for (int j = 1; j <= k; j++) {
                coinsBandits[j - 1] = j * R + (k * R * (R - 1)) / 2;
            }

            // Now, there will be more 1 row which contains the left over.
            // For all the leftover cases, the coins collected by individual
            // bandits are to collected individually

            int j = 1;
            while (leftOverElements > 0) {
                if (leftOverElements > (R * k + j)) {
                    coinsBandits[j - 1] = coinsBandits[j - 1] + (R * k + j);

```

```
        leftOverElements = leftOverElements - (R * k + j);
        j++;
    } else {
        coinsBandits[j - 1] = coinsBandits[j - 1] + leftOverElements;
        leftOverElements = 0;
    }
}

// Display the coins received by each bandit
for (int c = 1; c <= k; c++) {
    System.out.printf("%d ", coinsBandits[c - 1]);
}

System.out.println();
}
}
```

9. Publish or Perish

"Publish or perish" is the academic life's fundamental motto. It refers to the fact that publishing your work frequently is the only way to guarantee access to research funds, bright students and career advances. But publishing is not enough. It is necessary that your work is *referenced* (or *cited*). That is, your papers must be mentioned as source of information in other people's publications, to attest the quality and relevance of your research. The more citations a paper receives from other authors, the more it is considered influential.

In 2005 Jorge E. Hirsch, a physicist at the University of California at San Diego, proposed a way to evaluate the scientific impact of a researcher, based on the citations his or her papers have received. The *h-index*, as Hirsch's proposal became known, is a number based on the set of a researcher's most cited papers. It is defined in Hirsch's own words as: A scientist has index h if h of his N_p papers have at least h citations each, and the other $(N_p - h)$ papers have at most h citations each.

Albert Einstein, for example, published 319 papers in scientific journals and has an h -index equal to 46. It means 46 of his papers have received 46 or more citations each, and all of his remaining 273 papers have 46 citations or less each. Given the information of how many citations each paper from a given researcher has received, write a program to calculate that researcher's h -index.

Input

The input contains several test cases. The first line of a test case contains one integer N indicating the number of papers a researcher has published ($1 \leq N \leq 10^3$). The second line contains a list of N integers M_i , separated by one space, representing the number of citations each of the N papers from that author has received ($0 \leq M_i \leq 10^3$, for $1 \leq i \leq N$). The end of input is indicated by a line containing only one zero.

Output

For each test case in the input, your program must print a single line, containing one single integer, the h -index for the given list of citations.

Sample Input

```
4
1003 1 200 2
10
1 1 1 0 1 1 0 1 1 1
7
6 5 4 3 2 1 0
6
100 213 551 90 111 990
4
0 0 0 0
0
```

Sample Output

```
2
1
3
6
0
```

Time Limit: 1s

Source Limit: 50000B

Solution:

The solution can be achieved by following a very simple logic. First of all choose a certain number of papers which can be the potential h -index. Now, iterate over the citations for each paper to find that how many papers has citations which is greater than or equal the potential number of papers you have chosen i.e. the h -index. This takes care of the 1st clause of the problem statement - "A scientist has index h if h of his N_p papers have at least h citations each". Once this is taken care, do we really need to care of the second clause of the problem statement - "and the other $(N_p - h)$ papers have at most h citations each" ? I don't think so, this is automatically taken care of.

For example, lets assume the following scenario

4 papers and the papers have citations of 1002, 2, 1 and 200 respectively.

So, let's start with 4 papers, which can be potential h -index. How many papers have citations ≥ 4 ? Only two papers, paper 1 and paper 4 who have citations 1002 and 200 respectively. So, definitely 4 is not the desired h -index because we need at least 4 papers with citations ≥ 4 . In this way if we move down, selecting 4 then 3 then 2, we can see that if we select 2, we get three papers, paper 1, paper 3 and paper 4 whose citations are 1002, 2 and 200 and are at least equal to or greater than the number of papers chosen i.e. 2. Hence, it is the maximum h -index and our desired result.

Source Code (C++):

```
#include <iostream>
#include <conio.h>
using namespace std;

class PublishOrPerish {
public:
    static void main() {
        while (true) {
            // get the number of papers
            int n;
            scanf("%d", &n);

            int *citations = new int[n];

            if (n == 0)
                break;
            // get the citations per paper
            for (int i = 0; i < n; i++) {
                scanf("%d", &citations[i]);
            }

            int papersWithCitationsMoreThanProbablehIndex = 0;
            // start with the highest no of papers as the potential h-index
            for (int probablehindex = n; probablehindex >= 1; probablehindex--) {
                papersWithCitationsMoreThanProbablehIndex = 0;

                // iterate over the citations o find out how many papers
                // have citations >= potential h-index
                for (int j = 0; j < n; j++) {
                    if (citations[j] >= probablehindex)
                        papersWithCitationsMoreThanProbablehIndex++;
                }

                // if the no of citations is at least greater then or equal
                // to the potential h-index, then its the one, our desired result
                if (papersWithCitationsMoreThanProbablehIndex >= probablehindex) {
                    printf("%d\n", probablehindex);
                    break;
                }
            }
            // display also when there is no h-index
            if (papersWithCitationsMoreThanProbablehIndex == 0) {
                printf("%d\n", 0);
            }
        }
    }
}
```

```

    }
};

int main() {
    PublishOrPerish::main();
    getch();
    return 0;
}

```

Source Code (Java):

```

import java.io.BufferedReader;
import java.io.InputStreamReader;

public class PublishOrPerish {

    public static void main(String[] args) throws Exception {

        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        while (true) {
            // get the number of papers
            int n = Integer.parseInt(br.readLine());
            int citations[] = new int[n];

            if (n == 0)
                break;
            // get the citations per paper
            String[] params = br.readLine().split("\\s");
            for (int i = 0; i < params.length; i++) {
                citations[i] = Integer.parseInt(params[i]);
            }

            int papersWithCitationsMoreThanProbablehIndex = 0;
            // start with the highest no of papers as the potential h-index
            for (int probablehindex = n; probablehindex >= 1; probablehindex--) {
                papersWithCitationsMoreThanProbablehIndex = 0;

                // iterate over the citations o find out how many papers
                // have citations >= potential h-index
                for (int j = 0; j < n; j++) {
                    if (citations[j] >= probablehindex)
                        papersWithCitationsMoreThanProbablehIndex++;
                }

                // if the no of citations is at least greater then or equal
                // to the potential h-index, then its the one, our desired result
                if (papersWithCitationsMoreThanProbablehIndex >= probablehindex) {
                    System.out.println(probablehindex);
                    break;
                }
            }
            // display also when there is no h-index
            if (papersWithCitationsMoreThanProbablehIndex == 0) {
                System.out.println(0);
            }
        }
    }
}

```


10. Zig-Zag Rabbit

A $N \times N$ matrix is filled with numbers 1 to N^2 , diagonally in a zig-zag fashion.

The table below shows numbers in the matrix for $N = 6$.

1	2	6	7	15	16
3	5	8	14	17	26
4	9	13	18	25	27
10	12	19	24	28	33
11	20	23	29	32	34
21	22	30	31	35	36

There is a rabbit in the cell containing number 1. A rabbit can jump to a neighboring cell (up, down, left or right) if that cell exists.

Given K valid rabbit jumps, write a program that will calculate the sum of numbers of all cells that rabbit visited (add the number to the sum each time rabbit visits the same cell).

Input

The first line contains two integers N and K ($1 \leq N \leq 100\,000$, $1 \leq K \leq 300\,000$), the size of the matrix and the number of rabbit jumps.

The second line contains a sequence of K characters 'U', 'D', 'L' and 'R', describing the direction of each jump. The sequence of jumps will not leave the matrix at any moment.

Output

Output one integer, the sum of numbers on visited cells.

Note: This number doesn't always fit in 32-bit integer type.

Example

Input:

6 8
DDRRUULL

Output:

47

Input:

3 8
DDRRUULL

Output:

41

Input:

6 10
RRRRRDDDDD

Output:

203

Clarification for the first sample: The rabbit visits cells 1, 3, 4, 9, 13, 8, 6, 2 and 1. **Clarification for the second sample:** The rabbit visits cells 1, 3, 4, 8, 9, 7, 6, 2 and 1. **Clarification for the third sample:** The rabbit visits cells 1, 2, 6, 7, 15, 16, 26, 27, 33, 34 and 36.

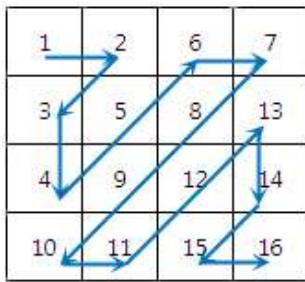
Time Limit: 1s

Source Limit: 50000B

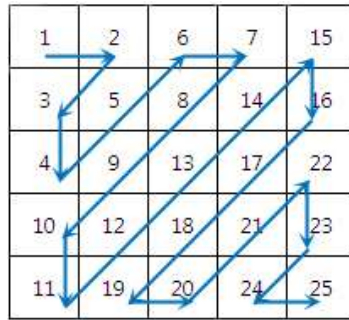
Solution:

We cannot simply form a 2D array of elements by calculating the value of elements of each row and column in a zig-zag fashion because the ' N ' can be in the order of $1 \leq N \leq 100\,000$, which means in the worst case, we will have a 2D Array of $10^5 \times 10^5$, which is not a feasible solution and will lead to huge memory requirements or TLE.

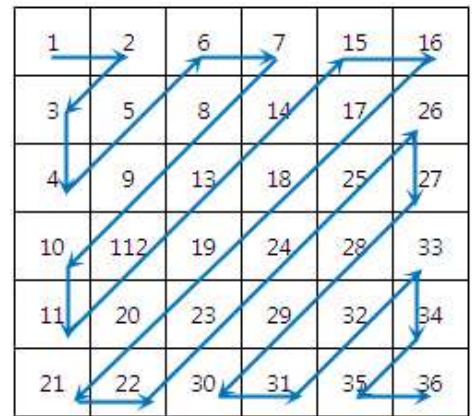
Let us consider the following examples with the zig-zag traversal paths shown:



A 4 x 4 zigzag maze



A 5 x 5 zigzag maze



A 6 x 6 zigzag maze

From observation, it is evident that the number of diagonals (or the upward and downward diagonal paths) are in an AP series and always odd numbered.

N	Diagonals	N	Diagonals
1	0	4	5
2	1	6	9
3	3	7	11

So, for $N \geq 2$, no. of diagonals is given by $2(N - 1) - 1 = 2N - 3$ (1)

Since the number of diagonals is always odd, so the middle diagonal D_{mid} is given by,

$$\frac{2N-3+1}{2} = N - 1 \quad \text{..... (2)}$$

And by observation, the no. of elements on D_{mid} is always = N (3)

So, for a mid diagonal, number of elements along it is $D_{mid} + 1$ i.e. N . Hence, for any diagonal $D \leq D_{mid}$, the number of elements along it is $D + 1$

For any diagonal $D \geq D_{mid}$, the number of elements along it is given by, $N_D = N_{D_{mid}} - (D - D_{mid})$

$$\text{i.e. } N_D = N - (D - (N - 1)) = 2N - D - 1$$

$$\text{Hence, } N_D = \begin{cases} D + 1, & \text{when } D < D_{mid} \\ N, & \text{when } D = D_{mid} \\ 2N - D - 1, & \text{when } D > D_{mid} \end{cases} \quad \text{..... (4)}$$

Also by observation, odd numbered diagonals are coming down, while even numbered diagonals are going upward. (5)

Let us take an example of a 5 x 5 matrix.

$$\text{Total number of diagonals} = 2N - 3 = 2 * 5 - 3 = 7 \quad \text{..... From (1)}$$

Also, consider that the rabbit is at cell (4, 4) which has the value of 21. We would devise a method to find out the value of any cell in the matrix given the coordinates of the cell.

So, the diagonal on which the rabbit is given by $(\text{row} + \text{col} - 2) = 6$ assuming the index to be starting from 1,1 and not 0,0.

So, the search coordinates can be anywhere on the 6th diagonal, start, end or between the ends. But, one thing is sure that at least 5 diagonals are already completed by the zigzag.

So, when $N = 5$, we have $D_{\text{mid}} = N - 1 = 4$ From (2)

We have, $N_D = \begin{cases} D + 1, \text{ when } D < 4 \\ N, \text{ when } D = 4 \\ 2 - D - 1, \text{ when } D > 4 \end{cases}$ From (4)

So, for diagonals 1, 2 and 3 we have no. of elements traversed by zigzag is $(1 + 1) + (2 + 1) + (3 + 1) = 9$

for diagonal 4 we have no. of elements traversed by zigzag is = 5

for diagonal 5 we have no. of elements traversed by zigzag is = $2 \times 5 - 5 - 1 = 4$

So, total no. of elements traversed by zigzag till now is = $9 + 5 + 4 = 18$

In order to be eligible for movement, the zigzag must traverse at least one cell i.e. (1, 1) the top left corner most box. So, adding that cell value (1) gives no. of elements traversed by zigzag till now is = $18 + 1 = 19$

So, we can see that we have reached the end of traversal till the last diagonal. Since, the zigzag is moving on the upward track, there must be at least one shift horizontally right, so that the zigzag reaches the 6th diagonal. So, no. of elements traversed by zigzag till now is = $19 + 1 = 20$. Since, the zigzag always travels in a diagonal (either up or down), and since (4, 4) with row = 4 is not the last row, is also on the upward directed diagonal and is also on the right side on the mid diagonal, so it must have traversed $\text{row}_{\text{max}} - \text{row}$ i.e. $5 - 1$ or 1 box up to reach its current position. So, adding that we get the no. of elements traversed by zigzag till now is = $20 + 1 = 21$. This is the desired result.

Similar logic can be extended for cells which are on the downward diagonals. Depending on the fact whether the cell is on the upward or downward diagonal, and whether it is on the right or left of the mid diagonal, 4 different possible combinations of finding the residual traversal can be formulated based on the above logic.

Source Code (C++):

```
#include <iostream>
using namespace std;

class ZigZagRabbit {
private:
    static long long int getCellValue(int n, int row, int col){
        // if (row, col) = (1, 1) or (n, n) then the rabbit on the top left or
        // bottom right corner locations.
        if(row == 1 && col == 1){
            return 1;
        }else if(row == n && col == n){
            return n * n;
        }

        // for n >= 2 the total number of diagonals i.e 2n - 3
        // since the number of diagonals is always odd, so the
        // mid diagonal is given by (2n - 3 + 1)/2 = n - 1;
        int midDiagonal = n - 1;

        // find the rabbit is on which diagonal, i.e. row + col - 2
        // assuming the index to be starting from 1,1 and not 0,0
        int rabbitIsOnDiagonal = row + col - 2;
```

```
// determine whether the rabbit is on a upward or downward diagonal
// odd numbered diagonal signifies downward and even numbered
// diagonals signify upward
bool rabbitMovingUpward = (rabbitIsOnDiagonal % 2) == 0;

// store the the value of the box. It is initialized to 1 because to be
// eligible for movement it has to traverse at least 1 box, (1, 1)
long long int totalCellValue = 1;

// if the rabbit is on a diagonal which is larger than the mid diagonal
// then we have to calculate in two steps, one for the diagonals which
// are less than or equal to the mid diagonal, and the other for diagonals
// which are greater than the mid diagonal
// NOTE: The for loops used below can further be optimized using formulas
// from AP Series.
if(rabbitIsOnDiagonal > midDiagonal){
    // When the diagonal number(d) is <= the midDiagonal, the number of elements
    // traversed by the zigzag is d + 1. Calculating for all such diagonals
    for(int diagonalNum = 1; diagonalNum <= midDiagonal; diagonalNum++){
        totalCellValue = totalCellValue + (diagonalNum + 1);
    }
    // When the diagonal number(d) is > the midDiagonal, the number of elements
    // traversed by the zigzag is 2n- d - 1. Calculating for all such diagonals
    // but spare the last diagonal, that have to be left for now and figured out
    // how much the rabbit has traversed along that diagonal
    for(int diagonalNum = midDiagonal + 1; diagonalNum < rabbitIsOnDiagonal; diagonalNum++){
        totalCellValue = totalCellValue + (2 * n - diagonalNum - 1);
    }
}else{
    // When the diagonal number(d) is <= the midDiagonal, the number of elements
    // traversed by the zigzag is d + 1. Calculating for all such diagonals
    for(int diagonalNum = 1; diagonalNum < rabbitIsOnDiagonal; diagonalNum++){
        totalCellValue = totalCellValue + (diagonalNum + 1);
    }
}
// since we have deliberately left the calculation for the last diagonal,
// in order for the zigzag to move to the last diagonal, it has to make at least
// 1 shift, either horizontally right or vertically down.
totalCellValue = totalCellValue + 1;
if(rabbitMovingUpward == true){
    // if the zigzag is moving upward, and if the rabbitIsOnDiagonal <= midDiagonal then
    // zigzag must have followed a downward shift of 1 box first and then moved upward
    if(rabbitIsOnDiagonal <= midDiagonal){
        totalCellValue = totalCellValue + (rabbitIsOnDiagonal + 1 - row);
    }else{
        // if the zigzag is moving upward, and if the rabbitIsOnDiagonal > midDiagonal then
        // zigzag must have followed a horizontal shift of 1 box first and then moved upward
        totalCellValue = totalCellValue + (n - row);
    }
}else{
    // if the zigzag is moving downward, and if the rabbitIsOnDiagonal <= midDiagonal then
    // zigzag must have followed a horizontal shift of 1 box first and then moved downward
    if(rabbitIsOnDiagonal <= midDiagonal){
        totalCellValue = totalCellValue + (rabbitIsOnDiagonal + 1 - col);
    }else{
        // if the zigzag is moving downward, and if the rabbitIsOnDiagonal > midDiagonal then
        // zigzag must have followed a vertical shift of 1 box first and then moved downward
        totalCellValue = totalCellValue + (n - col);
    }
}

return totalCellValue;
}

public:
static void main() {
    while(true){
        int N, K;
        scanf("%d %d",&N,&K);

        // read the moves
        char *moves = new char[K];
        scanf("%s",moves);
    }
}
```

```

int row = 1;
int col = 1;

// at least to be eligible for a move rabbit has to start from cell(1, 1),
// so the sumOfNumbersOfVisitedCells initialized to 1
long long int sumOfNumbersOfVisitedCells = 1;

// for each move manipulate the row cols appropriately
for(int i = 0, len = strlen(moves); i < len; i++){
    switch(moves[i]){
        case 'D':
            row = row + 1;
            break;

        case 'U':
            row = row - 1;
            break;

        case 'R':
            col = col + 1;
            break;

        case 'L':
            col = col - 1;
            break;

        default:
            break;
    }
    sumOfNumbersOfVisitedCells = sumOfNumbersOfVisitedCells + getCellValue(N, row, col);
}

printf("%d\n", sumOfNumbersOfVisitedCells);
}
}
};

int main() {
    ZigZagRabbit::main();
    return 0;
}

```

Source Code (Java):

```

import java.io.BufferedReader;
import java.io.InputStreamReader;

public class ZigZagRabbit {

    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        while(true){
            String[] params = br.readLine().split("\\s");
            int N = Integer.parseInt(params[0]);
            //int K = Integer.parseInt(params[1]);

            // read the moves
            char[] moves = br.readLine().toCharArray();

            int row = 1;
            int col = 1;

            // at least to be eligible for a move rabbit has to start from cell(1, 1),
            // so the sumOfNumbersOfVisitedCells initialized to 1
            long sumOfNumbersOfVisitedCells = 1;

            // for each move manipulate the row cols appropriately
            for(char move : moves){
                switch(move){
                    case 'D':
                        row = row + 1;
                        break;

```

```

        case 'U':
            row = row - 1;
            break;

        case 'R':
            col = col + 1;
            break;

        case 'L':
            col = col - 1;
            break;

        default:
    }
    sumOfNumbersOfVisitedCells = sumOfNumbersOfVisitedCells + getCellValue(N, row, col);
}

System.out.println(sumOfNumbersOfVisitedCells);
}

}

private static long getCellValue(int n, int row, int col){
    // if (row, col) = (1, 1) or (n, n) then the rabbit on the top left or
    // bottom right corner locations.
    if(row == 1 && col == 1){
        return 1;
    }else if(row == n && col == n){
        return n * n;
    }

    // for n >= 2 the total number of diagonals i.e 2n - 3
    // since the number of diagonals is always odd, so the
    // mid diagonal is given by (2n - 3 + 1)/2 = n - 1;
    int midDiagonal = n - 1;

    // find the rabbit is on which diagonal, i.e. row + col - 2
    // assuming the index to be starting from 1,1 and not 0,0
    int rabbitIsOnDiagonal = row + col - 2;

    // determine whether the rabbit is on a upward or downward diagonal
    // odd numbered diagonal signifies downward and even numbered
    // diagonals signify upward
    boolean rabbitMovingUpward = (rabbitIsOnDiagonal % 2) == 0;

    // store the the value of the box. It is initialized to 1 because to be
    // eligible for movement it has to traverse at least 1 box, (1, 1)
    long totalCellValue = 1;

    // if the rabbit is on a diagonal which is larger than the mid diagonal
    // then we have to calculate in two steps, one for the diagonals which
    // are less than or equal to the mid diagonal, and the other for diagonals
    // which are greater than the mid diagonal
    // NOTE: The for loops used below can further be optimized using formulas
    // from AP Series.
    if(rabbitIsOnDiagonal > midDiagonal){
        // When the diagonal number(d) is <= the midDiagonal, the number of elements
        // traversed by the zigzag is d + 1. Calculating for all such diagonals
        for(int diagonalNum = 1; diagonalNum <= midDiagonal; diagonalNum++){
            totalCellValue = totalCellValue + (diagonalNum + 1);
        }
        // When the diagonal number(d) is > the midDiagonal, the number of elements
        // traversed by the zigzag is 2n - d - 1. Calculating for all such diagonals
        // but spare the last diagonal, that have to be left for now and figured out
        // how much the rabbit has traversed along that diagonal
        for(int diagonalNum = midDiagonal + 1; diagonalNum < rabbitIsOnDiagonal; diagonalNum++){
            totalCellValue = totalCellValue + (2 * n - diagonalNum - 1);
        }
    }else{
        // When the diagonal number(d) is <= the midDiagonal, the number of elements
        // traversed by the zigzag is d + 1. Calculating for all such diagonals
        for(int diagonalNum = 1; diagonalNum < rabbitIsOnDiagonal; diagonalNum++){
            totalCellValue = totalCellValue + (diagonalNum + 1);
        }
    }
}

```

```

    }
    // since we have deliberately left the calculation for the last diagonal,
    // in order for the zigzag to move to the last diagonal, it has to make at least
    // 1 shift, either horizontally right or vertically down.
    totalCellValue = totalCellValue + 1;
    if(rabbitMovingUpward == true){
        // if the zigzag is moving upward, and if the rabbitIsOnDiagonal <= midDiagonal then
        // zigzag must have followed a downward shift of 1 box first and then moved upward
        if(rabbitIsOnDiagonal <= midDiagonal){
            totalCellValue = totalCellValue + (rabbitIsOnDiagonal + 1 - row);
        }else{
            // if the zigzag is moving upward, and if the rabbitIsOnDiagonal > midDiagonal then
            // zigzag must have followed a horizontal shift of 1 box first and then moved upward
            totalCellValue = totalCellValue + (n - row);
        }
    }
    }else{
        // if the zigzag is moving downward, and if the rabbitIsOnDiagonal <= midDiagonal then
        // zigzag must have followed a horizontal shift of 1 box first and then moved downward
        if(rabbitIsOnDiagonal <= midDiagonal){
            totalCellValue = totalCellValue + (rabbitIsOnDiagonal + 1 - col);
        }else{
            // if the zigzag is moving downward, and if the rabbitIsOnDiagonal > midDiagonal then
            // zigzag must have followed a vertical shift of 1 box first and then moved downward
            totalCellValue = totalCellValue + (n - col);
        }
    }
}

return totalCellValue;
}
}

```

11. Dinostratus Numbers

Recent archaeological discoveries of researchers from the University of Alberta in Canada showed that a strange sequence of numbers were found on the walls of the pyramids of Egypt, the ruins of Macchu Picchu and the stones of Stonehenge. Intrigued by the apparent coincidence researchers triggered the Department of Mathematics to decipher what were special about that sequence or numbers?

The discovery was startling. All numbers were generated by matrices of Dinostratus. Dinostratus was a famous Greek mathematician who lived from 390 to 320 BC and worked in major geometry problems like squaring the circle. Dinostratus studied matrices M of size 3×3 formed by nine distinct integers with the property that for every position (i, j) , $i = 1, \dots, 3$, $j = 1, \dots, 3$ of matrix, the element $m(i, j)$ is a multiple of its neighbors $m(i-1, j)$, $m(i-1, j-1)$ and $m(i, j-1)$ (if they exist). In his honor, we say that n is a Dinostratus number if exist a matrix M with the property above such that $m(3, 3) = n$.

See an example with $n = 36$.

$$\begin{bmatrix} 1 & 2 & 4 \\ 3 & 6 & 12 \\ 9 & 18 & 36 \end{bmatrix}$$

The relationship between the Dinostratus numbers, the pyramids of Egypt, Stonehenge and the stones of the ruins of Machu Picchu still remains a great mystery. But researchers in Alberta are willing to study these magic numbers. Your task is to make a program that receives an integer n and checks whether this is a Dinostratus number.

Input

The input consists of several instances. Each instance is given by a line containing an integer n ($1 \leq n \leq 1048576$). The input ends with end of file.

Output

For each instance, you must print an identifier Instance k , where k is the number of the current instance. On the next line print yes if n is a Dinostratus number otherwise print no.

Example

Input

36
37
38

Output

Instance 1
yes

Instance 2
no

Instance 3
no

Time Limit: 4s

Source Limit: 50000B

Solution:

The necessary condition (may not be sufficient) to be a DINONUM is that, it must have at least 9 distinct factors since the matrix is of 3×3 order. (i)

Multiple of a DINONUM must be a DINONUM (ii)

The following cases can be derived on the basis of prime factorization. It is known that

For any number $n = p_1^{a_1} \times p_2^{a_2} \times p_3^{a_3} \times \dots \times p_n^{a_n}$

Total number of primes = n

Total numbers of divisors = $(a_1+1) \times (a_2+1) \times (a_3+1) \times \dots \times (a_n+1)$ (iii)

Let us consider the following scenarios:

Case1: $n = a^x$ (i.e. nos. with 1 prime factor). For a^x to be a DINONUM, it should have at least 9 divisors, which means that $x + 1 \geq 9$. Hence, $x \geq 8$ (It is necessary condition but may not be sufficient, because we still need to represent a^8 as a 3 x 3 Dinostratus matrix)

a^8 can be shown as:

$$\begin{bmatrix} 1 & a & a^3 \\ a^2 & a^5 & a^6 \\ a^4 & a^7 & a^8 \end{bmatrix}$$

So, a^8 is a DINONUM e.g. 256.

Hence, $n = a^x$ (i.e. nos. with 1 prime factor) is a DINONUM if $x \geq 8$ (1)

Case2: $n = a^x \times b^y$. For $a^x \times b^y$ to be a DINONUM, it should have at least 9 divisors, which means that, either

$x \geq 2$ & $y \geq 2$ OR $x \geq 4$ & $y = 1$ (or vice versa)

(It is necessary condition but may not be sufficient, because we still need to represent $a^x \times b^y$ as a 3 x 3 Dinostratus matrix)

$a^2 \times b^2$ is a DINONUM e.g. 36

$$\begin{bmatrix} 1 & a & a^2 \\ b & ab & a^2b \\ b^2 & ab^2 & a^2b^2 \end{bmatrix}$$

No representation exist for $a^4 \times b$ e.g. 48

$a^5 \times b$ is DINONUM e.g. 96

$$\begin{bmatrix} 1 & a & a^2 \\ b & ab & a^2b \\ a^3b & a^4b & a^5b \end{bmatrix}$$

Hence, $n = a^x \times b^y$ is a DINONUM for either $x \geq 2$ & $y \geq 2$ OR $x \geq 5$ & $y = 1$ (or vice versa) (2)

Case3: $n = a^x \times b^y \times c^z$. For $a^x \times b^y \times c^z$ to be a DINONUM, it should have at least 9 divisors, which means that

$x \geq 2$ or $y \geq 2$ or $z \geq 2$

(It is necessary condition but may not be sufficient, because we still need to represent $a^x \times b^y \times c^z$ as a 3 x 3 Dinostratus matrix)

$a^2 \times b \times c$ is DINONUM e.g. 60

$$\begin{bmatrix} 1 & a & a^2 \\ b & ab & a^2b \\ bc & ab & a^2bc \end{bmatrix}$$

Hence, $n = a^x \times b^y \times c^z$ is a DINONUM for any $(x, y, z) \geq 2$ (3)

Case4: $n = a^x \times b^y \times c^z \times d^w$. For $a^x \times b^y \times c^z \times d^w$ to be a DINONUM, it should have at least 9 divisors. Here, even if all the indices are 1, the total no. of divisors = $(x + 1) * (y + 1) * (z + 1) * (w + 1) \geq 16$. (It is necessary condition but may not be sufficient, because we still need to represent $a^x \times b^y \times c^z \times d^w$ as a 3 x 3 Dinostratus matrix)

abcd is a DINONUM e.g. 210

$$\begin{bmatrix} 1 & a & ab \\ c & ac & abc \\ cd & acd & abcd \end{bmatrix}$$

Hence, $n = a^x \times b^y \times c^z \times d^w$ with 4 prime factors must be DINONUM (4)

If n is having more than 4 prime factors, it must be a multiple of the form $abcd$, so it is also DINONUM (Remember multiples of DINONUM is also a DINONUM).

So, therefore the problem is reduced to only prime factorizing n and finding its total number of prime and total number of divisors.

Source Code (C++):

```
#include <iostream>
#include <vector>
#include <conio.h>
using namespace std;

class DinostratusNumbers {
private:
    /*
     * Check whether the number is Dinostratus
     * For any number  $n = p_1^{a_1} \times p_2^{a_2} \times p_3^{a_3} \times \dots \times p_n^{a_n}$ 
     * Total number of primes =  $n$ 
     * Total numbers of divisors =  $(a_1 + 1) \times (a_2 + 1) \times (a_3 + 1) \times \dots \times (a_n + 1)$ 
     */
    static bool isDinoStratusNumber(int n) {
        int primecount = 0;
        int totalDivisors = 1;
        vector<int> factors;

        int radix = 0;
        // check how many times it is divisible by 2. We have to check for 2,
        // because it is the only even prime number
        while (n % 2 == 0) {
            n /= 2;
            radix++;
        }
        // if it was divisible at least once, then 2 is prime factor of n
        if (radix > 0) {
            primecount++;
            factors.push_back(radix);
            totalDivisors *= (radix + 1);
        }
        // all the prime factors for a given number are less than  $\sqrt{n}$ . If there is
        // any prime factor for a given  $n > \sqrt{n}$  then there is only 1 such number

        // Here we are taking only the odd numbers in the loop, because prime
        // numbers are always odd. And also we are not finding the prime
        // numbers at all to save time.
        // For example, if a number is divisible by 3 it takes care of all the
        // multiples of 3 like 9, 27 etc. Since, we are always reducing
        //  $n$  to  $n/k$  from previous step, So by the time when the turn for 9, 27 etc
```

```
// comes to check whether they can divide left over n, they will not be
// able to do so. So multiples of primes are also eliminated by this simple process.
// This is applicable for all prime numbers

for (int k = 3; k * k <= n; k += 2) {
    radix = 0;
    while (n % k == 0) {
        radix++;
        n /= k;
    }
    // if it was divisible at least once, then k is prime factor of n
    if (radix > 0) {
        primecount++;
        factors.push_back(radix);
        totalDivisors *= (radix + 1);
    }
}
// if leftover n > 1 then it signifies that there exists the 1 prime
// factor > SQRT(n),
// otherwise n would have been 1 by now
if (n > 1) {
    primecount++;
    factors.push_back(radix);
    totalDivisors *= (radix + 1);
}

// there must be atleast 9 prime factors
if (totalDivisors < 9) {
    return false;
} else {
    // if the number is of form a^x * b^y,
    // then reject if x = 4 && y = 1 (or vice versa)
    if (primecount == 2) {
        if ((factors[0] == 4 && factors[1] == 1) || (factors[0] == 1 && factors[1] == 4)) {
            return false;
        } else {
            return true;
        }
    } else {
        return true;
    }
}
}
}

public:
    static void main() {
        int instance = 0, n;

        // get the number
        while (scanf("%d", &n) != EOF) {
            instance++;
            printf("Instance %d\n%s\n\n", instance, isDinoStratusNumber(n) == true ? "yes" : "no");
        }
    }
};

int main() {
    DinostratusNumbers::main();
    getch();
    return 0;
}
```

Source Code (Java):

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.LinkedList;
import java.util.List;

public class DinostratusNumbers {

    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

```

int instance = 0;
String input = null;
while ((input = br.readLine()) != null) {
    // get the number
    int n = Integer.parseInt(input);
    instance++;

    System.out.printf("Instance %d\n%s\n", instance, isDinoStratusNumber(n) == true ? "yes" : "no");
}

/**
 * Check whether the number is Dinostratus
 *
 * @param n the number
 * @return true or false
 * For any number  $n = p_1^{a_1} \times p_2^{a_2} \times p_3^{a_3} \times \dots \times p_n^{a_n}$ 
 * Total number of primes = n
 * Total numbers of divisors =  $(a_1 + 1) \times (a_2 + 1) \times (a_3 + 1) \times \dots \times (a_n + 1)$ 
 */
private static boolean isDinoStratusNumber(int n) {
    int primecount = 0;
    int totalDivisors = 1;
    List<Integer> factors = new LinkedList<Integer>();

    int radix = 0;
    // check how many times it is divisible by 2. We have to check for 2,
    // because it is the only even prime number
    while (n % 2 == 0) {
        n /= 2;
        radix++;
    }
    // if it was divisible at least once, then 2 is prime factor of n
    if (radix > 0) {
        primecount++;
        factors.add(radix);
        totalDivisors *= (radix + 1);
    }
    // all the prime factors for a given number are less than  $\sqrt{n}$ . If there is
    // any prime factor for a given  $n > \sqrt{n}$  then there is only 1 such number

    // Here we are taking only the odd numbers in the loop, because prime
    // numbers are always odd. And also we are not finding the prime
    // numbers at all to save time.
    // For example, if a number is divisible by 3 it takes care of all the
    // multiples of 3 like 9, 27 etc. Since, we are always reducing
    // n to n/k from previous step, So by the time when the turn for 9, 27 etc
    // comes to check whether they can divide left over n, they will not be
    // able to do so. So multiples of primes are also eliminated by this simple process.
    // This is applicable for all prime numbers

    for (int k = 3; k * k <= n; k += 2) {
        radix = 0;
        while (n % k == 0) {
            radix++;
            n /= k;
        }
        // if it was divisible at least once, then k is prime factor of n
        if (radix > 0) {
            primecount++;
            factors.add(radix);
            totalDivisors *= (radix + 1);
        }
    }
    // if leftover n > 1 then it signifies that there exists the 1 prime
    // factor >  $\sqrt{n}$ ,
    // otherwise n would have been 1 by now
    if (n > 1) {
        primecount++;
        factors.add(radix);
        totalDivisors *= (radix + 1);
    }

    // there must be atleast 9 prime factors

```

```

    if(totalDivisors < 9) {
        return false;
    } else{
        // if the number is of form a^x * b^y,
        // then reject if x = 4 && y = 1 (or vice versa)
        if(primecount == 2){
            if((factors.get(0)== 4 && factors.get(1) == 1) || (factors.get(0) == 1 && factors.get(1) == 4)){
                return false;
            } else{
                return true;
            }
        } else{
            return true;
        }
    }
}
}
}
}
}

```

12. Another Game With Numbers

Little Chikoo likes to play with numbers. Often he plays the following game:
He chooses a number N and a set of positive integers.

He writes down all the numbers from 1 to N.

He chooses the first number (say x) from the set and cancels out all the multiples of x from 1 to N, including x.

He repeats step 3 for all the numbers from the set.

One day Little Chikoo was in a mood to play pranks. So his brother asked him to play the game with a certain challenge. He made the game a little harder and asked him to find out the number of integers which aren't cancelled after he completes step 4. If he does that then Little Chikoo gets to play on his brother's Nintendo for one full day. Now Little Chikoo is in a hurry and wants to finish the job as soon as possible. He has asked for your help.

Input

The first line of the input contains N and K. ($N \leq 10^9$, $K \leq 15$)

Then K numbers follow all in a single line. All numbers are ≤ 100 .

Output

The output file must contain the number of integers that aren't cancelled after he finishes step 4 of the game.

Example

Input:

```
10 3
2 4 5
```

Output:

```
4
(The numbers 1, 3, 7 and 9 weren't cancelled).
```

Note : All intermediate results will fit in a 64-bit integer

Time Limit: 1s

Source Limit: 50000B

Solution:

In combinatorial mathematics, the inclusion-exclusion principle (also known as the sieve principle) states that if A and B are two (finite) sets, then

$$|A \cup B| = |A| + |B| - |A \cap B|$$

The meaning of the statement is that the number of elements in the union of the two sets is the sum of the elements in each set, respectively, minus the number of elements that are in both. Similarly, for three sets A, B and C,

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |B \cap C| - |A \cap C| + |A \cap B \cap C|$$

For the general case of the principle, let A_1, \dots, A_n be finite sets. Then

$$\left| \bigcup_{i=1}^n A_i \right| = \sum_{i=1}^n |A_i| - \sum_{i,j:1 \leq i < j \leq n} |A_i \cap A_j| + \sum_{i,j,k:1 \leq i < j < k \leq n} |A_i \cap A_j \cap A_k| - \dots + (-1)^{n-1} |A_1 \cap \dots \cap A_n|$$

where $|A|$ denotes the cardinality of the set A. The name comes from the idea that the principle is based on over-generous inclusion, followed by compensating exclusion. When $n > 2$, the exclusion of the pair wise intersections are (possibly) too severe, and the correct formula is as shown with alternating signs.

In this problem we have to implement the above idea. For example

Given N and let say 3 numbers(ka, kb, kc) we first cancel out the multiples of ka, kb, kc each.

So, there are $\frac{N}{ka}, \frac{N}{kb}, \frac{N}{kc}$ number of multiples of ka, kb, kc. Then we cancel out the common multiples of (ka, kb), (kb, kc) and (ka, kc).

So, there are $\frac{N}{lcm(ka, kb)}, \frac{N}{lcm(kb, kc)}, \frac{N}{lcm(ka, kc)}$ multiples

And finally the common multiples of (ka, kb, kc) ie., $\frac{N}{lcm(ka, kb, kc)}$

So, the numbers left will be,

$$N - \left[\frac{N}{ka} + \frac{N}{kb} + \frac{N}{kc} - \frac{N}{lcm(ka, kb)} - \frac{N}{lcm(kb, kc)} - \frac{N}{lcm(ka, kc)} + \frac{N}{lcm(ka, kb, kc)} \right]$$

Acknowledgement: Hendrik Maryns provided the codes to generate the set of combinations for given n and r.

Source Code (Java):

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.Arrays;
import java.util.Iterator;
import java.util.StringTokenizer;

public class AnotherGameWithNumbers{

    public static long lcm(long a, long b) {
        return (a * b) / gcd(a, b);
    }

    public static long lcm(int a[]) {
        int size = a.length;
        if (size > 2) {
            return lcm(lcm(Arrays.copyOf(a, size - 1)), a[size - 1]);
        } else {
            return lcm(a[0], a[1]);
        }
    }

    public static long gcd(long a, long b) {
        if (b == 0)
            return a;
        else
            return gcd(b, a % b);
    }

    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String str = br.readLine();
        StringTokenizer st = new StringTokenizer(str, " ");

        int N = Integer.parseInt(st.nextToken());
        int K = Integer.parseInt(st.nextToken());
        int S[] = new int[K];

        str = br.readLine();
        st = new StringTokenizer(str, " ");
```

```

    for(int k = 0; k < K; k++){
        S[k] = Integer.parseInt(st.nextToken());
    }

    long x = 0;
    for (int s = 0, len = S.length; s < len; s++) {
        x = x + N / S[s];
    }

    for (int i = 2; i <= S.length; i++) {
        for (int[] combination : new Combinator(S, i)) {
            long lcm = lcm(combination);
            x = x + (i % 2 == 0 ? -1 : 1) * N / lcm;
        }
    }

    System.out.println(N - x);
}

abstract class CombinatoricOperator implements Iterator<int[]>, Iterable<int[]> {

    /**
     * The elements the operator works upon.
     */
    protected int[] elements;

    /**
     * An integer array backing up the original one to keep track of the
     * indices.
     */
    protected int[] indices;

    /**
     * Initialise a new operator, with given elements and size of the arrays to
     * be returned.
     *
     * @param elements
     *         The elements on which this combinatoric operator has to act.
     * @param r
     *         The size of the arrays to compute.
     * @pre r should not be smaller than 0. | 0 <= r
     * @post The total number of iterations is set to the correct number. |
     *         new.getTotal() == initialiseTotal();
     * @post The number of variations left is set to the total number. |
     *         new.getNumLeft() == new.getTotal()
     */
    protected CombinatoricOperator(int[] elements, int r) {
        indices = new int[r];
        this.elements = Arrays.copyOf(elements, elements.length);
        total = initialiseTotal(elements.length, r);
        reset();
    }

    /**
     * Initialise the array of indices. By default, it is initialised with
     * incrementing integers.
     */
    protected void initialiseIndices() {
        for (int i = 0; i < indices.length; i++) {
            indices[i] = i;
        }
    }

    /**
     * The variations still to go.
     */
    private long numLeft;

    /**
     * The total number of variations to be computed.
     */
    private long total;

```



```

/**
 * Compute the total number of elements to return.
 *
 * @param n
 *         The number of elements the operator works on.
 * @param r
 *         The size of the arrays to return.
 * @return The total number of elements is always bigger than 0. | result >=
 *         0
 */
protected abstract long initialiseTotal(int n, int r);

/**
 * Reset the iteration.
 *
 * @post The number of iterations to go is the same as the total number of
 *        iterations. | new.getNumLeft() == getTotal()
 */
public void reset() {
    initialiseIndices();
    numLeft = total;
}

/**
 * Return number of variations not yet generated.
 */
public long getNumLeft() {
    return numLeft;
}

/**
 * Return the total number of variations.
 *
 * @return The factorial of the number of elements divided by the factorials
 *         of the size of the variations and the number of elements minus
 *         the size of the variations. That is, with the number of elements
 *         = n and the size of the variations = r:  $n^r$ 
 */
public long getTotal() {
    return total;
}

/**
 * Returns <tt>true</tt> if the iteration has more elements. This is the
 * case if not all n! permutations have been covered.
 *
 * @return The number of permutations to go is bigger than zero. | result ==
 *         getNumLeft().compareTo(BigInteger.ZERO) > 0;
 * @see java.util.Iterator#hasNext()
 */
public boolean hasNext() {
    return numLeft > 0;
}

/**
 * Compute the next combination.
 *
 * @see java.util.Iterator#next()
 */
public int[] next() {
    if (numLeft != total) {
        computeNext();
    }
    numLeft = numLeft - 1;
    return getResult(indices);
}

/**
 * Compute the next array of indices.
 */
protected abstract void computeNext();

```

```

    * Compute the result, based on the given array of indices.
    *
    * @param indexes
    *       An array of indices into the element array.
    * @return An array consisting of the elements at positions of the given
    *         array. | result[i] == elements[indexes[i]]
    */
    @SuppressWarnings("unchecked")
    private int[] getResult(int[] indexes) {
        int[] result = new int[indexes.length];
        for (int i = 0; i < result.length; i++) {
            result[i] = elements[indexes[i]];
        }
        return result;
    }

    /**
     * Not supported.
     *
     * @see java.util.Iterator#remove()
     */
    public void remove() {
        throw new UnsupportedOperationException();
    }

    /**
     * A combinatoric operator is itself an iterator.
     *
     * @return Itself. | result == this
     * @see java.lang.Iterable#iterator()
     */
    public Iterator<int[]> iterator() {
        return this;
    }

    /**
     * Compute the factorial of n.
     */
    public static long factorial(int n) {
        long fact = 1L;
        for (int i = n; i > 1; i--) {
            fact = fact * i;
        }
        return fact;
    }
}

/**
 * A class that sequentially returns all combinations of a certain number out of
 * an array of given elements. Thanks to Michael Gillegand for the base
 * implementation: {@link http://www.merriampark.com/comb.htm}
 *
 * @author <a href="mailto:hendrik.maryns@uni-tuebingen.de">Hendrik Maryns</a>
 * @param <T> The type of the elements of which combinations are to be
 *           returned.
 */
class Combinator extends CombinatoricOperator {
    /**
     * Initialise a new Combinator, with given elements and size of the arrays
     * to be returned.
     *
     * @param elements
     *       The elements of which combinations have to be computed.
     * @param r
     *       The size of the combinations to compute.
     * @pre   r should not be greater than the length of the elements, and
     *       not smaller than 0.
     *       | 0 <= r <= elements.length
     * @post  The total number of iterations is set to the factorial of the
     *         number of elements divided by the factorials of the size of the
     *         combinations and the number of elements minus the size of the
     *         combinations. That is, with the number of elements = n and the
     *         size of the combinations = r:
     */

```

```

*          n          n!
*          (    ) = -----
*          r          (n-r)!r!
*      | new.getTotal() == factorial(elements.length).divide(
*      | factorial(r).multiply(factorial(elements.length-r))
* @post The number of combinations left is set to the total number.
*      | new.getNumLeft() == new.getTotal()
*/
public Combinator(int[] elements, int r) {
    super(elements, r);
}

/**
 * Compute the total number of elements to return.
 *
 * @return The factorial of the number of elements divided by the
 *         factorials of the size of the combinations and the number of
 *         elements minus the size of the combinations.
 *         That is, with the number of elements = n and the size of the
 *         combinations = r:
 *         n          n!
 *         (    ) = -----
 *         r          (n-r)!r!
 * @see CombinatoricOperator#initialiseTotal(int, int)
 */
@Override
protected long initialiseTotal(int n, int r) {
    long nFact = factorial(n);
    long rFact = factorial(r);
    long nminusrFact = factorial(n - r);
    return nFact / (rFact * nminusrFact);
}

/**
 * Compute the next array of indices.
 *
 * @see CombinatoricOperator#computeNext()
 */
@Override
protected void computeNext() {
    int r = indices.length;
    int i = r - 1;
    int n = elements.length;
    while (indices[i] == n - r + i) {
        i--;
    }
    indices[i] = indices[i] + 1;
    for (int j = i + 1; j < r; j++) {
        indices[j] = indices[i] + j - i;
    }
}
}

```

13. Polynomial

The number of spectators at the FIFA World Cup increases year after year. As you sell the advertisement slots during the games for the coming years, you need to come up with the price a company has to pay in order to get an advertisement slot. For this, you need a good estimate for the number of spectators in the coming games, based on the number of spectators in the past games.

Your intuition tells you that maybe the number of spectators could be modeled precisely by a polynomial of degree at most 3. The task is to check if this intuition is true.

Input

The input starts with a positive integer N , the number of test cases. Each test case consists of one line. The line starts with an integer $1 \leq n \leq 500$, followed by n integers x_1, \dots, x_n with $0 \leq x_i \leq 50000000$ for all i , the number of spectators in past games.

Output

For each test case, print YES if there is a polynomial p (with real coefficients) of degree at most 3 such that $p(i) = x_i$ for all i . Otherwise, print NO.

Example

Input:

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

Output:

```
YES
YES
NO
```

Time Limit: 1s

Source Limit: 50000B

Solution:

This problem is one of the trickiest and hardest problems on SPOJ.

Let us assume the polynomial $p(i) = ai^3 + bi^2 + ci + d$

So this problem consists of finding the coefficients a, b, c, d when the p_i 's are given for different values of i . this can be solved using Gauss or Gauss Jordan Elimination method of solving polynomial equations. But that would lead to TLE. To do this, we take an offbeat mathematical approach.

$$p(1) = a + b + c + d$$

$$p(2) = 8a + 4b + 2c + d$$

$$p(3) = 27a + 9b + 3c + d$$

$$p(4) = 64a + 16b + 4c + d$$

Now we are given with p_i 's i.e., $p(1), p(2), p(3), p(4)$

So we can construct a matrix from the above 4 equations

$$\begin{bmatrix} p(1) \\ p(2) \\ p(3) \\ p(4) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 8 & 4 & 2 & 1 \\ 27 & 9 & 3 & 1 \\ 64 & 16 & 4 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$$

For convenience, Let $A_{4 \times 4} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 8 & 4 & 2 & 1 \\ 27 & 9 & 3 & 1 \\ 64 & 16 & 4 & 1 \end{bmatrix}$

Hence, $\begin{bmatrix} p(1) \\ p(2) \\ p(3) \\ p(4) \end{bmatrix} = A_{4 \times 4} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$

So, $\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = A^{-1} \begin{bmatrix} p(1) \\ p(2) \\ p(3) \\ p(4) \end{bmatrix}$

Now by multiplying the RHS we get coefficients. Replace the coefficients (a, b, c, d) in our original equation $p(i) = ai^3 + bi^2 + ci + d$ to get the polynomial.

Rest is putting the values of 'i' from 1 to n and compare with the given xi's. If for some i, $xi \neq p(i)$ then we print "NO" i.e., there is no such polynomial, else we print "YES".

But A^{-1} will be a matrix with fractional values. Calculating with them may lead to precision errors. So we multiply both sides of (1) by $\det(A)$. So,

$$\det(A) \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \det(A) \times A^{-1} \times \begin{bmatrix} p(1) \\ p(2) \\ p(3) \\ p(4) \end{bmatrix}$$

$$\begin{bmatrix} a' \\ b' \\ c' \\ d' \end{bmatrix} = [\text{cofactor}(A)]^T \times \begin{bmatrix} p(1) \\ p(2) \\ p(3) \\ p(4) \end{bmatrix}$$

This $[\text{cofactor}(A)]^T$ is a matrix which has integer values as

$$A^{-1} = \frac{1}{\det(A)} \times [\text{cofactor}(A)]^T$$

So now we have to check whether $\det(A) \times xi = p(i)$ for $i = 1$ to n or not and print "YES" or "No" accordingly.

We calculate $\det(A) = 12$

$$[cofactor(A)]^T = \begin{bmatrix} -2 & 6 & -6 & 2 \\ 18 & -48 & 42 & -12 \\ -52 & 114 & -84 & 22 \\ 48 & -72 & 48 & -12 \end{bmatrix}$$

$$A^{-1} = \begin{bmatrix} -0.1667 & 0.5000 & -0.5000 & 0.1667 \\ 1.5000 & -4.000 & 3.5000 & -1.0000 \\ -4.3333 & 9.5000 & -7.0000 & 1.8333 \\ 4.0000 & -6.0000 & 4.0000 & -1.0000 \end{bmatrix} \quad [\text{Done by MATLAB}]$$

Source Code (C++):

```
#include <iostream>
#include <conio.h>
using namespace std;

class Polynomial {
/**
 * p(1)= a + b + c + d
 * p(2)= 8a + 4b + 2c + d
 * p(3)= 27a + 9b + 3c + d
 * p(4)= 64a + 16b + 4c + d
 *
 *      1      1      1      1
 * A =  8      4      2      1
 *      27      9      3      1
 *      64      16      4      1
 *
 * det(A) = 12
 *
 *      -2      6      -6      2
 * cofactor(A) = 18      -48      42      -12
 *               -52      114      -84      22
 *               48      -72      48      -12
 */
public:
    static void main() {
        int cofactorATranspose[4][4] = {
            { -2, 6, -6, 2, },
            { 18, -48, 42, -12, },
            { -52, 114, -84, 22, },
            { 48, -72, 48, -12 }
        };

        int detA = 12;

        // get the number of test cases
        int T;
        for(scanf("%d",&T); T; --T){

            // get the number of games
            int n;
            scanf("%d",&n);

            // a column matrix to store all xi's. The size of xi has to be atleast 4
            // otherwise matrix multiplication will crash
            int **xi = new int* [n < 4 ? 4 : n];
            for(int i = 0; i < (n < 4 ? 4 : n); i++){
                xi[i] = new int[1];
            }
            // Initialize the array
            for(int i = 0; i < (n < 4 ? 4 : n); i++){
                xi[i][0] = 0;
            }

            // transfer all the inputs to xi matrix
```

```

for (int i = 0; i < n; i++) {
    scanf("%d",&xi[i][0]);
}

// Multiplying cofactor(A) and [xi[0] xi[1] xi[2] xi[3]]
// cfctrTmltXi = cofactor(A)Transpose multiplied with A
int **cfctrTmltXi = new int* [4];
for(int i = 0; i < 4; i++){
    cfctrTmltXi[i] = new int[1];
}
// Initialize the array
for(int i = 0; i < 4; i++){
    cfctrTmltXi[i][0] = 0;
}

for (int i = 0; i < 4; i++) {
    for (int k = 0; k < 4; k++){
        cfctrTmltXi[i][0] += cofactorATranspose[i][k] * xi[k][0];
    }
}
bool coeffsFound = true;
// replace the coefficients found into the polynomial and determine
// whether the polynomial satisfies xi. Since the coefficients are
// calculated from the cofactors, we have to check that whether the
// polynomial with new found coefficients satisfy det(A) * xi
for (int i = 1; i <= n; i++) {
    int pi = i * i * i * cfctrTmltXi[0][0] +
            i * i * cfctrTmltXi[1][0] +
            i * cfctrTmltXi[2][0] +
            cfctrTmltXi[3][0];
    int x = detA * xi[i - 1][0];
    // if any polynomial does not match, we have not found
    // our desired co-efficients, hence discard
    if (pi != x) {
        coeffsFound = false;
        break;
    }
}
if (coeffsFound == true) {
    printf("%s\n", "YES");
} else {
    printf("%s\n", "NO");
}
}
}
};

int main() {
    Polynomial::main();
    getch();
    return 0;
}

```

Source Code (Java):

```

import java.io.BufferedReader;
import java.io.InputStreamReader;

public class Polynomial {
    /**
     * p(1)= a + b + c + d
     * p(2)= 8a + 4b + 2c + d
     * p(3)= 27a + 9b + 3c + d
     * p(4)= 64a + 16b + 4c + d
     *
     *      1      1      1      1
     * A =  8      4      2      1
     *      27      9      3      1
     *      64      16      4      1
     *
     * det(A) = 12
     *
     *           -2      6      -6      2
     */
}

```

```

* cofactor(A)   = 18   -48   42   -12
*               -52   114  -84   22
*               48   -72   48   -12
*
*/

public static void main(String[] args) throws Exception {
    int cofactorATranspose[][] = {
        { -2, 6, -6, 2, },
        { 18, -48, 42, -12, },
        { -52, 114, -84, 22, },
        { 48, -72, 48, -12 }
    };

    int detA = 12;

    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

    // get the number of test cases
    int T = Integer.parseInt(br.readLine());

    while (T-- > 0) {

        String params[] = br.readLine().split("\\s");

        // get the number of games
        int n = Integer.parseInt(params[0]);

        // a column matrix to store all xi's. The size of xi has to be atleast 4
        // otherwise matrix multiplication will crash
        int xi[][] = new int[n < 4 ? 4 : n][1];

        // transfer all the inputs to xi matrix
        for (int i = 1; i <= n; i++) {
            xi[i - 1][0] = Integer.parseInt(params[i]);
        }

        // Multiplying cofactor(A) and [xi[0] xi[1] xi[2] xi[3]]
        // cfctrTmltXi = cofactor(A)Transpose multiplied with A
        int cfctrTmltXi[][] = new int[4][1];

        for (int i = 0; i < 4; i++) {
            for (int k = 0; k < 4; k++){
                cfctrTmltXi[i][0] += cofactorATranspose[i][k] * xi[k][0];
            }
        }

        boolean coeffsFound = true;
        // replace the coefficients found into the polynomial and determine
        // whether the polynomial satisfies xi. Since the coefficients are
        // calculated from the cofactors, we have to check that whether the
        // polynomial with new found coefficients satisfy det(A) x xi
        for (int i = 1; i <= n; i++) {
            int pi = i * i * i * cfctrTmltXi[0][0] +
                i * i * cfctrTmltXi[1][0] +
                i * cfctrTmltXi[2][0] +
                cfctrTmltXi[3][0];

            int x = detA * xi[i - 1][0];

            // if any polynomial does not match, we have not found
            // our desired co-efficients, hence discard
            if (pi != x) {
                coeffsFound = false;
                break;
            }
        }
        if (coeffsFound == true) {
            System.out.println("YES");
        } else {
            System.out.println("NO");
        }
    }
}

```


14. Instruction Decoder

Mathews uses a brand new 16-bit instruction processor. (Yeah I am being sarcastic!). It has one register (say R) and it supports two instructions:

ADD X; Impact: $R = (R + X) \bmod 65536$

MUL X; Impact: $R = (R * X) \bmod 65536$

[For both instructions $0 \leq X \leq 65535$]

Mathews sees a segment of code, but does not know what value the register had before the code was being executed. How many possible values can the register have after the segment completed execution?

Input Format:

The input file consists of multiple test cases.

The first line of each test case contains one integer, N. ($1 \leq N \leq 100,000$).

The following N lines contain one instruction each.

Input terminates with a line containing N=0, which must not be processed.

Output Format:

For each test case print one integer in a single line, denoting the number of different values the register can take after code execution.

Sample Input:

```
1
ADD 3
1
MUL 0
5
MUL 3
ADD 4
MUL 5
ADD 3
MUL 2
8
ADD 32
MUL 5312
ADD 7
MUL 7
ADD 32
MUL 5312
ADD 7
MUL 7
0
```

Sample Output:

```
65536
1
32768
16
```

Time Limit: 3s
Source Limit: 50000B

Solution:

This problem has a very tricky logic. Let's say the register R has an initial value 'a'. So, 'a' can be anything out of $0 \leq a \leq 65535$ because the register is of 16 bits. So, it can hold values from 0 to $2^{16} - 1$ i.e. 0 to 65535. So, there are 65536 possible values of R. When we do ADD X it is actually $(R+X) \% 65536$. So, if R+X crosses 65535 it wraps around and starts from 0. But it does not wrap around and overlap with the initial value of R i.e. 'a', because we cannot find any X such that $(a + X) \% 65536 = a$. The only X value that satisfies it is 65536 as $(a+65536) \% 65536 = a$ but we know that a register(X) can hold only up to 65535 as it is a 16 bit register. So, after ADD X number of different possible values the register can hold is 65536. So, there is no change in the different possible values the register can hold after the execution of the ADD instruction. So, we can ignore the instruction ADD X. Now we need to concentrate only on MUL X. For these we should multiply all the operands of MUL operations together and then store the entire product of MUL operands, $(\prod X_i)$ in a temporary variable. Now we maintain an array of size 65536 and multiply the entire operand product with values starting from 0 to 65535, divide each product by 65536 and store each of those values in the same array. We take 0 to 65535 because that is the number of distinct values possible for R. The distinct values that are stored in the array are the number of possible values of the execution of entire instruction execution.

Optimization

While calculating the temporary product of operands of MUL operations, we have devised a nice optimization technique here. e.g. lets us consider a system of 6 bits instead of 65536 for easy understanding. So the distinct possible values of R would be $0 - 2^6 - 1$ i.e. 0 - 63, total 64. If commands are MUL 9, MUL 8, MUL 87 then the value in the register(R) is multiplied by $9 \times 8 \times 87 = 6264$. So effective value in the register is $(R \times 6264) \% 64$, if $R = 2$, then $2 \times 6264 \% 64 = 48$. Now, instead of multiplying $2 \times 9 \times 8 \times 87$ we take 2×9 first which is 18. Since $18 < 64$ we do nothing. Now multiply next number i.e. 18×8 which is 144. Since $144 > 64$ so we take $144 \% 64 = 16$. Now multiply next number with the remainder, so 16×87 which is 1392. Since $1392 > 64$ so we take $1392 \% 64 = 48$. We arrive at the same result as above. We do this recursively because in worst case if the input had been MUL 65535 for n times (and as per the problem definition $1 \leq n \leq 100000$) we would need 65535^n , which is a HUGE HUGE number to store. Programatically it is not required to check the if clause whether product ≥ 64 , a mod operation irrespective of the fact whether the product is ≥ 64 or not would yield the same result. Actually, it brings us to a well known formula of mods i.e.

$$(ab) \% m = ((a \% m) \times (b \% m)) \% m, \text{ and } (a + b) \% m = ((a \% m) + (b \% m)) \% m$$

Source Code (C++):

```
#include <iostream>
#include <string.h>
#include <conio.h>
using namespace std;

class IDecoder {
public:
    static void main() {
        const int MAX = 65536;
        while (true) {
            // get the number of register instructions
            int n;
            scanf("%d", &n);

            if (n == 0) {
                break;
            } else {
                // store the register instructions
                char command[100005][5];
                int commandValue[100005];

                // number of MUL commands
                int noOfMul = 0;
```

```

// e.g. if bits = 6, so values are 0 - 2^6 -1 i.e. 0 - 63, total 64
// if commands are MUL 9, MUL 8, MUL 87 then the
// value in the register(R) is multiplied by 9 x 8 x 87 = 6264
// so effective value is (R x 6264) % 64, if R = 2, then 2 x 6264 % 64 = 48
// we devise a unique optimization here, instead of multiplying 2 x 9 x 8 x 87
// we take 2 x 9 first which is 18. Since 18 < 64 we do nothing. Now multiply
// next number i.e. 18 x 8 which is 144. Since 144 > 64 so we take 144 % 64 = 16
// now multiply next number, so 16 x 87 which is 1392. Since 1392 > 64
// so we take 1392 % 64 = 48
// We do this because in worst case if the input had been MUL 65535 for n times
// we would need 65535 ^ n, which is a HUGE HUGE number to store
int effectiveMultiplier = 1;
for (int i = 0; i < n; i++) {
    scanf("%s %d", &command[i], &commandValue[i]);

    // check if it's a multiply operation, after all we are
    // concerned with MUL only and ignoring ADD
    if (!strcmp(command[i], "MUL")) {
        noOfMul++;
        effectiveMultiplier = (effectiveMultiplier * commandValue[i]) % MAX;
    }
}
// if there is no MUL operation then they must be only ADD
if (noOfMul == 0) {
    printf("%d\n", MAX);
} else {
    // if one of the instruction is MUL 0, then entire
    // product is 0, hence only one distinct value is possible i.e. 1
    // irrespective of the initial value of R
    // if there is only one instruction MUL 1 or all instructions are MUL 1
    // then entire product is 1, in that case 65536(MAX) distinct values are possible
    // because R can take 65536(MAX) distinct values
    if (effectiveMultiplier == 0) {
        printf("%d\n", 1);
    } else if (effectiveMultiplier == 1) {
        printf("%d\n", MAX);
    } else {
        // define a array which stores the distinct elements
        int distinctRegisterValueAt[MAX];
        memset(distinctRegisterValueAt, 0, sizeof(distinctRegisterValueAt));
        // track the no of distinct elements
        int distinctElements = 0;

        for (int R = 0; R < MAX; R++) {
            long long int distinctRegisterValueAtIndex =
                ((long long int)effectiveMultiplier * R) % MAX;

            if (distinctRegisterValueAt[distinctRegisterValueAtIndex] == 0) {
                distinctRegisterValueAt[distinctRegisterValueAtIndex] = 1;
                distinctElements++;
            }
        }
        printf("%d\n", distinctElements);
    }
}
}
}
}
};

int main() {
    IDecoder::main();
    getch();
    return 0;
}

```

Source Code (Java):

```
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class InstructionDecoder {

    private static final int MAX = 65536;

    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        while (true) {
            // get the number of register instructions
            int n = Integer.parseInt(br.readLine());

            if (n == 0) {
                break;
            } else {
                // store the register instructions
                String[] command = new String[n];
                int[] commandValue = new int[n];

                // number of MUL commands
                int noOfMul = 0;

                // e.g. if bits = 6, so values are 0 - 2^6 -1 i.e. 0 - 63, total 64
                // if commands are MUL 9, MUL 8, MUL 87 then the
                // value in the register(R) is multiplied by 9 x 8 x 87 = 6264
                // so effective value is (R x 6264) % 64, if R = 2, then 2 x 6264 % 64 = 48
                // we devise a unique optimization here, instead of multiplying 2 x 9 x 8 x 87
                // we take 2 x 9 first which is 18. Since 18 < 64 we do nothing. Now multiply
                // next number i.e. 18 x 8 which is 144. Since 144 > 64 so we take 144 % 64 = 16
                // now multiply next number, so 16 x 87 which is 1392. Since 1392 > 64
                // so we take 1392 % 64 = 48
                // We do this because in worst case if the input had been MUL 65535 for n times
                // we would need 65535 ^ n, which is a HUGE HUGE number to store
                int effectiveMultiplier = 1;
                for (int i = 0; i < n; i++) {
                    String params[] = br.readLine().split("\\s");
                    command[i] = params[0];
                    commandValue[i] = Integer.parseInt(params[1]);

                    // check if it's a multiply operation, after all we are
                    // concerned with MUL only and ignoring ADD
                    if (command[i].compareTo("MUL") == 0) {
                        noOfMul++;
                        effectiveMultiplier = (effectiveMultiplier * commandValue[i]) % MAX;
                    }
                }

                // if there is no MUL operation then they must be only ADD
                if (noOfMul == 0) {
                    System.out.printf("%d\n", MAX);
                } else {
                    // if one of the instruction is MUL 0, then entire
                    // product is 0, hence only one distinct value is possible i.e. 1
                    // irrespective of the initial value of R
                    // if there is only one instruction MUL 1 or all instructions are MUL 1
                    // then entire product is 1, in that case 65536(MAX) distinct values are possible
                    // because R can take 65536(MAX) distinct values
                    if (effectiveMultiplier == 0) {
                        System.out.printf("%d\n", 1);
                    } else if (effectiveMultiplier == 1) {
                        System.out.printf("%d\n", MAX);
                    } else {
                        // define a array which stores the distinct elements
                        int distinctRegisterValueAt[] = new int[MAX];

                        // track the no of distinct elements
                        int distinctElements = 0;

                        for (int R = 0; R < MAX; R++) {
```

```
        long distinctRegisterValueAtIndex = ((long)effectiveMultiplier * R) % MAX;

        if (distinctRegisterValueAt[(int)distinctRegisterValueAtIndex] == 0) {
            distinctRegisterValueAt[(int)distinctRegisterValueAtIndex] = 1;
            distinctElements++;
        }
    }
    System.out.printf("%d\n", distinctElements);
}
}
}
}
}
```

15. Seinfeld

I'm out of stories. For years I've been writing stories, some rather silly, just to make simple problems look difficult and complex problems look easy. But alas! not for this one.

You're given a non empty string made in its entirety from opening and closing braces. Your task is to find the minimum number of "operations" needed to make the string stable. The definition for being stable is as follows:

1. An empty string is stable.
2. If S is stable, then {S} is also stable.
3. If S and T are both stable, then ST (the concatenation of the two) is also stable.

All of these strings are stable: {}, {}, and {}; But none of these: {}, {}, nor {}. The only operation allowed on the string is to replace an opening brace with a closing brace, or visa-versa.

Input

Your program will be tested on one or more data sets. Each data set is described on a single line. The line is a non-empty string of opening and closing braces and nothing else. No string has more than 2000 braces. All sequences are of even length.

The last line of the input is made of one or more '-' (minus signs.)

Output

For each test case, print the following line:

k. N

Where k is the test case number (starting at one) and N is the minimum number of operations needed to convert the given string into a balanced one.

Example

Input:

```
{
{}{}
{{}}
---
```

Output:

```
1. 2
2. 0
3. 1
```

Time Limit: 1s

Source Limit: 50000B

Solution:

To solve this we apply a greedy approach. We scan the entire bracket sequence from the left and keep a track of how many brackets are opened and how many are closed. To do this, we maintain a count variable, we increase count by 1 when we get a '{' and decrease it by 1 when we get a '}'. At any point of time, if

- count > 0, that means a bracket has been opened that has still not been closed, i.e., some opened brackets do not have a matching closing brackets so far.

- count = 0, that means the bracket sequence up to that point is balanced, i.e., all the brackets that have been opened so far are closed, i.e., all the opened brackets have matching closing brackets.
- count < 0, that means an extra bracket has been closed. So this extra closed bracket would not have any matching opened bracket to its right. So in order to balance it in future we must flip it and change it to an open status. On changing the bracket to an open one we also increase the count by 1. Now we maintain a variable (say flips) to keep a count of these kinds of flips whenever our count variable becomes < 0.

So, after scanning the whole bracket sequence we have a bracket sequence that has a number of opened brackets, but unmatched brackets, whose count is stored in our count variable. In order to balance this sequence, the minimum number of flips required is count/2. Since the input sequence is guaranteed to be even, count will always be even (as count = total input sequence – balanced sequence, so count = even – even, so count = even) and count/2 is a whole number. So the total number of flips is x + count/2.

Let us consider this example

```
1  2  3  4  5  6
{  }  }  } {  {
```

At 1, there is an open bracket, hence bracketOpenCloseCount = 1, flips = 0;

At 2, there is a closed bracket, hence bracketOpenCloseCount = 1 – 1 = 0, flips = 0;

At 3, there is a closed bracket, hence bracketOpenCloseCount = 0 – 1 = -1, flips = 0;

Since, bracketOpenCloseCount < 0, so flip it. Change bracketOpenCloseCount to 1, and add 1 to flips

At 3,

```
1  2  3  4  5  6
{  }  {  } {  {
```

bracketOpenCloseCount = 1, flips = 1;

At 4, there is a closed bracket, hence bracketOpenCloseCount = 1 – 1 = 0, flips = 1;

At 5, there is an open bracket, hence bracketOpenCloseCount = 0 + 1 = 1, flips = 1;

At 6, there is an open bracket, hence bracketOpenCloseCount = 1 + 1 = 2, flips = 1;

So, at the end of the entire scan, bracketOpenCloseCount = 2 i.e. 5 & 6 are open. So to make that part of the string stable we have to flip half of that number i.e. 2 / 2 = 1 flip. So, we had already 1 flip previously, adding this new flip brings the total number of flips to 1 + 1 = 2.

```
1  2  3  4  5  6
{  }  {  } {  }
```

Source Code (C++):

```
#include <iostream>
#include <string.h>
using namespace std;

class Seinfeld {
public:
    static void main() {
        int testCases = 0;
        while (true) {
```

```

int bracketOpenCloseCount = 0, flips = 0;

// get the input pattern { } { { { {
char brackets[2002];
scanf("%s",&brackets);

if (brackets[0] == '-') {
    break;
} else {

    testCases++;
    int len = strlen(brackets);
    for (int i = 0; i < len; i++) {
        // check if it is '{' otherwise it is '}'
        if (brackets[i] == '{')
            bracketOpenCloseCount++;
        else {
            // since it is '}' reduce bracketOpenCloseCount
            bracketOpenCloseCount--;
            // bracketOpenCloseCount < 0, it signifies more brackets
            // are closed than opened. So flip this bracket and change
            // it to open status. Change bracketOpenCloseCount to 1
            // and add 1 to flip count
            if (bracketOpenCloseCount < 0) {
                brackets[i] = '{';
                bracketOpenCloseCount = 1;
                flips++;
            }
        }
    }
    // calculate the number f flips
    flips = flips + bracketOpenCloseCount / 2;
    printf("%. %d\n", testCases, flips);
}
}
};

int main() {
    Seinfeld::main();
    return 0;
}

```

Source Code (Java):

```

import java.io.BufferedReader;
import java.io.InputStreamReader;

public class Seinfeld {

    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        int testCases = 0;
        while (true) {
            int bracketOpenCloseCount = 0, flips = 0;

            // get the input pattern { } { { { {
            char[] brackets = br.readLine().toCharArray();

            if (brackets[0] == '-') {
                break;
            } else {

                testCases++;
                int len = brackets.length;
                for (int i = 0; i < len; i++) {
                    // check if it is '{' otherwise it is '}'
                    if (brackets[i] == '{')
                        bracketOpenCloseCount++;
                    else {
                        // since it is '}' reduce bracketOpenCloseCount

```



```
        bracketOpenCloseCount--;  
        // bracketOpenCloseCount < 0, it signifies more brackets  
        // are closed than opened. So flip this bracket and change  
        // it to open status. Change bracketOpenCloseCount to 1  
        // and add 1 to flip count  
        if (bracketOpenCloseCount < 0) {  
            brackets[i] = '{';  
            bracketOpenCloseCount = 1;  
            flips++;  
        }  
    }  
}  
// calculate the number of flips  
flips = flips + bracketOpenCloseCount / 2;  
System.out.printf("%d. %d\n", testCases, flips);  
}  
}  
}
```

16. Picking Up Chicks

A flock of chickens are running east along a straight, narrow road. Each one is running with its own constant speed. Whenever a chick catches up to the one in front of it, it has to slow down and follow at the speed of the other chick. You are in a mobile crane behind the flock, chasing the chicks towards the barn at the end of the road. The arm of the crane allows you to pick up any chick momentarily, let the chick behind it pass underneath and place the picked up chick back down. This operation takes no time and can only be performed on a pair of chicks that are immediately next to each other, even if 3 or more chicks are in a row, one after the other.

Given the initial locations (X_i) at time 0 and natural speeds (V_i) of the chicks, as well as the location of the barn (B), what is the minimum number of swaps you need to perform with your crane in order to have at least K of the N chicks arrive at the barn no later than time T ?

You may think of the chicks as points moving along a line. Even if 3 or more chicks are at the same location, next to each other, picking up one of them will only let one of the other two pass through. Any swap is instantaneous, which means that you may perform multiple swaps at the same time, but each one will count as a separate swap.

Input

The first line of the input gives the number of test cases, C . C test cases follow. Each test case starts with 4 integers on a line -- N , K , B and T . The next line contains the N different integers X_i , in increasing order. The line after that contains the N integers V_i . All distances are in meters; all speeds are in meters per second; all times are in seconds.

Output

For each test case, output one line containing "Case #x: S ", where x is the case number (starting from 1) and S is the smallest number of required swaps, or the word "IMPOSSIBLE".

Limits

$1 \leq C \leq 100$;
 $1 \leq B \leq 1,000,000,000$;
 $1 \leq T \leq 1,000$;
 $0 \leq X_i < B$;
 $1 \leq V_i \leq 100$;
 $1 \leq N \leq 50$;
 $0 \leq K \leq N$;

All the X_i 's will be distinct and in increasing order.

Example

Input:

```
3
5 3 10 5
0 2 5 6 7
1 1 1 1 4
5 3 10 5
0 2 3 5 7
2 1 1 1 4
5 3 10 5
0 2 3 4 7
2 1 1 1 4
```

Output:

Case #1: 0
Case #2: 2
Case #3: IMPOSSIBLE

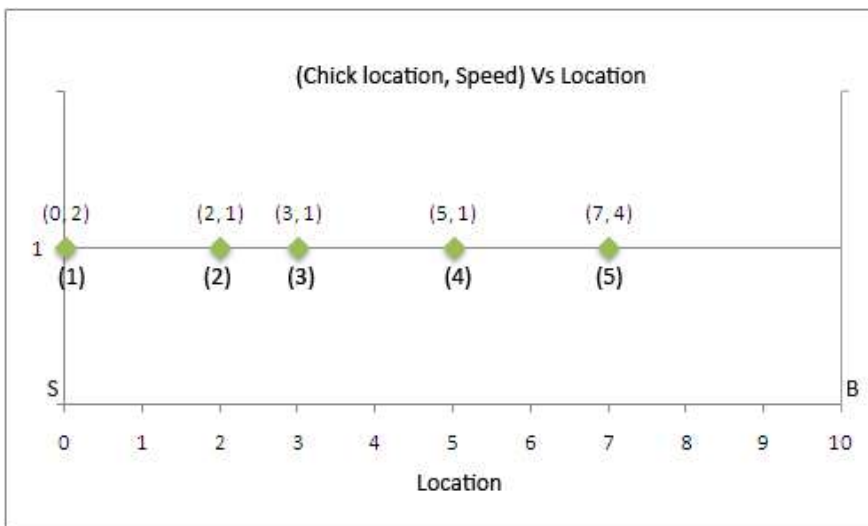
Time Limit: 1s
Source Limit: 50000B

Solution:

If any chick can reach the barn then the number of swaps it has to make is the number of chicks before it who cannot reach the barn. So the total number of swaps is the cumulative swaps of all chicks.

Let us take testcase 2 as the example.

N	K	B	T	
5	3	10	5	
X_1	X_2	X_3	X_4	X_5
0	2	3	5	7
V_1	V_2	V_3	V_4	V_5
2	1	1	1	4



The testcase is pictorially depicted on the left. Distance of Chick 5 from barn, B is $10 - 7 = 3$. In T time the chick will cover $T \times V_5 = 5 \times 4 = 20$. So Chick 5 can reach the barn. Similarly, Chick 4 will also be able to reach the barn. But chick 3 and 4 will not be able to reach the barn in T (5) time because they have a speed of 1 only, and their distance from the barn is > 5 . However, Chick 1 will be able to reach the barn because $T \times V_1 = 5 \times 2 = 10$ and distance between Chick 1 and barn is 10. In order for Chick 1 to reach the barn, it has to cross

chicks 3 and 4 who cannot reach the barn. So, total number of swaps is 2. If there had been other chicks beyond 3 and 4 who too can also reach the barn then the swaps count would have increased by further 2.

Source Code (C++):

```
#include <stdio.h>
#include <conio.h>
#include <iostream>
#include <vector>
using namespace std;

class PickingUpChicks {
public:
    static void main() {
        // get the number of testcase
        int C;
        scanf("%d", &C);

        for (int eachcase = 1; eachcase <= C; eachcase++) {
            // get the inputs

            int N, K, B, T;
            scanf("%d %d %d %d", &N, &K, &B, &T);
```

```

vector<int> X;
vector<int> V;

// get the chick locations
for (int x = 0, p; x < N; x++) {
    scanf("%d",&p);
    X.push_back(p);
}

// get the velocities of the chicks
for (int v = 0, p; v < N; v++) {
    scanf("%d",&p);
    V.push_back(p);
}

// store the number of chicks who can reach
int swaps = 0;
int cannotreach = 0;
int canreach = 0;
for (int i = N - 1; i >= 0; i--) {
    // distance to be covered by the chick
    int D = B - X[i];
    // time taken by the chick to reach the barn
    if (T * V[i] < D)
        cannotreach++;
    else {
        // if the chick can reach the barn then the number of swaps it has
        // to make is the number of chicks before it who cannot reach
        // total swaps is the cumulative swaps of all chicks
        swaps += cannotreach;
        canreach++;
        if (canreach == K)
            break;
    }
}
if (canreach == K)
    printf("Case #d: %d\n", eachcase, swaps);
else
    printf("Case #d: IMPOSSIBLE\n", eachcase);
}
};

int main() {
    PickingUpChicks::main();
    getch();
    return 0;
}

```

Source Code (Java):

```

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.StringTokenizer;

public class PickingUpChicks {

    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        // get the number of testcase
        int C = Integer.parseInt(br.readLine());

        for (int eachcase = 1; eachcase <= C; eachcase++) {
            String data = br.readLine();

            // get the inputs
            StringTokenizer st = new StringTokenizer(data, " ");
            int N = Integer.parseInt(st.nextToken());
            int K = Integer.parseInt(st.nextToken());
            int B = Integer.parseInt(st.nextToken());
            int T = Integer.parseInt(st.nextToken());

```

```

int X[] = new int[N];
int V[] = new int[N];

// get the chick locations
data = br.readLine();
st = new StringTokenizer(data, " ");
for (int x = 0; x < N; x++) {
    X[x] = Integer.parseInt(st.nextToken());
}

// get the velocities of the chicks
data = br.readLine();
st = new StringTokenizer(data, " ");
for (int v = 0; v < N; v++) {
    V[v] = Integer.parseInt(st.nextToken());
}

// store the number of chicks who can reach
int swaps = 0;
int cannotreach = 0;
int canreach = 0;

for (int i = N - 1; i >= 0; i--) {
    // distance to be covered by the chick
    int D = B - X[i];
    // time taken by the chick to reach the barn
    if (T * V[i] < D)
        cannotreach++;
    else {
        // if the chick can reach the barn then the number of swaps it has
        // to make is the number of chicks before it who cannot reach
        // total swaps is the cumulative swaps of all chicks
        swaps += cannotreach;
        canreach++;
        if (canreach == K)
            break;
    }
}

if (canreach == K)
    System.out.println("Case #" + eachcase + ": " + swaps);
else
    System.out.println("Case #" + eachcase + ": IMPOSSIBLE");
}
}
}

```

17. Feanor The Elf

Feanor is an elf, and of course, he really likes arrows and bows. Surprisingly enough, Feanor has a laptop, but he knows nothing about programming, so he requires your help.

Feanor lives in a tower of height H , and he loves throwing arrows from the top of it. He had a good amount of intensive training and he knows that he always throws his arrows with the same initial speed V . He wants you to make a program that given H and V returns the maximum distance that a Feanor's arrow can reach when it hits the ground, measured from the base of the tower. With this information, he will be able to place a nice circular fence to prevent disoriented little elves from being killed.

Newtonian laws apply in Feanor's world and the gravity has the same strength as in ours. These laws can be summarized as follows:

- The position of Feanor is assumed to be a point. The same occurs with the position of his arrow at each moment in time.
- The initial speed V of the arrow can be expressed as $V_x^2 + V_y^2 = V^2$, where V_x and V_y are the horizontal and vertical components of V , respectively. Speed V_x is always non-negative, while speed V_y is positive if the arrow is thrown up, and negative if the arrow is thrown down.
- The initial position of the arrow is the position of Feanor.
- The horizontal position of the arrow (relative to Feanor's position) at time t is $x(t) = V_x t$
- The vertical position of the arrow (relative to Feanor's position) at time t is $y(t) = V_y t - gt^2/2$, where $g = 9.8 \text{ m/s}^2$

Input

The input contains several test cases. Each test case is described in a single line that contains two integers V and H separated by a single space. The value V is the initial speed of Feanor's arrow measured in m/s ($0 \leq V \leq 1000$), while the value H is the tower's height in meters ($0 \leq H \leq 1000$). The last line of the input contains the number -1 twice separated by a single space and should not be processed as a test case.

Output

For each test case output a single line with the radius of Feanor's fence in meters, rounded up to 6 decimal digits (he wants to be sure that he doesn't kill those cute little elves).

Example

Input:

```
1 0
10 0
100 0
1000 0
-1 -1
```

Output:

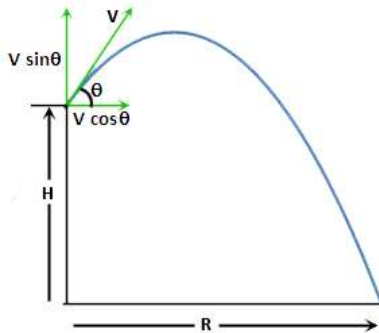
```
0.102041
10.204082
1020.408163
102040.816327
```

Time limit: 1s

Source limit: 50000B

Solution:

In physics, assuming a flat Earth with a uniform gravity field, an arrow launched with specific initial conditions will have a predictable range. As in Trajectory of a projectile, we will use:



g : the gravitational acceleration - usually taken to be 9.80 m/s² (32 f/s²) near the Earth's surface
 θ : the angle at which the arrow is launched
 V : the velocity at which the arrow is launched
 H : the initial height of the arrow
 R : the total horizontal distance travelled by the arrow

The horizontal position $x(t)$ of the arrow is

$$x(t) = V \cos \theta t \quad \dots (1)$$

In the vertical direction

$$y(t) = H + V \sin \theta t - \frac{1}{2}gt^2 \quad \dots (2)$$

We solve for (t) in the case where the (y) position of the arrow is at zero (when the arrow has reached the ground the equation becomes)

$$0 = H + V \sin \theta t - \frac{1}{2}gt^2$$

Again by applying the quadratic formula we find two solutions for the time. After several steps of algebraic manipulation

$$t = \frac{V \sin \theta}{g} \pm \frac{\sqrt{(V \sin \theta)^2 + 2gH}}{g} \quad \dots (3)$$

From (1) and (3), the horizontal distance covered by the arrow before hitting the ground

$$R = V \cos \theta t = \frac{V \cos \theta}{g} [V \sin \theta + \sqrt{(V \sin \theta)^2 + 2gH}] \quad \dots (4)$$

Maximum Range

For cases where the arrow lands at the same height from which it is launched, the maximum range is obtained by using a launch angle of 45°. An arrow that is launched with an elevation of 0 degrees will strike the ground immediately (range = 0). An arrow that is fired with an elevation of 90° (i.e. straight up) will travel straight up, then straight down, and strike the ground at the point from which it is launched, again yielding a range of 0.

The elevation angle which will provide the maximum range when launching the arrow from a non-zero initial height can be computed by finding the derivative of the range with respect to the elevation angle and setting the derivative to zero to find the maximum:

$$\frac{dR}{d\theta} = \frac{V^2}{g} \left[\cos \theta \left(\cos \theta + \frac{\sin \theta \cos \theta}{\sqrt{(\sin \theta)^2 + C}} \right) - \sin \theta \left(\sin \theta + \sqrt{(\sin \theta)^2 + C} \right) \right], \text{ where } C = \frac{2gH}{V^2}$$

Setting the derivative to zero provides the equation:

$$(\cos \theta)^2 + \frac{\sin \theta (\cos \theta)^2}{\sqrt{(\sin \theta)^2 + C}} - (\sin \theta)^2 - \sin \theta \sqrt{(\sin \theta)^2 + C} = 0$$

Substituting $u = (\cos \theta)^2$ and $1 - u = (\sin \theta)^2$ produces:

$$u + \frac{u\sqrt{1-u}}{\sqrt{1-u+C}} - (1-u) - (\sqrt{1-u})\sqrt{1-u+C} = 0$$

which reduces to the surprisingly simple expression:

$$u = \frac{C+1}{C-1}$$

Replacing our substitutions yields the angle that produces the maximum range for uneven ground, ignoring air resistance:

$$\theta = \cos^{-1} \sqrt{\frac{2gH+V^2}{2gH+2V^2}} \quad \dots (5)$$

Replacing $\cos \theta = \sqrt{\frac{2gH+V^2}{2gH+2V^2}}$ and $\sin \theta = \sqrt{1 - (\cos \theta)^2}$ in equation (4), we get

$$R = \frac{V}{g} \sqrt{\frac{2gH+V^2}{2gH+2V^2}} \left[V \sqrt{1 - \frac{2gH+V^2}{2gH+2V^2}} + \sqrt{V^2 \left(1 - \frac{2gH+V^2}{2gH+2V^2}\right) + 2gH} \right]$$

$$\text{or, } R = \frac{V}{g} \sqrt{\frac{2gH+V^2}{2gH+2V^2}} \left[\frac{V^2}{\sqrt{2gH+2V^2}} + \sqrt{\frac{V^4}{2gH+2V^2} + 2gH} \right] \quad \dots (6)$$

Source Code (C++):

```
#include <stdio.h>
#include <conio.h>
#include <iostream>
#include <cmath>
#define g 9.8
using namespace std;

class FeanorTheElf {
public:
    static void main() {
        double R, V, H;
        while(true){
            // get the velocity and height as input
            scanf("%lf %lf",&V,&H);
            if(V == -1 && H == -1) {
                break;
            }else if(V == 0 && H == 0) {
                printf("0.000000\n");
            } else{
                // the formula is explained in the document
                R = (V / g)
                    * sqrt((2 * g * H + V * V)
                        / (2 * g * H + 2 * V * V))
                    * ((V * V / sqrt(2 * g * H + 2 * V * V)) +
                        (sqrt((V * V * V * V / ((2 * g * H) + (2 * V * V)))
                            + 2 * g * H)));
                printf("%f\n", R);
            }
        }
    }
}
```



```

    }
}

int main() {
    FeanorTheElf::main();
    getch();
    return 0;
}

```

Source Code (Java):

```

import java.io.BufferedReader;
import java.io.InputStreamReader;

public class FeanorTheElf {

    public static void main(String[] args) throws Exception {
        final double g = 9.8;

        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        while (true) {
            // get the velocity and height as input
            String[] params = br.readLine().split("\\s");
            double V = Double.parseDouble(params[0]);
            double H = Double.parseDouble(params[1]);

            if (V == -1d && H == -1d)
                break;
            if (V == 0d && H == 0d)
                System.out.printf("0.000000\n");
            else {
                // the formula is explained in the document
                double R = (V / g)
                    * Math.sqrt((2 * g * H + V * V)
                        / (2 * g * H + 2 * V * V))
                    * ((V * V / Math.sqrt(2 * g * H + 2 * V * V)) +
                        (Math.sqrt((V * V * V * V / ((2 * g * H) + (2 * V * V))
                            + 2 * g * H)))));
                System.out.printf("%.6f\n", R);
            }
        }
    }
}

```

18. Use of function Arctan

It's easy to know that $\arctan(1/2) + \arctan(1/3) = \arctan(1)$. The problem is, to some fixed number A, you have to write a program to calculate the minimum sum B+C. A, B and C are all positive integers and satisfy the equation below:

$$\arctan(1/A) = \arctan(1/B) + \arctan(1/C)$$

Input

The first line contains a integer number T. T lines follow, each contains a single integer A, $1 \leq A \leq 60000$.

Output

T lines, each contain a single integer which denotes to the minimum sum B+C.

Example

Sample input:

```
4
1
50404
60000
17
```

Sample output:

```
5
2540664026
407570
73
```

Time limit: 10s

Source limit: 256B

Solution:

The above statement provides the equation

$$\tan^{-1}\left(\frac{1}{A}\right) = \tan^{-1}\left(\frac{1}{B}\right) + \tan^{-1}\left(\frac{1}{C}\right) \quad \text{.....(1)}$$

And also from trigonometric formulas we have

$$\tan^{-1}(p) + \tan^{-1}(q) = \tan^{-1}\left(\frac{p+q}{1-pq}\right) \quad \text{.....(2)}$$

Combining equations (1) and (2) we get,

$$\tan^{-1}\left(\frac{1}{A}\right) = \tan^{-1}\left(\frac{\frac{1}{B} + \frac{1}{C}}{1 - \frac{1}{BC}}\right) \text{ or, } \frac{1}{A} = \frac{\frac{1}{B} + \frac{1}{C}}{1 - \frac{1}{BC}} \text{ or, } A = \frac{BC-1}{B+C} \quad \text{.....(3)}$$

$$\text{From (3), expressing B in terms of A and C, we have, } B = \frac{AC+1}{C-A} \quad \text{.....(4)}$$

$$\text{Similarly, expressing C in terms of A and B, we have, } C = \frac{AB+1}{B-A}$$

$$\text{Hence, from (4) we get, } B + C = \frac{AC+1}{C-A} + C = \frac{AC+1+C^2-AC}{C-A} = \frac{C^2+1}{C-A} \quad \text{.....(5)}$$

Now, we are required to find the minimum value of $B + C$. In order to find the minimum value we have to take the derivative of the expression and equate it to 0 (zero).

Applying the formula for derivative given by, $\frac{d}{dx} \left(\frac{u}{v} \right) = \frac{v \frac{d}{dx}(u) - u \frac{d}{dx}(v)}{v^2}$ to equation (5) yields,

$$\frac{d}{dC} (B + C) = \frac{(C - A) \frac{d}{dC} (C^2 + 1) - (C^2 + 1) \frac{d}{dC} (C - A)}{(C - A)^2}$$

$$\text{or, } \frac{d}{dC} (B + C) = \frac{(C - A)2C - (C^2 + 1)}{(C - A)^2}$$

$$\text{or, } \frac{d}{dC} (B + C) = \frac{2C^2 - 2C - 1 - C^2}{(C - A)^2} = \frac{C^2 - 2AC - 1}{(C - A)^2}$$

In order for $B + C$ to be extremum, we must equate the derivative to 0, i.e.

$$\frac{d}{dC} (B + C) = \frac{C^2 - 2AC - 1}{(C - A)^2} = 0$$

$$\text{or, } C^2 - 2AC - 1 = 0$$

Solving the above equation by Shridhar Acharya's formula for roots of a quadratic equation yields,

$$C = \frac{2A \pm \sqrt{4A^2 + 4}}{2}, \text{ or } C = A \pm \sqrt{A^2 + 1} \quad \dots(6)$$

Following the same principle, we can say, we would get the extremum value of $B + C$, when $B = A \pm \sqrt{A^2 + 1}$

To find which of the above two values yields the minimum value of $B + C$, we must find for which value of C

$$\frac{d^2}{dC^2} (B + C) > 0, \text{ or } \frac{d}{dC} \left(\frac{d}{dC} (B + C) \right) > 0, \text{ or } \frac{d}{dC} \left(\frac{C^2 - 2AC - 1}{(C - A)^2} \right) > 0$$

$$\text{So, } \frac{d^2}{dC^2} (B + C) = \frac{(C - A)^2 \frac{d}{dC} (C^2 - 2AC - 1) - (C^2 - 2AC - 1) \frac{d}{dC} (C - A)^2}{(C - A)^4}$$

$$\text{or, } \frac{d^2}{dC^2} (B + C) = \frac{(C - A)^2 (2C - 2A) - (C^2 - 2AC - 1) 2(C - A)}{(C - A)^4}$$

$$\text{or, } \frac{d^2}{dC^2} (B + C) = \frac{2(A^2 + 1)}{(C - A)^3} = \frac{2(A^2 + 1)}{(C - A)(C - A)^2}$$

Putting, $C = A + \sqrt{A^2 + 1}$ in above equation, we get

$$\frac{d^2}{dC^2} (B + C) = \frac{2(A^2 + 1)}{(C - A)^3} = \frac{2(A^2 + 1)}{(C - A)(C - A)^2} = \frac{2(A^2 + 1)}{\sqrt{A^2 + 1}(A^2 + 1)} = \frac{2}{\sqrt{A^2 + 1}}, \text{ which is } > 0$$

Putting, $C = A - \sqrt{A^2 + 1}$ in above equation, we get

$$\frac{d^2}{dC^2} (B + C) = \frac{2(A^2 + 1)}{(C - A)^3} = \frac{2(A^2 + 1)}{(C - A)(C - A)^2} = \frac{2(A^2 + 1)}{-\sqrt{A^2 + 1}(A^2 + 1)} = -\frac{2}{\sqrt{A^2 + 1}}, \text{ which is } < 0$$

Hence, $C = A + \sqrt{A^2 + 1}$ for $B + C$ to be minimum(7)

So, it follows from above and by symmetry of B and C , that to get the minimum value of $B + C$, we should have $B = C = A + \sqrt{A^2 + 1}$, but by the problem statement it is not possible. Since both B and C are integers, and $A + \sqrt{A^2 + 1}$ will always be a fraction and can never be an integer, it follows that $B \neq C$. So, either one of B or C , should have to move left of $A + \sqrt{A^2 + 1}$ while the other has to shift right of $A + \sqrt{A^2 + 1}$. Let us chose B to be on the left side and C to be on the right side. It is as if, that two numbers B and C are trying their best to reach $A + \sqrt{A^2 + 1}$ from both right and left sides and be as close as possible to $A + \sqrt{A^2 + 1}$ to make $B + C$ minimum, yet at the same time they both should be integers. So, the closest integers possible to $A + \sqrt{A^2 + 1}$ are $2A$ and $2A + 1$ on the left and right side respectively.

More precisely, $B \leq \text{floor}(A + \sqrt{A^2 + 1}) \leq \lfloor A + \sqrt{A^2 + 1} \rfloor \leq 2A$, the largest integer not greater than $A + \sqrt{A^2 + 1}$. Also, $B > A$ otherwise, the value of $C = \frac{AB+1}{B-A}$ would be -ve.

and, $C \geq \text{ceiling}(A + \sqrt{A^2 + 1}) \geq \lceil A + \sqrt{A^2 + 1} \rceil \geq 2A + 1$, the smallest integer not less than $A + \sqrt{A^2 + 1}$

So, the ranges of B and C are given by, **$A < B \leq 2A$ and $2A + 1 \leq C$** (8)

So, the program logic would be, find a B where is $A < B \leq 2A$, for which value of $C = \frac{AB+1}{B-A}$, is an integer value. This is the optimum combination of B and C for $B + C$ to be minimum.

Source Code (C++):

```
#include <conio.h>
#include <stdio.h>

class Arctan {
public:
    static void main() {
        long long int T, A;
        scanf("%lld",&T);

        while(T--){
            scanf("%lld",&A);
            for(long long int B = 2 * A; B > A; B--){
                long long int numerator = A * B + 1;
                // Check the documentation, we have determined ranges for B and C
                // A < B <= 2A and C >= 2A + 1
                // Now, A is given by A = (BC - 1) / (B + C)
                // Also, C is given by C = (AB + 1) / (B - A)
                // Take values of B from 2A to A + 1 and find a integral value of C
                // Once C is found, B + C is the desired answer
                long long int denominator = B - A;
                // check for integral value of C
                if(numerator % denominator == 0){
                    long long int C = numerator/denominator;
                    printf("%lld\n", B + C);
                    break;
                }
            }
        }
    }
};

int main(){
    Arctan::main();
    getch();
    return 0;
}
```

Source Code (Java):

```
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class Arctan {

    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        // get the number of test cases
        int T = Integer.parseInt(br.readLine());

        while (T-- > 0) {
            int A = Integer.parseInt(br.readLine());
            // Check the documentation, we have determined ranges for B and C
            //  $A < B \leq 2A$  and  $C \geq 2A + 1$ 
            // Now, A is given by  $A = (BC - 1) / (B + C)$ 
            // Also, C is given by  $C = (AB + 1) / (B - A)$ 
            // Take values of B from 2A to A + 1 and find a integral value of C
            // Once C is found, B + C is the desired answer
            for(long B = 2 * A; B > A; B--){
                long numerator = A * B + 1;
                long denominator = B - A;
                // check for integral value of C
                if(numerator % denominator == 0){
                    long C = numerator / denominator;
                    System.out.printf("%d\n", B + C);
                    break;
                }
            }
        }
    }
}
```

19. Headshot

You have a revolver gun with a cylinder that has n chambers. Chambers are located in a circle on a cylinder. Each chamber can be empty or can contain a round. One chamber is aligned with the gun's barrel. When trigger of the gun is pulled, the gun's cylinder rotates, aligning the next chamber with the barrel, hammer strikes the round, making a shot by firing a bullet through the barrel. If the chamber is empty when the hammer strikes it, then there is no shot but just a "click".

You have found a use for this gun. You are playing Russian Roulette with your friend. Your friend loads rounds into some chambers, randomly rotates the cylinder, aligning a random chamber with a gun's barrel, puts the gun to his head and pulls the trigger. You hear "click" and nothing else - the chamber was empty and the gun did not shoot.

Now it is your turn to put the gun to your head and pull the trigger. You have a choice. You can either pull the trigger right away or you can randomly rotate the gun's cylinder and then pull the trigger. What should you choose to maximize the chances of your survival?

Input

The input first line contains a single line with a string of n digits "0" and "1" ($2 \leq n \leq 100$). This line of digits represents the pattern of rounds that were loaded into the gun's chambers. "0" represent an empty chamber, "1" represent a loaded one. In this representation, when cylinder rotates before a shot, the next chamber to the right gets aligned with the barrel for a shot. Since the chambers are actually located on a circle, the first chamber in this string follows the last one. There is at least one "0" in this string.

Output

Write to the output single one of the following words (without quotes):

"SHOOT" - if pulling the trigger right away makes you less likely to be actually shot in the head with the bullet (more likely that the chamber will be empty).

"ROTATE" - if randomly rotating the cylinder before pulling the trigger makes you less likely to be actually shot in the head with the bullet (more likely that the chamber will be empty).

"EQUAL" - if both of the above choices are equal in terms of probability of being shot.




Example

Input: 0011	Input: 0111	Input: 000111	Input: 0000
Output: EQUAL	Output: ROTATE	Output: SHOOT	Output: EQUAL

Time limit: 1s

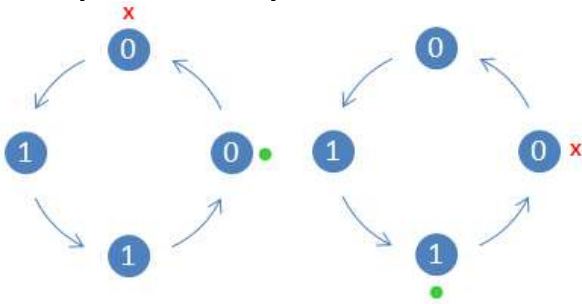
Source limit: 50000B

Solution:

At the first look the problem seems to involve probability theory but it can also be solved by simple observation of patterns and logic. Let us take the sample input test cases  that are provided as examples. The red  denotes the position of the round on the gun which was struck by my friend, and the green  represents the next position that will come directly under the hammer of the gun when the gun's cylinder rotates to bring in the next round.

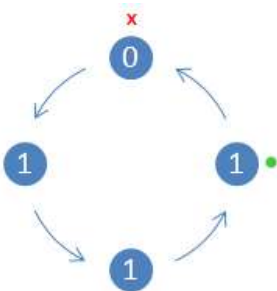
Case 1:

Since my friend has only heard a 'click' that means the hammer struck at chambers where there was no round. There are only two such possible scenarios as marked by red **x**. For the 1st scenario, after my friend strikes, the next chamber that comes under the hammer when the cylinder rotates does not have a round. So, I will be "saved". In the 2nd scenario, after my friend strikes, the next chamber that comes under the hammer when the cylinder rotates contains a round. So, I will "die". Since the possibility of being "saved" or "die" are both equal to 1, so, the outcome should be "EQUAL". There are equal chances of me being saved or killed whether I rotate or shoot.



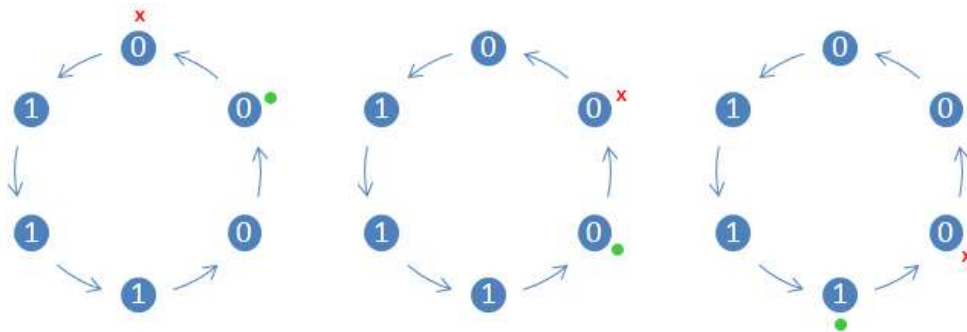
Case 2:

Since my friend has only heard a 'click' that means the hammer struck at chambers where there was no round. There is only one such possible scenario as marked by red **x**. For this scenario, after my friend strikes, the next chamber that comes under the hammer when the cylinder rotates contains a round. So, I will "die". Since, the possibility of being "saved" is 0 and the possibility to "die" is 1, so, in order to save myself the outcome should be to "ROTATE" in the hope that I may land up at an empty chamber.



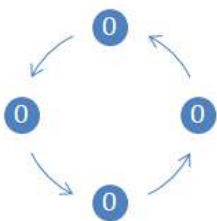
Case 3:

Since my friend has only heard a 'click' that means the hammer struck at chambers where there was no round. There are only three such possible scenarios as marked by red **x**. For the 1st scenario, after my friend strikes, the next chamber that comes under the hammer when the cylinder rotates does not have a round. So, I will be "saved". For the 2nd scenario also, after my friend strikes, the next chamber that comes under the hammer when the cylinder rotates does not have a round. So, I will be "saved". In the 3rd scenario, after my friend strikes, the next chamber that comes under the hammer when the cylinder rotates contains a round. So, I will "die". So, the possibility of being "saved" is 2 and the possibility to "die" is 1. So, in order to save myself the outcome should be "SHOOT".



Case 4:

This is a special case. Here, all the chambers are empty, so it does not matter whether I shoot or rotate. I will be "saved" in all the cases. Since, my possibility of being save is equal irrespective of the fact whether I rotate or shoot so the outcome should be "EQUAL".



So in general, we should find out the number of possibilities when one "dies" and also the number of possibilities when one is "saved". If the number of possibilities of dying is more than that of being saved,

the output must be "ROTATE" so that we might land up at a chamber that is empty. If the number of possibilities of being saved is more than that of being dying the output must be "SHOOT". If the number of possibilities of being saved is equal to that of being dying the output must be "EQUAL". Output should also be "EQUAL" when we will be saved always irrespective of the fact whether we choose to shoot or rotate.

To find the number of possibilities of being saved we have to count the sequence of "00". To find the number of possibilities of being dying we have to count the sequence of "01".

Source Code (C++):

```
#include <conio.h>
#include <stdio.h>
#include <iostream>
using namespace std;

class Headshot {
public:
    static void main() {
        char rounds[105];
        // zeroone counts the sequence of "01"
        // zerozero counts the sequence of "00"
        int len, zeroone = 0, zerozero = 0;
        scanf("%s", rounds);
        len = strlen(rounds);
        // the round is circular, so we just copy the first element
        // once again and append it to the end
        // just to save time we do not want to do rounds[i%len]
        rounds[len] = rounds[0];
        rounds[len + 1] = '\0';

        // check how many sequences of "00" and "01"
        // "00" signifies possibility to be saved
        // "01" signifies possibility of death
        for(int i = 0; i < len; i++){
            if(rounds[i] == '0'){
                if(rounds[i + 1] == '0') {
                    zerozero++;
                } else {
                    zeroone++;
                }
            }
        }
        // if possibility of being saved > possibility of death, "SHOOT"
        // if possibility of being saved < possibility of death, "ROTATE"
        // if possibility of being saved = possibility of death, "EQUAL"
        // if all 0's, then possibility of being saved is same irrespective
        // of the fact we rotate or shoot. Hence in that case we print "EQUAL"
        if(zerozero == len) {
            printf("EQUAL\n");
        } else if(zerozero == zeroone) {
            printf("EQUAL\n");
        } else if(zerozero > zeroone) {
            printf("SHOOT\n");
        } else {
            printf("ROTATE\n");
        }
    }
};

int main(){
    Headshot::main();
    getch();
    return 0;
}
```


Source Code (Java):

```
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class Headshot {

    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        StringBuilder rounds = new StringBuilder(br.readLine());
        int len = rounds.length();
        // the round is circular, so we just copy the first element
        // once again and append it to the end
        // just to save time we do not want to do rounds[i%len]
        rounds.append(rounds.charAt(0));

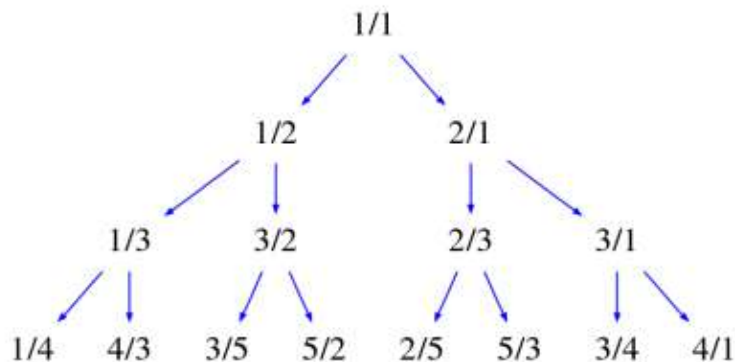
        // onezero counts the sequence of "01"
        // zerozero counts the sequence of "00"
        int zeroone = 0, zerozero = 0;
        // check how many sequences of "00" and "01"
        // "00" signifies possibility to be saved
        // "01" signifies possibility of death
        for(int i = 0; i < len; i++){
            if(rounds.charAt(i) == '0'){
                if(rounds.charAt(i + 1) == '0') {
                    zerozero++;
                }else{
                    zeroone++;
                }
            }
        }
        // if possibility of being saved > possibility of death, "SHOOT"
        // if possibility of being saved < possibility of death, "ROTATE"
        // if possibility of being saved = possibility of death, "EQUAL"
        // if all 0's, then possibility of being saved is same irrespective
        // of the fact we rotate or shoot. Hence in that case we print "EQUAL"
        if(zerozero == len) {
            System.out.println("EQUAL");
        } else if(zerozero == zeroone) {
            System.out.println("EQUAL");
        } else if(zerozero > zeroone) {
            System.out.println("SHOOT");
        } else {
            System.out.println("ROTATE");
        }
    }
}
```

20. Fractions on Tree

A fraction tree is an infinite binary tree defined as follows:

- 1) Every node of tree contains a fraction
- 2) Root of tree contains the fraction $1/1$
- 3) Any node with fraction i/j has two children : left child with fraction $i/(i+j)$ and right child with fraction $(i+j)/j$

For example, a fraction tree upto 3 level is as shown:



We number the nodes according to increasing levels (root is at level 1) and at any same level, nodes are numbered from left to right. So first node holds the fraction $1/1$, second one holds $1/2$, third one holds $2/1$ fourth one holds $1/3$ and so on.

Your task is simple, as always! Given two numbers a and b , you are to find the product of fractions at all those nodes whose number is between a and b both inclusive.

Input

Every line of the input contains two numbers a and b separated by a space. You are to find the product of all fractions which are at node having number between a and b both inclusive. Input file terminates with a $0\ 0$ which is not to be processed.

Output

For each input, print numerator and denominator of the lowest form of the fraction separated by a $/$. Output of each case to be done in separate lines.

Example

Input:

```

1 1
1 2
2 4
0 0
  
```

Output:

```

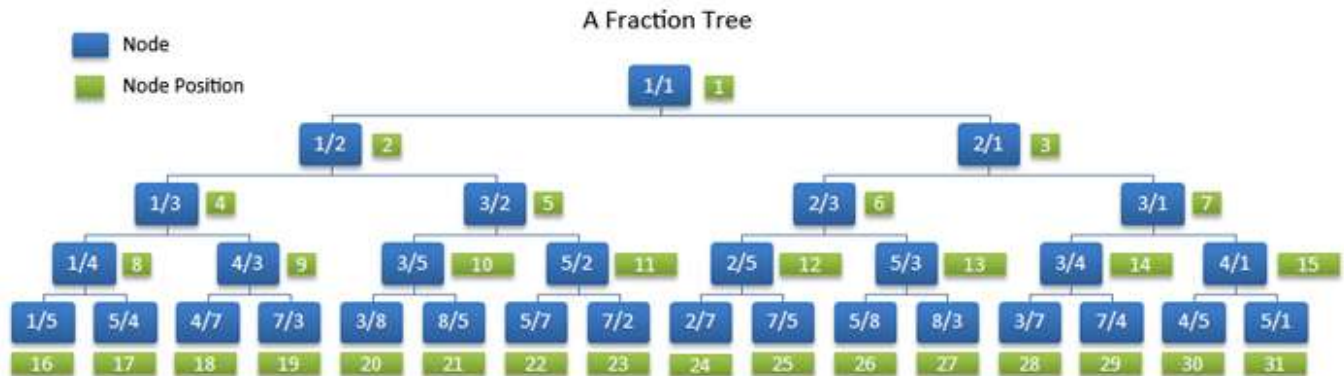
1/1
1/2
1/3
  
```

Time limit: 1.5s

Source limit: 50000B

Solution:

The figure below shows a sample fraction tree. Nodes are marked in blue with node positions beside each node marked in green.



For example let us take any two random positions on the tree, let's say position 9 and position 20. So, the product of all the nodes between 9 and 20 both inclusive is

$$\frac{4}{3} \times \frac{3}{5} \times \frac{5}{2} \times \frac{2}{5} \times \frac{5}{3} \times \frac{3}{4} \times \frac{4}{1} \times \frac{1}{5} \times \frac{5}{4} \times \frac{4}{7} \times \frac{7}{3} \times \frac{3}{8} \times \frac{8}{5} \times \frac{5}{7} \times \frac{7}{2} \times \frac{2}{7} \times \frac{7}{5} \times \frac{5}{8} \times \frac{8}{3} \times \frac{3}{7} \times \frac{7}{4} \times \frac{4}{5} \times \frac{5}{1} = \frac{4}{8} = \frac{1}{2}$$

Interesting point to observe is, when all the fractions between any two positions 'a' and 'b' are multiplied, the resulting fraction has the numerator equal to the numerator of the fraction at position 'a' and has the denominator equal to the denominator of the fraction at position 'b'. Here, in our case it is $\frac{4}{8}$. Since, the problem states to print numerator and denominator of the lowest form of the fraction, so we have to take the GCD of numerator and denominator of the resulting fraction, divide the numerator and denominator by GCD of the resulting fraction to reduce it to its lowest form. In the above example GCD of 4 and 8 is 4, so the fraction $\frac{4}{8}$ reduces to $\frac{1}{2}$.

So, the program logic is, find the numerator of the fraction at position 'a' and find the denominator of the fraction at position 'b'. The resulting fraction is this numerator / denominator in its lowest form. To, find the fraction at any position, we need to back track from that position to the root node and get the path track. Now we have to again traverse forward from root node (i.e. 1/1) through the path track to the given node and while traversing find the fractions at the intermediate nodes. In a binary tree, for any node, if it is even numbered then it is the left child of its root node and if it is odd numbered then it is the right child of its root node. And also from the problem description which states that any node with fraction i/j has two children: left child with fraction i/(i+j) and right child with fraction (i+j)/j. Combining the above two statements easily yields the intermediate nodes.

Source Code (C++):

```
#include <iostream>
#define fracNumerator(a) frac(a, true)
#define fracDenominator(a) frac(a, false)
using namespace std;

class FractionsOnTree {
public:
    // find the fraction, given the node position
    static long long int frac(long long int a, bool returnNum) {
        // since 1<= posn <= 10^10, and each level(x) contains 2^(1 - 1) elements
        // we have 2^x = 10^10, or x log2 = 10 log 10, which yields x = 33.22
        // hence max path levels is 33
        long long int path[34];
        long long int i = a;
        path[0] = i;

        // track the path count
        long long int j = 1;
```

```

// track back from the given node to the root node
while(i != 1) {
    i = i/2;
    path[j] = i;
    j++;
}

long long int num = 1, den = 1;
for(long long int i = j - 2; i >= 0; i--) {
    if(path[i] % 2 == 0) {
        den = num + den;
    } else {
        num = num + den;
    }
}

// return either numerator or denominator
if(returnNum) {
    return num;
} else {
    return den;
}
}

// find the gcd of two numbers
static long long int gcd(long long int a, long long int b) {
    if(b == 0) {
        return a;
    } else {
        return gcd(b, a % b);
    }
}

static void main() {
    long long int a, b;

    while(true) {
        scanf("%lld %lld", &a, &b);
        if(a == 0 && b == 0)
            break;

        long long int num_a = fracNumerator(a);
        long long int den_b = fracDenominator(b);

        // since the problem states to print numerator and denominator of
        // the lowest form of the fraction, we need to find the gcd of
        // the numerator and denominator
        long long int gcdab = gcd(num_a, den_b);
        printf("%lld/%lld\n", num_a/gcdab, den_b/gcdab);
    }
}

};

int main(){
    FractionsOnTree::main();
    return 0;
}

```

Source Code (Java):

```

import java.io.BufferedReader;
import java.io.InputStreamReader;

public class FractionsOnTree {

    // find the fraction, given the node position
    private static long fracNumerator(long a) {
        return frac(a, true);
    }

    private static long fracDenominator(long a) {
        return frac(a, false);
    }
}

```

```

}
private static long frac(long a, boolean returnNum) {
    // since 1<= posn <= 10^10, and each level(x) contains 2^(l - 1) elements
    // we have 2^x = 10^10, or x log2 = 10 log 10, which yields x = 33.22
    // hence max path levels is 33
    long path[] = new long[34];
    long n = a;
    path[0] = n;
    // track the path count
    int j = 1;

    // track back from the given node to the root node
    while(n != 1) {
        n = n/2;
        path[j] = n;
        j++;
    }

    long num = 1, den = 1;
    for(int i = j - 2; i >= 0; i--) {
        if(path[i] % 2 == 0) {
            den = num + den;
        } else {
            num = num + den;
        }
    }

    // return either numerator or denominator
    if(returnNum) {
        return num;
    } else {
        return den;
    }
}

// find the gcd of two numbers
static long gcd(long a, long b) {
    if(b == 0) {
        return a;
    } else {
        return gcd(b, a % b);
    }
}

public static void main(String[] args) throws Exception {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

    while(true) {
        String params[] = br.readLine().split("\\s");
        long a = Long.parseLong(params[0]);
        long b = Long.parseLong(params[1]);

        if(a == 0 && b == 0)
            break;

        long num_a = fracNumerator(a);
        long den_b = fracDenominator(b);

        // since the problem states to print numerator and denominator of
        // the lowest form of the fraction, we need to find the gcd of
        // the numerator and denominator
        long gcdab = gcd(num_a, den_b);
        System.out.printf("%d/%d\\n", num_a/gcdab, den_b/gcdab);
    }
}

```

21. Two Professors

There are two professors at the great Academy of X that really do not get along with each other. In order not to reveal their names, we will call them 1 and 2. The Academy employs exactly n professors; each of them has to give exactly one lecture. As their schedules are rather tight (they are professors, remember?), the starting and the ending time of each lecture is already fixed. However, it is not yet fixed where each lecture will take place. Obviously, it is impossible to schedule two lectures in the same room if their durations overlap; on the other hand, it is possible if one of them starts exactly at the same time that the other one ends. Your task is to find the minimal number of rooms allowing arranging all the lectures. But know that professors 1 and 2 hate each other so much that they will never give their lectures in the same room.

Input

The input contains several test cases. The first line contains the number of test cases t ($t \leq 250$). Each test begins with a line containing the number of professors n ($2 \leq n \leq 100000$). Next n lines follow, i -th of which contains two integers $start_i$ and end_i ($0 \leq start_i < end_i \leq 1000000000$), the starting and the ending time of the lecture that the i -th professor gives, respectively.

Output

For each test case output the minimal number of rooms necessary to schedule all the lectures.

Example

Input:

```
7
2
0 10
10 20
3
0 10
10 20
10 20
5
4 14
3 13
2 12
1 11
0 10
4
0 10
10 20
20 30
30 40
5
3 13
2 14
18 23
13 17
17 23
5
2 3
5 6
2 4
3 5
```

4 7
7
1 2
6 7
1 2
2 4
3 5
4 6
5 7

Output:

2
2
5
2
2
3
2

Time Limit: 17s
Source Limit: 50000B

Solution:

Source Code (C++):

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

struct ScheduleEvent {
    int time, prof;
    bool isStart;

    bool operator < (const ScheduleEvent& e) const {
        if (time != e.time) return time < e.time;
        if (isStart != e.isStart) return !isStart;
        return prof < e.prof;
    }
};

class TwoProfessors {
public:
    static void main(){
        // get the number of test cases
        int T;
        cin >> T;

        while(T-- > 0){
            // get the number of professors
            int N;
            cin >> N;

            // store the start and end time in ascending order
            vector<ScheduleEvent> events;
            events.reserve(2 * N);

            // check whether same schedule as prof 2 or prof 1 exists for any other professor
            // in that case we need to check for the clash between prof 1 and 2 below, that will
            // lead to unnecessary extra room
            bool samexists = false;
```

```

for (int i = 0; i < N; i++) {
    // get the professor schedules
    int start, end;
    cin >> start >> end;

    ScheduleEvent startEvent = {start, i, true};
    ScheduleEvent endEvent = {end, i, false};

    events.push_back(startEvent);
    events.push_back(endEvent);

    // check the current schedule whether it clashes with the schedule of prof 2 or 1
    if(i >= 2 && samexists == false){
        if((events[0].time == startEvent.time && events[1].time == endEvent.time)
            || (events[2].time == startEvent.time && events[3].time == endEvent.time)){
            samexists = true;
        }
    }
}

// sort the events array with ascending order of starting
// and then ending time
sort(events.begin(), events.end());

// a map of prof no and room allocated to him
vector<int> profRoomMap(N, -1);
// a list for the rooms used
vector<bool> rooms;

int roomCount = 0;
int busyRooms = 0;

for(int i = 0, len = events.size(); i < len; i++){
    if(events[i].isStart){
        if(roomCount == busyRooms){
            // create a new room and associate with the professor
            rooms.push_back(true);
            profRoomMap[events[i].prof] = roomCount;
            roomCount = roomCount + 1;
        }else{
            bool isRoomAssigned = false;
            // try to schedule the current event in question to be as close as to the previous
            // schedule which has just ended. In order to achieve that, traverse back from the last
            // ended schedule to the start of the events list. If a schedule has ended just before
            // and the room corresponding to it is not busy, assign the current schedule to it.
            // This is to ensure, that in a room, there is minimum gap between assigned schedules
            for(int j = i - 1; j > 0 && isRoomAssigned == false; j--){
                // However, if the professor of the current schedule to be allocated is 1 or 2
                // then we have to check that 2 does not get allocated to room allocated to 1 and vice versa
                if(events[j].isStart == false && rooms[profRoomMap[events[j].prof]] == false) {
                    if((samexists == false)
                        && ((events[i].prof == 0 && profRoomMap[events[j].prof] == profRoomMap[1])
                            || (events[i].prof == 1 && profRoomMap[events[j].prof] == profRoomMap[0]))) {
                        continue;
                    }else{
                        rooms[profRoomMap[events[j].prof]] = true;
                        profRoomMap[events[i].prof] = profRoomMap[events[j].prof];
                        isRoomAssigned = true;
                    }
                }
            }
            // control will come to this block in the event when only one room is available that is
            // associated with either prof 1 or 2 while the current event is for prof 2 or 1
            if(isRoomAssigned == false){
                rooms.push_back(true);
                profRoomMap[events[i].prof] = roomCount;
                roomCount = roomCount + 1;
            }
        }
        busyRooms++;
    }else{
        rooms[profRoomMap[events[i].prof]] = false;
        busyRooms--;
    }
}

```



```

    }
}

printf("%d\n", roomCount);
}
};

int main(){
    TwoProfessors::main();
    return 0;
}

```

Source Code (Java):

```

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Queue;

class Event implements Comparable<Event> {
    public int pos, id, isStart;

    public Event(int pos, int id, int isStart){
        this.pos = pos;
        this.id = id;
        this.isStart = isStart;
    }

    @Override
    public int compareTo(Event e) {
        if(e == null) return -1;
        if (pos != e.pos) return pos < e.pos ? -1 : 1;
        return isStart < e.isStart ? -1 : 1;
    }
}

public class TwoProfessorsAlfonso {
    static final int MAXN = 1 << 17;
    static int N, ans;
    static int room[] = new int[MAXN];
    static int boss[] = new int[MAXN];
    static int from[] = new int[MAXN];
    static int busyRooms[] = new int[MAXN];

    static boolean canUniquelyReach(int a, int b)
    {
        while (true)
        {
            if (a == b)
                return true;

            if (ans - busyRooms[a] > 1)
                break;

            a = from[a];
            if (a < 0) break;
        }

        return false;
    }

    public static void main(String[] args) throws Exception {

        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        int T = Integer.parseInt(br.readLine());

        while(T-- > 0)
        {
            N = Integer.parseInt(br.readLine());

```

```

List<Event> events = new ArrayList<Event>(N * 2);

for(int i = 0; i < N; i++) {
    String params[] = br.readLine().split("\\s");

    int start = Integer.parseInt(params[0]);
    int end = Integer.parseInt(params[1]);

    events.add(new Event(start, i, 1));
    events.add(new Event(end, i, 0));

    boss[i] = -1;
}

Queue<Integer> Q = new PriorityQueue<Integer>();
Collections.sort(events);

ans = 0;
for (int i = 0, j; i < events.size(); i = j)
{
    int currBusy = ans - Q.size();
    for (j = i; j < events.size() && events.get(j).pos == events.get(i).pos; j++)
    {
        if (events.get(j).isStart != 0) {
            if (Q.isEmpty())
                Q.add(ans++);

            int prof = events.get(j).id;

            busyRooms[prof] = currBusy;
            room[prof] = Q.peek();
            from[prof] = boss[Q.peek()];
            boss[Q.peek()] = prof;

            Q.poll();
        }
        else
        {
            currBusy--;
            Q.add(room[ events.get(j).id ]);
        }
    }
}

if (room[0] == room[1])
{
    if (canUniquelyReach(0, 1) || canUniquelyReach(1, 0))
        ans++;
}

System.out.println(ans);
}
}
}

```

22. Travelling Shoemaker

Once upon a time there was a very peaceful country named Nlogonia. Back then, Poly the Shoemaker could come to the country and travel freely from city to city doing his job without any harassment. This task was very easy, as every city in Nlogonia had a direct road to every other city in the country. He could then easily travel the whole country visiting each city exactly once and fixing everybody's shoes.

But not anymore. The times have changed and war has come to Nlogonia. The age when people could travel freely is over.

Confederations identified by colors were formed among the cities all over the country, and now each city belongs to at least one and at most two confederations. When trying to enter a city, you must give to the border officer a ticket from one of the confederations this city belongs to. When leaving the city, you receive a ticket from the other confederation the city belongs to (i.e. different from the one you gave when entering) or from the same confederation if the city only belongs to one.

As Poly the Shoemaker is a long time friend of Nlogonia, he is allowed to choose a ticket and a city he wants to enter as the first city in the country, but after that he must obey the confederations rules. He wants to do the same routine he did before, visiting each city exactly once in Nlogonia, but now it's not easy for him to do this, even though he can choose where to start his journey.

For example, suppose there are four cities, labeled from 0 to 3. City 0 belongs to confederations red and green; city 1 belongs only to red; city 2 belongs to green and yellow; and city 3 belongs to blue and red. If Poly the Shoemaker chooses to start at city 0, he can enter it carrying either the red or the green ticket and leave receiving the other. Should he choose the red ticket, he will leave with a green ticket, and then there is only city 2 he can travel to. When leaving city 2 he receives the yellow ticket and now can't go anywhere else. If he had chosen the green ticket as the first he would receive the red one when leaving, and then he could travel to cities 1 or 3. If he chooses city 3, when leaving he will receive the blue ticket and again can't go anywhere else. If he chooses city 1, he receives the red ticket again when leaving (city 1 belongs only to the red confederation) and can only travel to city 3 and will never get to city 2. Thus, it is not possible to visit each city exactly once starting at city 0. It is possible, however, starting at city 2 with the yellow ticket, leaving the city with the green ticket, then visiting city 0, leaving with red ticket, then visiting city 1, leaving with red ticket again and, at last, visiting city 3.

As you can see, it got really difficult for Poly the Shoemaker to accomplish the task, so he asks you to help him. He wants to know if it's possible to choose a city to start such that he can travel all cities from Nlogonia exactly once.

Can you help Poly the Shoemaker?

Input

The input contains several test cases. The first line of a test case contains two integers N and C , separated by one space, indicating respectively the number of cities ($1 \leq N \leq 500$) and confederations ($1 \leq C \leq 100$) in the country. Each of the next C lines describes a confederation. It starts with one integer K ($0 \leq K \leq N$) and then K integers representing the cities which belong to this confederation. All integers are separated by single spaces and cities are numbered from 0 to $N - 1$. Each city will appear at least once and at most twice and no city will be repeated on the same confederation.

The end of input is indicated by a line containing two zeroes separated by a single space.

Output

For each test case in the input, your program must print a single line, containing the integer -1 if it's not possible to match the requirements or one integer representing the city where Poly the Shoemaker can start his journey. If there are multiple correct answers, print the smallest one.

Example

Input:

```
4 4
1 3
3 0 1 3
2 0 2
1 2
3 4
1 0
3 0 1 2
1 1
1 2
3 4
1 1
2 1 0
2 0 2
1 2
5 5
3 0 1 2
2 1 3
2 2 3
2 0 4
1 4
6 6
2 0 1
2 0 2
2 1 2
2 3 4
2 3 5
2 4 5
2 4
1 0
2 0 1
1 1
0
0 0
```

Output:

```
2
-1
1
1
-1
0
```

Time limit: 1s

Source limit: 50000B

Solution:

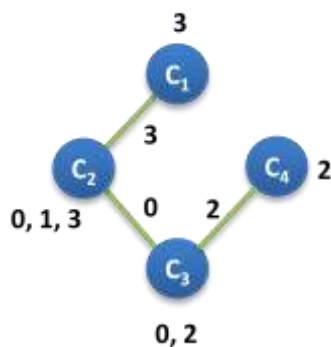
In graph theory, an **Eulerian path** is a path in a graph which visits each edge exactly once. Similarly, an **Eulerian circuit** is an Eulerian path which starts and ends on the same vertex. They were first discussed by Leonhard Euler while solving the famous Seven Bridges of Königsberg problem in 1736. So, does the problem statement sound similar? If we consider the Confederations as Vertices and Cities as the Edges, then the given scenario can be modelled as an Eulerian Path problem http://en.wikipedia.org/wiki/Eulerian_path.

For the existence of Eulerian paths it is necessary that no more than two vertices have an odd degree (degree of a vertex means the number of edges incident on it, an example of an odd degree vertex would be a vertex which has 1, 3 or any odd number of edges). If there are no vertices of odd degree, all Eulerian paths are circuits. If there are exactly two vertices of odd degree, all Eulerian paths start at one of them and end at the other. Sometimes a graph that has an Eulerian path, but not an Eulerian circuit (in other words, it is an open path, and does not start and end at the same vertex) is called **semi-Eulerian**.

Below are few graphs with confederations as vertices and cities as edges.

Eulerian Path

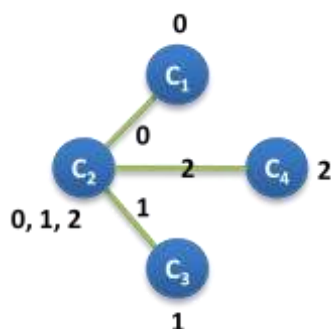
3	4		
1	3		C ₁
3	0	1	C ₂
2	0	2	C ₃
1	2		C ₄



The number odd degree vertices in this graph is 2 i.e. C₁ and C₂, who have 1 edge each. Since, there are exactly two vertices of odd degree, the graph has an Eulerian Path, which starts from either C₁ and ends at C₄ or vice versa. If one selects C₁, then city to start with is 3(edge), whereas if one selects C₄, then the city to start with is 2(edge). The graph is semi-Eulerian.

Non Eulerian Graph

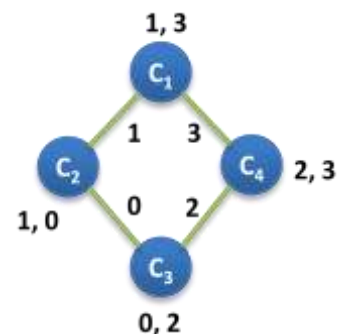
3	4		
1	0		C ₁
3	0	1	C ₂
1	1		C ₃
1	2		C ₄



The number of odd degree vertices in this graph is 4 i.e. C₁, C₂, C₃ and C₄, who have 1 edge each. Since, there are more than two vertices of odd degree, the graph does not have an Eulerian Path. Hence, we cannot have a path that visits all cities exactly once.

Eulerian Circuit

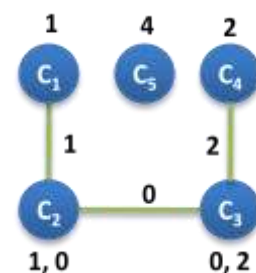
3	4		
2	1	3	C ₁
2	1	0	C ₂
2	0	2	C ₃
2	2	3	C ₄



The number of odd degree vertex in this graph is 0 because all the vertices have degree equal to 2. Since there are no vertex of odd degree, all Eulerian paths are circuits. So we can start from any vertex C₁, C₂, C₃ or C₄. We select C₃ here because C₃ has an edge 0 which is least city of all. If one chooses any other vertex, then edge connected to it gives the city from where he starts.

Isolated

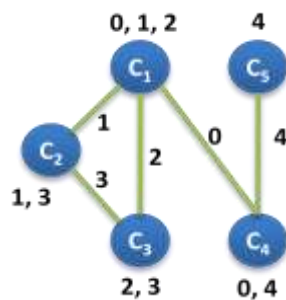
3	5		
1	1		C ₁
2	1	0	C ₂
2	0	2	C ₃
1	2		C ₄
1	4		C ₅



Though the number of odd degree vertices in this graph is 2 i.e. C₁ and C₄, who have 1 edge each still the graph is non Eulerian, because we have a vertex C₅ which is disconnected. Hence, we cannot have a path that visits all cities exactly once.

Eulerian Graph (with bridge)

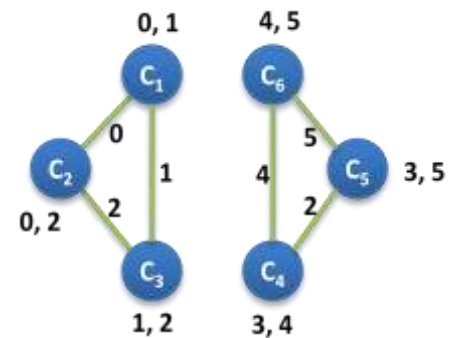
5	5			
3	0	1	2	C ₁
2	1	3		C ₂
2	2	3		C ₃
2	0	4		C ₄
1	4			C ₅



This graph is interesting. The number of odd degree vertices in this graph is 2 i.e. C₁ and C₅, who have 3 edges and 1 edge respectively. So, we can start from either C₁ and end at C₅, or start from C₅ and end at C₁. If we consider C₁, then we can take start city as 0, 1 or 2. We would be tempted to choose city 0 as our starting point because it is the least one. However, edge(city) 0 is a bridge here. It connects left and right half of the graph. If we remove edge(city) 0, then the graph will have two disconnected subcomponents. Also from the picture it is evident that if we select our starting city as 0, then it is not possible to traverse all edges without travelling edge 0 twice.

Disconnected Components

6	6			
2	0	1		C ₁
2	0	2		C ₂
2	1	2		C ₃
2	3	4		C ₄
2	3	5		C ₅
2	4	5		C ₆



This graph is also interesting because it contains two disconnected subcomponents, but each of the subcomponent are Eulerian themselves. So, it might be possible, we might end up finding Eulerian paths although the graph has subcomponents. In order to check whether a graph has subcomponents or is disconnected, we must run "Depth First Search" on the entire graph to ensure that all vertices are connected. If any vertex is disconnected, then also the graph is not Eulerian.

In the mathematical field of graph theory, a **Hamiltonian path** (or traceable path) is a path in an undirected graph which visits each vertex exactly once. A **Hamiltonian cycle** (or **Hamiltonian circuit**) is a cycle in an undirected graph which visits each vertex exactly once and also returns to the starting vertex. So, does the problem statement sound similar? If we consider the Cities as Vertices and Confederations as the Edges, then the given scenario can be modelled as a Hamiltonian Path problem http://en.wikipedia.org/wiki/Hamilton_path.

So, once we determine whether the given scenario has an Eulerian Path and our potential cities to start with, we can then very well find the Hamiltonian Path starting with the minimum city and check whether a path exists starting with that city or not. If it does not exist, we then take the next least city from the set of potential cities and continue until we find our desired minimum city.

Source Code (C++):

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <set>
using namespace std;

struct Vertex;
struct Edge;

struct Confederation {
    int confNo;
    vector<int> cities;
};

enum State {
    Unvisited, Visiting, Visited
};

struct Vertex {
    int vertex;
    State vertexstate;
    vector<Edge> edges;
};
```

```

    void Vertex::reset();
};

struct Edge {
    Vertex *firstVertex, *secondVertex;
    int value;
    State edgestate;
    Vertex* Edge::getOtherVertex(Vertex *startVertex);
};

struct Graph {
    int vertexCount;
    int oddCount;
    bool connected;
    vector<Vertex> vertices;

    void addVertex(int vertex);

    Vertex* getVertex(int vertex);

    void addEdge(int startVertex, int endVertex, int edgeValue);

    void addEdge(int startVertex, int endVertex);

    void complete();

    void runDFS(Vertex *startVertex);

    bool walkHamilton(Vertex *startVertex, Vertex *endVertex, int pathLength);

    bool isEulerian();

    void reset();
};

Vertex* Edge::getOtherVertex(Vertex *startVertex) {
    if(firstVertex->vertex == startVertex->vertex){
        return secondVertex;
    }else{
        return firstVertex;
    }
}

void Vertex::reset() {
    for (int i = 0, len = edges.size(); i < len; i++) {
        edges[i].edgestate = Unvisited;
    }
}

void Graph::addVertex(int vertex) {
    Vertex newVertex = {vertex, Unvisited};
    vertices.push_back(newVertex);
}

Vertex* Graph::getVertex(int vertex) {
    for(int i = 0, verticesSize = vertices.size(); i < verticesSize; i++){
        if(vertices[i].vertex == vertex){
            return &vertices[i];
        }
    }
}

void Graph::addEdge(int startVertex, int endVertex, int edgeValue){
    Vertex *firstVertex, *secondVertex;

    // find the corresponding vertices
    for(int i = 0, verticesSize = vertices.size(); i < verticesSize; i++){
        Vertex v = vertices[i];
        if(v.vertex == startVertex){
            firstVertex = &v;
        }else if(v.vertex == endVertex){
            secondVertex = &v;
        }
    }
}

```

```

    }
}

// create a new Edge
Edge e = {getVertex(startVertex), getVertex(endVertex), edgeValue, Unvisited};
getVertex(startVertex)->edges.push_back(e);
getVertex(endVertex)->edges.push_back(e);
}

void Graph::addEdge(int startVertex, int endVertex){
    addEdge(startVertex, endVertex, -1);
}

void Graph::complete() {
    if (vertexCount > 1) {

        // calculate oddCount
        for(int i = 0, verticesSize = vertices.size(); i < verticesSize; i++){
            if (vertices[i].edges.size() % 2 != 0) {
                oddCount++;
            }
        }

        // run DFS to check if it is connected
        runDFS(&vertices[0]);

        int visitedCount = 0;
        for(int i = 0, verticesSize = vertices.size(); i < verticesSize; i++){
            if (vertices[i].vertexstate == Visited) {
                visitedCount++;
            }
        }

        connected = visitedCount == vertexCount ? true : false;

        // since we have runDFS so we to reset all nodes to Unvisited again
        reset();
    }
}

void Graph::runDFS(Vertex* startVertex) {
    startVertex->vertexstate = Visiting;

    for(int i = 0, edgeSize = startVertex->edges.size(); i < edgeSize; i++){
        Edge* e = &startVertex->edges[i];
        if(e->getOtherVertex(startVertex)->vertexstate == Unvisited){
            runDFS(e->getOtherVertex(startVertex));
        }
    }
    startVertex->vertexstate = Visited;
}

bool Graph::walkHamilton(Vertex* startVertex, Vertex* endVertex, int pathLength){
    if(startVertex->vertex == endVertex->vertex){
        return pathLength == 0;
    }

    startVertex->vertexstate = Visited;

    for(int i = 0, edgeSize = startVertex->edges.size(); i < edgeSize; i++){
        Edge* nextStartEdge = &startVertex->edges[i];
        Vertex* nextVertex = nextStartEdge->getOtherVertex(startVertex);
        if(nextVertex->vertexstate != Visited){
            if(walkHamilton(nextVertex, endVertex, pathLength - 1) == true){
                return true;
            }
        }
    }
    startVertex->vertexstate = Unvisited;
    return false;
}

bool Graph::isEulerian() {
    if (connected == false || oddCount > 2) {

```



```

        return false;
    }
    return true;
}

void Graph::reset() {
    for(int i = 0, verticesSize = vertices.size(); i < verticesSize; i++){
        vertices[i].reset();
    }
}

class TravellingShoeMaker {
public:
    static void main(){
        while(true){
            // get the number of cities and confederations
            int N, C;
            cin >> N >> C;

            if (N == 0 && C == 0) {
                break;
            }

            // get all the confederations
            vector<Confederation> confederations;

            for (int conf = 0, confIndex = 0; conf < C; conf++) {
                // get the number of cities in current confederation
                int cityCount;
                cin >> cityCount;

                if (cityCount > 0) {
                    Confederation confederation = {confIndex};

                    // get the cities of the confederation
                    for (int j = 1; j <= cityCount; j++) {
                        int city;
                        cin >> city;
                        confederation.cities.push_back(city);
                    }

                    confederations.push_back(confederation);
                    confIndex++;
                }
            }

            // recalculate, some confederation may be without any city, safely ignore them
            C = confederations.size();

            // We model the confederations as a graph, where each confederation is a node
            // and all the cities are edges. We need to visit each city exactly once. Hence
            // it can be modeled as a graph and check for Eulerian path

            Graph confGraph = {C, 0, true};

            // add each confederation as a vertex in the graph
            for(int i = 0, confederationSize = confederations.size(); i < confederationSize; i++){
                confGraph.addVertex(confederations[i].confNo);
            }

            // create the edges between the vertices
            for (int conf = 0; conf < C - 1; conf++) {
                for (int nextconf = conf + 1; nextconf < C; nextconf++) {
                    for(int i = 0, curConfCityIndex = confederations[conf].cities.size(); i < curConfCityIndex; i++){
                        int matchCity = confederations[conf].cities[i];
                        for(int j = 0, nextConfCityIndex = confederations[nextconf].cities.size(); j < nextConfCityIndex; j++){
                            int nextcity = confederations[nextconf].cities[j];
                            if (matchCity == nextcity) {
                                confGraph.addEdge(conf, nextconf, matchCity);
                            }
                        }
                    }
                }
            }
        }
    }
}

```

```

    }
    }
}

// finish the graph
confGraph.complete();

if (C == 1) {
    // if there is only one confederation, then take the least city,
    // which is 0, since input cities are given from 0 to N - 1.
    printf("%d\n", 0);
} else if (confGraph.isEulerian() == false) {
    // if the graph is disconnected or has odd number of odd degree vertices
    // then no Eulerian Path (or a path that visits each edge exactly once)
    // is not possible
    printf("%d\n", -1);
} else {
    // When the graph has Eulerian Path and has 0 odd degree vertex, it is an Eulerian
    // circuit. The graph can be travelled starting from any node(confederation). Hence
    // we take the least city, which is 0, since input cities are given from 0 to N - 1.
    if (confGraph.oddCount == 0) {
        printf("%d\n", 0);
    } else {
        int minCity = 500; // the max value of N

        // From the graph find the nodes which have odd degree. It is known that
        // for an Eulerian path to exist, the number of odd degree paths must be
        // atmost 2 and is either 0 or 2. So, the path will start from one node
        // and end in the other. Now each node has a number of edges. So, we have
        // to find the least starting city connecting these two nodes

        vector< set<int> > listOfOddNodes;
        for (int i = 0, verticesSize = confGraph.vertices.size(); i < verticesSize; i++) {
            Vertex v = confGraph.vertices[i];

            if (v.edges.size() % 2 != 0) {
                set<int> oddCities(confederations[v.vertex].cities.begin(),
                    confederations[v.vertex].cities.end());
                listOfOddNodes.push_back(oddCities);
            }
        }

        // Now we make a new graph comprising of city as nodes, to trace the actual
        // path. The graph should have a path, where all node(city) is visited exactly once
        set<int> allCities;

        for (int i = 0, verticesSize = confGraph.vertices.size(); i < verticesSize; i++) {
            Vertex v = confGraph.vertices[i];
            allCities.insert(confederations[v.vertex].cities.begin(),
                confederations[v.vertex].cities.end());
        }

        // New graph of cities as nodes
        Graph cityGraph = {allCities.size(), 0, true};

        // add each city as node in the graph
        set<int>::iterator p = allCities.begin();
        while (p != allCities.end()) {
            cityGraph.addVertex(*p);
            p++;
        }

        // Now create the edges between the city nodes.
        // That can be done from our initial input
        for (int conf = 0, confederationSize = confederations.size(); conf < confederationSize; conf++) {
            int cityCount = confederations[conf].cities.size();
            for (int firstCityIndex = 0; firstCityIndex < cityCount - 1; firstCityIndex++) {
                for (int secondCityIndex = firstCityIndex + 1; secondCityIndex < cityCount;
                    secondCityIndex++) {
                    cityGraph.addEdge(confederations[conf].cities[firstCityIndex],
                        confederations[conf].cities[secondCityIndex]);
                }
            }
        }
    }
}

```

```

    }
}

// Find the Hamilton path
set<int>::iterator startCity = listOfOddNodes[0].begin();
while(startCity != listOfOddNodes[0].end()){
    set<int>::iterator endCity = listOfOddNodes[1].begin();
    while(endCity != listOfOddNodes[1].end()){
        if(*startCity < minCity || *endCity < minCity){
            if(cityGraph.walkHamilton(cityGraph.getVertex(*startCity),
                cityGraph.getVertex(*endCity), cityGraph.vertexCount - 1) == true){
                minCity = *startCity < *endCity ? *startCity : *endCity;
            }
            cityGraph.reset();
        }
        endCity++;
    }
    startCity++;
}

printf("%d\n", minCity);
}
}
}
};

int main(){
    TravellingShoeMaker::main();
    return 0;
}

```

Source Code (Java):

```

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Set;
import java.util.TreeSet;

class Confederation {
    private int confNo;
    private List<Integer> cities = new LinkedList<Integer>();

    public Confederation(int confNo){
        this.confNo = confNo;
    }
    public void addCity(int city){
        cities.add(city);
    }
    public int getConfNo() {
        return confNo;
    }
    public List<Integer> getCities() {
        return cities;
    }
}

enum State {
    Unvisited, Visiting, Visited
}

class Graph {
    private int vertexCount;
    private List<Vertex> vertices;
    private int oddCount;
    private boolean connected;

    public Graph(int vertexCount) {
        this.vertexCount = vertexCount;
        this.vertices = new LinkedList<Vertex>();
    }
}

```

```

}

public void complete() {
    if (vertexCount > 1) {

        // calculate oddCount
        for (Vertex v : vertices) {
            if (v.getEdges().size() % 2 != 0) {
                oddCount++;
            }
        }

        // run DFS to check if it is connected
        runDFS(vertices.get(0));

        int visitedCount = 0;
        for (Vertex v : vertices){
            if(v.getVertexstate() == State.Visited){
                visitedCount++;
            }
        }
        connected = visitedCount == vertexCount ? true : false;

        // since we have runDFS so we to reset all nodes to Unvisited again
        reset();
    }
}

public void runDFS(Vertex startVertex) {
    startVertex.setVertexstate(State.Visiting);

    for(Edge e : startVertex.getEdges()){
        if(e.getOtherVertex(startVertex).getVertexstate() == State.Unvisited){
            runDFS(e.getOtherVertex(startVertex));
        }
    }
    startVertex.setVertexstate(State.Visited);
}

public boolean isEulerian() {
    if (isConnected() == false || getOddCount() > 2) {
        return false;
    }
    return true;
}

public void reset() {
    for (Vertex v : vertices) {
        v.reset();
    }
}

public int getSize() {
    return vertexCount;
}

public boolean walkHamilton(Vertex startVertex, Vertex endVertex, int d){
    if(startVertex.getVertex() == endVertex.getVertex()){
        return d == 0;
    }

    startVertex.setVertexstate(State.Visited);

    for(Edge nextStartEdge : startVertex.getEdges()){
        Vertex nextVertex = nextStartEdge.getOtherVertex(startVertex);
        if(nextVertex.getVertexstate() != State.Visited){
            if(walkHamilton(nextVertex, endVertex, d - 1) == true){
                return true;
            }
        }
    }
    startVertex.setVertexstate(State.Unvisited);
    return false;
}

```

```

}

public void addVertex(int vertex) {
    Vertex newVertex = new Vertex(vertex);
    vertices.add(newVertex);
}

public void addEdge(int startVertex, int endVertex, int edgeValue){
    Vertex firstVertex = null, secondVertex = null;

    // find the corresponding vertices
    for(Vertex v: vertices){
        if(v.getVertex() == startVertex){
            firstVertex = v;
        }else if(v.getVertex() == endVertex){
            secondVertex = v;
        }
    }

    // create a new Edge
    Edge e = new Edge(firstVertex, secondVertex, edgeValue);
    firstVertex.addEdge(e);
    secondVertex.addEdge(e);
}

public void addEdge(int startVertex, int endVertex){
    addEdge(startVertex, endVertex, -1);
}

private boolean isConnected() {
    return connected;
}

public int getOddCount() {
    return oddCount;
}

public Vertex getVertex(int vertex) {
    for(Vertex v: vertices){
        if(v.getVertex() == vertex){
            return v;
        }
    }
    return null;
}

public List<Vertex> getVertices() {
    return vertices;
}
}

class Vertex {
    private int vertex;
    private List<Edge> edges;
    private State vertexstate = State.Unvisited;

    public Vertex(int vertex) {
        this.vertex = vertex;
        this.edges = new LinkedList<Edge>();
    }

    public int getVertex() {
        return vertex;
    }

    public List<Edge> getEdges() {
        return edges;
    }

    public void addEdge(Edge e) {
        edges.add(e);
    }

    public State getVertexstate() {

```

```

        return vertexstate;
    }

    public void setVertexstate(State vertexstate) {
        this.vertexstate = vertexstate;
    }

    public void reset() {
        vertexstate = State.Unvisited;
        for (Edge e : edges) {
            e.setEdgestate(State.Unvisited);
        }
    }
}

class Edge {
    private Vertex firstVertex;
    private Vertex secondVertex;
    private int value;
    private State edgestate = State.Unvisited;

    public Edge(Vertex firstVertex, Vertex secondVertex, int value) {
        this.firstVertex = firstVertex;
        this.secondVertex = secondVertex;
        this.value = value;
    }

    public Edge(Vertex firstVertex, Vertex secondVertex) {
        this(firstVertex, secondVertex, -1);
    }

    public Vertex getOtherVertex(Vertex startVertex) {
        if(this.firstVertex.getVertex() == startVertex.getVertex()){
            return secondVertex;
        }else{
            return firstVertex;
        }
    }

    public int getValue() {
        return value;
    }

    public State getEdgestate() {
        return edgestate;
    }

    public void setEdgestate(State edgestate) {
        this.edgestate = edgestate;
    }
}

public class TravellingShoemaker {

    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        while (true) {
            String params[] = br.readLine().split("\\s");

            int N = Integer.parseInt(params[0]);
            int C = Integer.parseInt(params[1]);

            if (N == 0 && C == 0) {
                break;
            }

            // get all the confederations
            List<Confederation> confederations = new LinkedList<Confederation>();

            for (int conf = 0, confIndex = 0; conf < C; conf++) {
                params = br.readLine().split("\\s");
                int cityCount = Integer.parseInt(params[0]);
            }
        }
    }
}

```

```

    if (cityCount > 0) {

        Confederation confederation = new Confederation(confIndex);

        for (int j = 1; j <= cityCount; j++) {
            int city = Integer.parseInt(params[j]);
            confederation.addCity(city);
        }

        confederations.add(confederation);
        confIndex++;
    }

    // recalculate, some confederation may be without any city, safely ignore them
    C = confederations.size();

    // We model the confederations as a graph, where each confederation is a node
    // and all the cities are edges. We need to visit each city exactly once. Hence
    // it can be modeled as a graph and check for Eulerian path

    Graph confGraph = new Graph(C);

    // add each confederation as a vertex in the graph
    for (Confederation conf : confederations) {
        confGraph.addVertex(conf.getConfNo());
    }

    // create the edges between the vertices
    for (int conf = 0; conf < C - 1; conf++) {
        for (int nextconf = conf + 1; nextconf < C; nextconf++) {
            for (int matchCity : confederations.get(conf).getCities()) {
                for (int nextcity : confederations.get(nextconf).getCities()) {
                    if (matchCity == nextcity) {
                        confGraph.addEdge(conf, nextconf, matchCity);
                    }
                }
            }
        }
    }

    // finish the graph
    confGraph.complete();

    if (C == 1) {
        // if there is only one confederation, then take the least city,
        // which is 0, since input cities are given from 0 to N - 1.
        System.out.println(0);
    } else if (confGraph.isEulerian() == false) {
        // if the graph is disconnected or has odd number of odd degree vertices
        // then no Eulerian Path (or a path that visits each edge exactly once)
        // is not possible
        System.out.println(-1);
    } else {
        // When the graph has Eulerian Path and has 0 odd degree vertex, it is an Eulerian
        // circuit. The graph can be travelled starting from any node(confederation). Hence
        // we take the least city, which is 0, since input cities are given from 0 to N - 1.
        if (confGraph.getOddCount() == 0) {
            System.out.println(0);
        } else {
            int minCity = 500; // the max value of N

            // From the graph find the nodes which have odd degree. It is known that
            // for an Eulerian path to exist, the number of odd degree paths must be
            // atmost 2 and is either 0 or 2. So, the path will start from one node
            // and end in the other. Now each node has a number of edges. So, we have
            // to find the least starting city connecting these two nodes
            List<Set<Integer>> listOfOddNodes = new ArrayList<Set<Integer>>(2);

            for (Vertex v : confGraph.getVertices()) {
                if (v.getEdges().size() % 2 != 0) {
                    Set<Integer> oddCities = new TreeSet<Integer>();
                    oddCities.addAll(confederations.get(v.getVertex()).getCities());
                    listOfOddNodes.add(oddCities);
                }
            }
        }
    }
}

```

```

    }
    // Now we make a new graph comprising of city as nodes, to trace the actual
    // path. The graph should have a path, where all node(city) is visited exactly once
    Set<Integer> allCities = new TreeSet<Integer>();
    for(Vertex v : confGraph.getVertices()){
        allCities.addAll(confederations.get(v.getVertex()).getCities());
    }

    // New graph of cities as nodes
    Graph cityGraph = new Graph(allCities.size());

    // add each city as node in the graph
    for(Integer city : allCities){
        cityGraph.addVertex(city);
    }

    // Now create the edges between the city nodes.
    // That can be done from our initial input
    for(Confederation conf : confederations){
        List<Integer> cities = conf.getCities();
        int cityCount = cities.size();
        for(int firstCityIndex = 0; firstCityIndex < cityCount - 1; firstCityIndex++){
            for(int secondCityIndex = firstCityIndex + 1; secondCityIndex < cityCount;
                secondCityIndex++){
                cityGraph.addEdge(cities.get(firstCityIndex), cities.get(secondCityIndex));
            }
        }
    }

    // Find the Hamilton path
    for(int startCity : listOfOddNodes.get(0)){
        for(int endCity : listOfOddNodes.get(1)){
            if(startCity < minCity || endCity < minCity){
                if(cityGraph.walkHamilton(cityGraph.getVertex(startCity),
                    cityGraph.getVertex(endCity), cityGraph.getSize() - 1) == true){
                    minCity = startCity < endCity ? startCity : endCity;
                }
                cityGraph.reset();
            }
        }
    }

    System.out.println(minCity);
}
}
}
}
}

```


23. Cross Country

Agness, a student of computer science, is very keen on crosscountry running, and she participates in races organised every Saturday in a big park. Each of the participants obtains a route card, which specifies a sequence of checkpoints, which they need to visit in the given order. Agness is a very attractive girl, and a number of male runners have asked her for a date. She would like to choose one of them during the race. Thus she invited all her admirers to the park on Saturday and let the race decide. The winner would be the one, who scores the maximum number of points. Agnes came up with the following rules:

- a runner scores one point if he meets Agnes at the checkpoint,
- if a runner scored a point at the checkpoint, then he cannot get another point unless he and Agnes move to the next checkpoints specified in their cards.
- route specified by the card may cross the same checkpoint more than once,
- each competitor must strictly follow race instructions written on his card.

Between two consecutive meetings, the girl and the competitors may visit any number of checkpoints. The boys will be really doing their best, so you may assume, that each of them will be able to visit any number of checkpoints whilst Agnes runs between two consecutive ones on her route.

Write a program which for each data set from a sequence of several data sets:

- reads in the contents of Agnes' race card and contents of race cards presented to Tom,
- computes the greatest number of times Tom is able to meet Agnes during the race,
- writes it to output.

Input

There is one integer d in the first line of the input file, $1 \leq d \leq 10$. This is the number of data sets. The data sets follow. Each data set consists of a number of lines, with the first one specifying the route in Agnes' race card. Consecutive lines contain routes on cards presented to Tom. At least one route is presented to Tom. The route is given as a sequence of integers from interval $[1, 1000]$ separated by single spaces. Number 0 stands for the end of the route, though when it is placed at the beginning of the line it means the end of data set. There are at least two and at most 2000 checkpoints in a race card.

Output

The i -th line of the output file should contain one integer. That integer should equal the greatest number of times Tom is able to meet with Agnes for race cards given in the i -th data set.

Example

Sample input:

```
3
1 2 3 4 5 6 7 8 9 0
1 3 8 2 0
2 5 7 8 9 0
1 1 1 1 1 1 2 3 0
1 3 1 3 5 7 8 9 3 4 0
1 2 35 0
0
1 3 5 7 0
3 7 5 1 0
0
1 2 1 1 0
1 1 1 0
0
```

Sample output:

6
2
3

Time limit: 5s

Source limit: 50000B

Solution:

This is an easy 'Dynamic Programing' problem. Let us explain it with an example. Suppose, the set of check points for Agness is 1 2 3 4 5 6 7 8 9 and that of Tom is 1 3 1 3 5 7 8 9 3 4. So, in order for Tom to meet Agness his sequence of checkpoints must coincide with that of Agness. The more the number of sub sequence of checkpoints of Tom coincide with Agness, the more is the number of points that Tom will score and take her on a date.

Agness: **1 2 3 4 5 6 7 8 9**

Tom : **1 3 1 3 5 7 8 9 3 4**

Observing both the sequence, we can find out that the 'Longest Common Subsequence' of checkpoints between the checkpoints of Agness and Tom are 1 3 5 7 8 9. Hence, Tom can meet Agness for 6 times and make 6 points. Thus, the problem reduces to merely finding the longest common subsequence of checkpoint between two pairs of strings and then finding the maximum of all such pairs

Solution(C++):

```
#include <iostream>
#include <string>
#include <vector>
#include <list>
#include <sstream>
using namespace std;

class CrossCountry {
private:
    // Find the longest common subsequence between two strings
    static int LCS(vector<string> X, vector<string> Y) {
        int m = X.size(), n = Y.size();

        int **C = new int*[m + 1];
        for(int i = 0; i <= m; i++){
            C[i] = new int[n + 1];
        }

        // The LCS dp table
        for (int i = 0; i <= m; i++) {
            for (int j = 0; j <= n; j++) {
                if(i == 0 || j == 0){
                    C[i][j] = 0;
                }else if (X[i - 1].compare(Y[j - 1]) == 0) {
                    C[i][j] = 1 + C[i - 1][j - 1];
                } else {
                    C[i][j] = max(C[i][j - 1], C[i - 1][j]);
                }
            }
        }
        return C[m][n];
    }

    static vector<string> tokenize(const string str){
        string buf; // Have a buffer string
        stringstream ss(str); // Insert the string into a stream

        vector<string> tokens; // Create vector to hold our words
    }
```

```

        while (ss >> buf){
            if(buf.compare("0") != 0){
                tokens.push_back(buf);
            }
        }
        return tokens;
    }
}

public:
    static void main() {
        // read the number of datasets
        int d;
        for(scanf("%d\n",&d); d; --d){
            string input;
            getline(cin, input, '\n');

            // read checkpoints of Agnes
            vector<string> AgnesChkPts = tokenize(input);

            // read set of checkpoints for Tom
            list< vector<string> > tomCheckPtsSet;
            getline(cin, input, '\n');
            while(input.compare("0") != 0){
                vector<string> tomCheckPts = tokenize(input);
                tomCheckPtsSet.push_back(tomCheckPts);
                getline(cin, input, '\n');
            }

            // iterate over the set of checkpoints for Tom and find the largest
            // common subsequence of checkpoints of Tom that matches with Agness
            int maxpoints = 0;
            list< vector<string> >::const_iterator p = tomCheckPtsSet.begin();
            while(p != tomCheckPtsSet.end()){
                maxpoints = max(maxpoints, LCS(AgnesChkPts, *p));
                p++;
            }
            printf("%d\n", maxpoints);
        }
    }
};

int main() {
    CrossCountry::main();
    return 0;
}

```

Source Code(Java):

```

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.ArrayList;

public class CrossCountry {

    // Find the longest common subsequence between two strings
    private static int LCS(String[] X, String[] Y) {
        int m = X.length, n = Y.length;

        int C[][] = new int[m + 1][n + 1];

        // The LCS dp table
        for (int i = 0; i <= m; i++) {
            for (int j = 0; j <= n; j++) {
                if(i == 0 || j == 0){
                } else if (X[i - 1].equals(Y[j - 1])) {
                    C[i][j] = 1 + C[i - 1][j - 1];
                } else {
                    C[i][j] = Math.max(C[i][j - 1], C[i - 1][j]);
                }
            }
        }
    }
}

```

```

        return C[m][n];
    }

    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

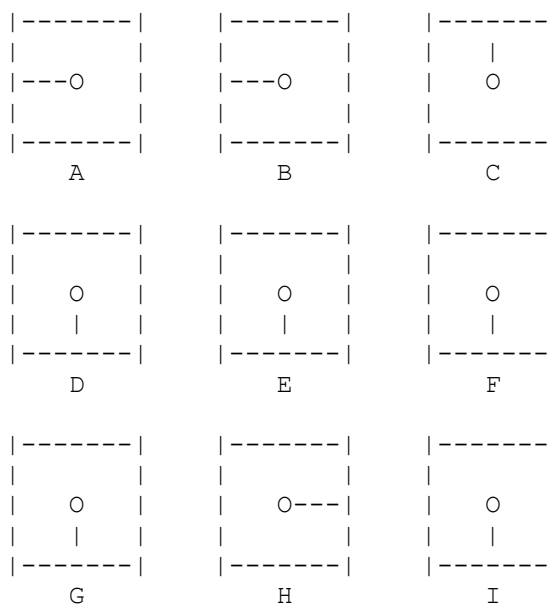
        // read the number of datasets
        int d = Integer.parseInt(br.readLine());
        while (d-- > 0) {
            // read checkpoints of Agnes
            String[] AgnesChkPts = br.readLine().split("\\s");
            ArrayList<String[]> tomCheckPtsSet = new ArrayList<String[]>(1);

            // read set of checkpoints for Tom
            String checkPts[] = br.readLine().split("\\s");
            while (checkPts.length != 1 && checkPts[0] != "0") {
                tomCheckPtsSet.add(checkPts);
                checkPts = br.readLine().split("\\s");
            }

            // iterate over the set of checkpoints for Tom and find the largest
            // common subsequence of checkpoints of Tom that matches with Agness
            int maxpoints = 0;
            for (String[] checkPt : tomCheckPtsSet) {
                maxpoints = Math.max(maxpoints, LCS(AgnesChkPts, checkPt));
            }
            // since all the input checkpoint sequence strings contain a '0' at the end,
            // we always get an extra match. Hence, we need to subtract 1 from maxpoints
            System.out.println(maxpoints - 1);
        }
    }
}

```

24. The Clocks



There are nine clocks in a 3*3 array (figure 1). The goal is to return all the dials to 12 o'clock with as few moves as possible. There are nine different allowed ways to turn the dials on the clocks. Each such way is called a move. Select for each move a number 1 to 9; that number will turn the dials 90° (degrees) clockwise on those clocks which are affected according to table below.

Move	Affected clocks
1	ABDE
2	ABC
3	BCEF
4	ADG
5	BDEFH
6	CFI
7	DEGH
8	GHI
9	EFHI

Input

Read nine numbers from standard input. These numbers give the start positions of the dials. 0=12 o'clock, 1=3 o'clock, 2=6 o'clock, 3=9 o'clock. The example in figure 1 gives the following input data file:

```
3 3 0
2 2 2
2 1 2
```

Output

Write to the standard output the shortest sequence of moves (numbers), which returns all the dials to 12 o'clock. In case there are many solutions, write the solution which is the least in lexicographic order. In our example the output is as follows:

```
4 5 8 9
```

Example

Each number represents a time according to following table:

0 = 12 o'clock
1 = 3 o'clock
2 = 6 o'clock
3 = 9 o'clock

3 3 0		3 0 0		3 0 0		0 0 0		0 0 0
2 2 2	5->	3 3 3	8->	3 3 3	4->	0 3 3	9->	0 0 0
2 1 2		2 2 2		3 3 3		0 3 3		0 0 0

Note: This just represents a valid sequence of moves, and not the solution.

Time limit: 1s
Source limit: 50000B

Solution:

There are 9 different kinds of moves in total, and each move be run for maximum 4 times(0, 1, 2 and 3). Once a single move is run 4 times, the clocks affected by that move again returns to the starting positions of the clocks i.e. they complete one cycle. Hence, there can be atmost $9 \times 4 = 36$ moves. Within these 36 moves all the clocks must return to 12 o' clock position. So, the implementation is simple, for all the 9 moves, run each of them 4 times and check when all clocks read 12 o' clock. To get the least lexicographical order, we store the run count made by each move and arrange the runcount starting from move 1 to move 9.

Source Code (C++):

```
#include <iostream>
#include <conio.h>
using namespace std;

class Clocks {
public:
    static void main(){
        int clockPos[3][3], totalMoves[100], clockMove[10];
        bool solFound = false;

        // get the clock positions
        for(int i = 0; i <= 2; i++){
            for(int j = 0; j <= 2; j++){
                scanf("%d",&clockPos[i][j]);
            }
        }

        // Since there are 9 total moves possible, so we run a nested loop of 9
        // And each move can be run max 4 times, because in that case we
        // again return to our original clock position. So, if there is a
        // solution it is bound to come before we complete 4 times on
        // each of the 9 totalMoves
        for(clockMove[1] = 0; clockMove[1] <= 3; clockMove[1]++){
            for(clockMove[2] = 0; clockMove[2] <= 3; clockMove[2]++){
                for(clockMove[3] = 0; clockMove[3] <= 3; clockMove[3]++){
                    for(clockMove[4] = 0; clockMove[4] <= 3; clockMove[4]++){
                        for(clockMove[5] = 0; clockMove[5] <= 3; clockMove[5]++){
                            for(clockMove[6] = 0; clockMove[6] <= 3; clockMove[6]++){
                                for(clockMove[7] = 0; clockMove[7] <= 3; clockMove[7]++){
                                    for(clockMove[8] = 0; clockMove[8] <= 3; clockMove[8]++){
                                        for(clockMove[9] = 0; clockMove[9] <= 3; clockMove[9]++){
                                            // check whether all clocks are reading 12 o' clock
                                            if(clockPos[0][0] == 0 && clockPos[0][1] == 0 && clockPos[0][2] == 0 &&
                                                clockPos[1][0] == 0 && clockPos[1][1] == 0 && clockPos[1][2] == 0 &&
                                                clockPos[2][0] == 0 && clockPos[2][1] == 0 && clockPos[2][2] == 0){
                                                solFound = true;
                                                break;
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
            clockPos[1][1] = (++clockPos[1][1]) % 4;
        }
    }
};
```

```

        clockPos[1][2] = (++clockPos[1][2]) % 4;
        clockPos[2][1] = (++clockPos[2][1]) % 4;
        clockPos[2][2] = (++clockPos[2][2]) % 4;
    }
    if(solFound == true)
        break;
    clockPos[2][0] = (++clockPos[2][0]) % 4;
    clockPos[2][1] = (++clockPos[2][1]) % 4;
    clockPos[2][2] = (++clockPos[2][2]) % 4;
}
if(solFound == true)
    break;
clockPos[1][0] = (++clockPos[1][0]) % 4;
clockPos[1][1] = (++clockPos[1][1]) % 4;
clockPos[2][0] = (++clockPos[2][0]) % 4;
clockPos[2][1] = (++clockPos[2][1]) % 4;
}
if(solFound == true)
    break;
clockPos[0][2] = (++clockPos[0][2]) % 4;
clockPos[1][2] = (++clockPos[1][2]) % 4;
clockPos[2][2] = (++clockPos[2][2]) % 4;
}
if(solFound == true)
    break;
clockPos[0][1] = (++clockPos[0][1]) % 4;
clockPos[1][0] = (++clockPos[1][0]) % 4;
clockPos[1][1] = (++clockPos[1][1]) % 4;
clockPos[1][2] = (++clockPos[1][2]) % 4;
clockPos[2][1] = (++clockPos[2][1]) % 4;
}
if(solFound == true)
    break;
clockPos[0][0] = (++clockPos[0][0]) % 4;
clockPos[1][0] = (++clockPos[1][0]) % 4;
clockPos[2][0] = (++clockPos[2][0]) % 4;
}
if(solFound == true)
    break;
clockPos[0][1] = (++clockPos[0][1]) % 4;
clockPos[0][2] = (++clockPos[0][2]) % 4;
clockPos[1][1] = (++clockPos[1][1]) % 4;
clockPos[1][2] = (++clockPos[1][2]) % 4;
}
if(solFound == true)
    break;
clockPos[0][0] = (++clockPos[0][0]) % 4;
clockPos[0][1] = (++clockPos[0][1]) % 4;
clockPos[0][2] = (++clockPos[0][2]) % 4;
}
if(solFound == true)
    break;
clockPos[0][0] = (++clockPos[0][0]) % 4;
clockPos[0][1] = (++clockPos[0][1]) % 4;
clockPos[1][0] = (++clockPos[1][0]) % 4;
clockPos[1][1] = (++clockPos[1][1]) % 4;
}
}

int totalMovesIndex = 0;
// for each clock move, find the number of moves and add that many
// number of that move to the total number of moves starting from 1
for(int i = 1; i <= 9; i++){
    for(int j = 1; j <= clockMove[i]; j++){
        totalMoves[totalMovesIndex++] = i;
    }
}
for(int i = 0; i < totalMovesIndex; i++)
    printf("%d ", totalMoves[i]);
}
};

int main(){
    Clocks::main();
    getch();
}

```

```
    return 0;  
}
```

Source Code (Java):

The C++ and Java codes are almost identical, no significance difference is there, hence it is not included here.

25. Garbage

A big office uses a cleaning robot to empty the trash can in every cubicle. The surface of each cubicle is a square and the office floor plan is organized as a rectangular matrix of R rows of C cubicles each. The cleaning process begins by the robot entering the floor by a door that accesses the topmost leftmost cubicle, and finishes with the robot exiting using the same door. Both entering and leaving take 26 seconds each. When the cleaning robot is in a cubicle it can empty the trash can using 13 seconds. The robot can also move to a cubicle that shares a side with its current location, using 38 seconds. The robot needs to enter in each cubicle at least once to empty its trash can.

The total time the robot takes for the entire process depends on the actual tour through the different cubicles. Among all possible tours, we are interested in those of minimum time.

Given the description of the office, you must indicate the minimum time required for the entire cleaning process (including entering, leaving, emptying the trash can in every cubicle, and moving around). Notice that it is possible that an optimal tour passes through a cubicle several times, but the robot has to take the time to empty the trash can only once.

Input

The input contains several test cases, each one described in a single line. The line contains two integers R and C separated by a single space, representing the number of rows and cubicles per row, respectively ($1 \leq R, C \leq 100$). The last line of the input contains the number -1 twice separated by a single space and should not be processed as a test case.

Output

For each test case output a single line with an integer representing the minimum number of seconds that the robot needs to complete the cleaning process.

Example

Input:

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

Output:

```
460
549
1365
1582
970
```

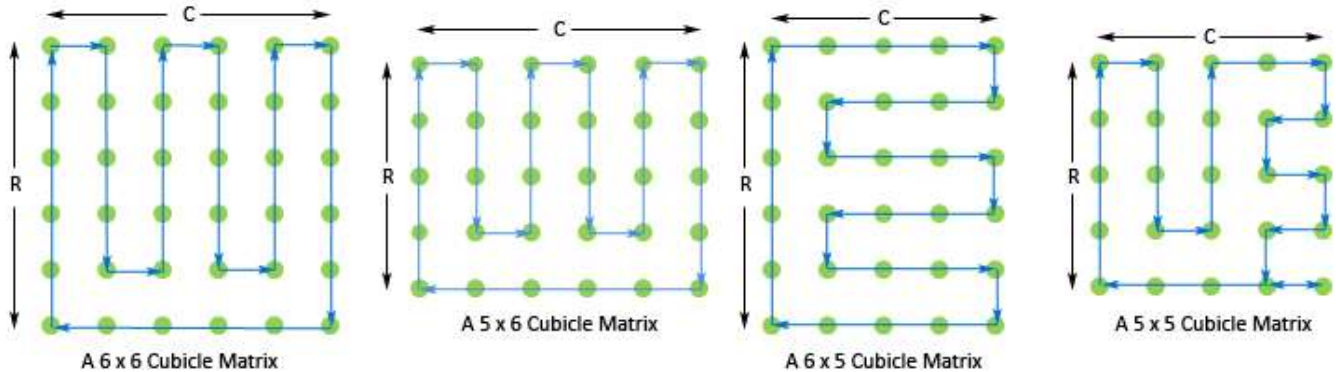
Time limit: 1s

Source limit: 50000B

Solution:

There is a boundary case, where one of the dimensions is 1. This case has to be solved separately. Then try drawing out some larger grids and see if you can construct a Hamiltonian cycle on them. Take row column combinations of even even, even odd, odd even and odd odd. You'll find that most of the time you can find a cycle quite easily. In the case of odd odd you will not be able to construct a cycle. In that case a single vertex will be repeated twice.

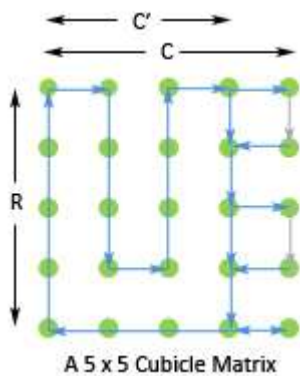
Let us consider the following examples which shows the lowest cost path traversed by the robot for different kinds of combinations.



So, it is evident that when either number of rows or columns or both are even, the path is always following a given pattern. Only when both rows and columns are odd e.g. 5 x 5, one of the edges is traversed twice, see the last figure rightmost bottommost edge.

So, when either the number of rows or columns or both are even, total number of edges traversed,
 $2(C - 1) + 2(R - 1) + (R - 2)(C - 2) = RC$ (1)

So, when the number of rows and columns are both odd, we reduce the matrix, into a submatrix with one of either row or column as even as shown. The downward arrows on the extreme right are now shifted left and are marked grey. The submatrix now reduces to a $R \times C - 1$ matrix, while the last column shows only horizontal arrows. So in the last column, we see $R + 1$ horizontal arrows (remember, the bottommost rightmost edge is travelled twice). So, from (1) and $C' = C - 1$ we can say, that the total number of edges traversed,



$$\begin{aligned} 2(C' - 1) + 2(R - 1) + (R - 2)(C' - 2) + R + 1 \\ = 2(C - 2) + 2(R - 1) + (R - 2)(C - 3) + R + 1 \\ = RC + 1 \end{aligned} \quad \text{.....(2)}$$

Using (1) and (2) we can easily find the time taken by the robot to empty all the garbage in all the cubicles.

Source Code (C++):

```
#include <iostream>
using namespace std;

class Garbage {
public:
    static void main(){
        while(true){
            int R, C, time;
            scanf("%d %d",&R,&C);

            if(R == -1 && C == -1) {
                break;
            }
            // 1. To enter and leave leftmost topmost room, time = 26 x 2 = 52
            // 2. For each edge time taken is 38
            // 3. Since the robot has to visit all cubicles, so R x C x 13 is common for all

            // when there is only 1 row or 1 column handle that case separately
            if(R == 1 && C == 1){
                time = 52 + 13;
            }else{
                if(R == 1){
```

```

        time = 2 * (C - 1) * 38 + 52 + R * C * 13;
    } else if(C == 1){
        time = 2 * (R - 1) * 38 + 52 + R * C * 13;
    } else if(R % 2 == 0 || C % 2 == 0){
        time = R * C * 38 + 52 + R * C * 13;
    } else{
        time = (R * C + 1) * 38 + 52 + R * C * 13;
    }
}
printf("%d\n",time);
}
}
};

int main(){
    Garbage::main();
    return 0;
}

```

Source Code (Java):

```

import java.io.BufferedReader;
import java.io.InputStreamReader;

public class Garbage {

    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        while (true) {
            int time;

            String[] params = br.readLine().split("\\s");
            int R = Integer.parseInt(params[0]);
            int C = Integer.parseInt(params[1]);

            if (R == -1 && C == -1) {
                break;
            }
            // 1. To enter and leave leftmost topmost room, time = 26 x 2 = 52
            // 2. For each edge time taken is 38
            // 3. Since the robot has to visit all cubicles, so R x C x 13 is common for all

            // when there is only 1 row or 1 column handle that case separately
            if (R == 1 && C == 1) {
                time = 52 + 13;
            } else {
                if (R == 1) {
                    time = 2 * (C - 1) * 38 + 52 + R * C * 13;
                } else if (C == 1) {
                    time = 2 * (R - 1) * 38 + 52 + R * C * 13;
                } else if (R % 2 == 0 || C % 2 == 0) {
                    time = R * C * 38 + 52 + R * C * 13;
                } else {
                    time = (R * C + 1) * 38 + 52 + R * C * 13;
                }
            }
            System.out.println(time);
        }
    }
}

```

25. Setnja

In an infinite binary tree:

- Each node has exactly two children - a left and a right child.
- If a node is labeled with the integer X , then its left child is labeled $2*X$ and its right child $2*X+1$.
- The root of the tree is labeled 1.

A walk on the binary tree starts in the root. Each step in the walk is either a jump onto the left child, onto the right child, or pause for rest (stay in the same node).

A walk is described with a string of letters 'L', 'R' and 'P':

- 'L' represents a jump to the left child;
- 'R' represents a jump to the right child;
- 'P' represents a pause.

The value of the walk is the label of the node we end up on. For example, the value of the walk LR is 5, while the value of the walk RPP is 3.

A set of walks is described by a string of characters 'L', 'R', 'P' and '*'. Each '*' can be any of the three moves; the set of walks contains all walks matching the pattern.

For example, the set L^*R contains the walks LLR, LRR and LPR. The set $**$ contains the walks LL, LR, LP, RL, RR, RP, PL, PR and PP.

Finally, the value of a set of walks is the sum of values of all walks in the set.

Calculate the value of the given set of walks.

Input

A string describing the set. Only characters 'L', 'R', 'P' and '*' will appear and there will be at most 10000 of them.

Output

Output the value of the set.

Example

Input:

```
L*R
R**LR
L*R*
L*****P*****RPL*R*LRLLR*L**R**L
```

Output:

```
25
341
128
488320323090939074
```

Time limit: 1s

Source limit: 50000B

Solution:

The problem on the first instance seems to be easy and tempts us to find all the possible combinations of the input string replacing the '*' by 'L', 'P' and 'R'. However, since the number of characters in the string can be up to 10000, hence in the worst case when all characters are '*' the set of all possible combinations will contain 3^{10000} distinct strings. That is a HUGE number and the algorithm will never complete in 1sec. Moreover, the value of a set of walks is the sum of values of all walks in the set, which also means that in the worst case the result will have a HUGE number of digits which is impossible to store in any available data types.

So, instead of calculating the value for each string in the set and then finding the sum, we would try to figure out a formula for the value of a set or sum as a whole. Let us take some examples and try to deduce a formula for the sum. We take here three different kinds of expressions viz. "R***", "R**L" and "R**R"

Expr	R	*	*	*
	x	x	x	x
				2x
				2x + 1
			2x	2x
				4x
				4x + 1
			2x + 1	2x + 1
				4x + 2
				4x + 3
		2x	2x	2x
				4x
				4x + 1
			4x	4x
				8x
				8x + 1
			4x + 1	4x + 1
				8x + 2
				8x + 3
		2x + 1	2x + 1	2x + 1
				4x + 2
				4x + 3
			4x + 2	4x + 2
				8x + 4
				8x + 5
			4x + 3	4x + 3
				8x + 6
				8x + 7
Sum	x	5x + 1	$25x + 8$ $= 5(5x + 1) + 3$	$125x + 49$ $= 5(25x + 8) + 3^2$

Expr	R	*	*	L
	x	x	x	2x
			2x	4x
			2x + 1	4x + 2
		2x	2x	4x
			4x	8x
			4x + 1	8x + 2
		2x + 1	2x + 1	4x + 2
			4x + 2	8x + 4
			4x + 3	8x + 6
Sum		5x + 1	$25x + 8$ $= 5(5x + 1) + 3$	$50x + 16$ $= 2(25x + 8)$

Expr	R	*	*	R
	x	x	x	2x + 1
			2x	4x + 1
			2x + 1	4x + 3
		2x	2x	4x + 1
			4x	8x + 1
			4x + 1	8x + 3
		2x + 1	2x + 1	4x + 3
			4x + 2	8x + 5
			4x + 3	8x + 7
Sum		5x + 1	$25x + 8$ $= 5(5x + 1) + 3$	$50x + 25$ $= 2(25x + 8) + 3^2$

If we denote the number of stars encountered so far by the variable "starCount" and starting value of the expression by "x", it is evident from the above chart that the value of the sum when we encounter the next '*' is

given by $5x + 3^{\text{starCount} - 1}$. Whereas, if we are encountering a 'L' or a 'R' the the value of the sum is given by $2x$ and $2x + 3^{\text{starCount}}$ respectively.

Let us take an expression $L * * R *$. So, the expression starts with a value of 1 which is the label of the root. When it encounters 'L', the value of sum becomes $2 \times 1 = 2$. When first '*' is encountered, $\text{starCount} = 1$, the value of sum is $5 \times 2 + 3^{\text{starCount} - 1} = 5 \times 2 + 3^0 = 11$. When it encounters second '*', $\text{starCount} = 2$, the value of sum is $5 \times 11 + 3^{\text{starCount} - 1} = 5 \times 11 + 3^1 = 58$. When 'R' is encountered the value of sum is $2 \times 58 + 3^{\text{starCount}} = 2 \times 58 + 3^2 = 125$. When it encounters last '*', $\text{starCount} = 3$, the value of sum is $5 \times 125 + 3^{\text{starCount} - 1} = 5 \times 125 + 3^2 = 634$.

The same expression is graphically depicted below.

Expr : L ** R *																							
L	<u>L</u>	<u>L</u>	R	<u>L</u>	=	34		L	<u>P</u>	<u>L</u>	R	<u>L</u>	=	18		L	<u>R</u>	<u>L</u>	R	<u>L</u>	=	42	
L	<u>L</u>	<u>L</u>	R	<u>P</u>	=	17		L	<u>P</u>	<u>L</u>	R	<u>P</u>	=	9		L	<u>R</u>	<u>L</u>	R	<u>P</u>	=	21	
L	<u>L</u>	<u>L</u>	R	<u>R</u>	=	35		L	<u>P</u>	<u>L</u>	R	<u>R</u>	=	19		L	<u>R</u>	<u>L</u>	R	<u>R</u>	=	43	
L	<u>L</u>	<u>P</u>	R	<u>L</u>	=	18		L	<u>P</u>	<u>P</u>	R	<u>L</u>	=	10		L	<u>R</u>	<u>P</u>	R	<u>L</u>	=	22	
L	<u>L</u>	<u>P</u>	R	<u>P</u>	=	9		L	<u>P</u>	<u>P</u>	R	<u>P</u>	=	5		L	<u>R</u>	<u>P</u>	R	<u>P</u>	=	11	
L	<u>L</u>	<u>P</u>	R	<u>R</u>	=	19		L	<u>P</u>	<u>P</u>	R	<u>R</u>	=	11		L	<u>R</u>	<u>P</u>	R	<u>R</u>	=	23	
L	<u>L</u>	<u>R</u>	R	<u>L</u>	=	38		L	<u>P</u>	<u>R</u>	R	<u>L</u>	=	22		L	<u>R</u>	<u>R</u>	R	<u>L</u>	=	46	
L	<u>L</u>	<u>R</u>	R	<u>P</u>	=	19		L	<u>P</u>	<u>R</u>	R	<u>P</u>	=	11		L	<u>R</u>	<u>R</u>	R	<u>P</u>	=	23	
L	<u>L</u>	<u>R</u>	R	<u>R</u>	=	39		L	<u>P</u>	<u>R</u>	R	<u>R</u>	=	23		L	<u>R</u>	<u>R</u>	R	<u>R</u>	=	47	
						228								128								278	
Sum = 228 + 128 + 278 = 634																							

Now, in order to store the large result of the sum we need to devise an array based data structure that will hold all the digits and perform data manipulation operations on that. However, In Java we already have the support of BigInteger, hence need not implement this data structure there.

Source Code (C++):

```
#include <string>
#include <vector>
#include <iostream>
#include <conio.h>
#define MODFACTOR 100000000
using namespace std;

// A class to represent very big integers that
// cannot be stored in the available datatypes
class BigInteger{
private:
    // represent the numbers as an array
    vector<int> nums;
    // store intermediate power of 3
    vector<int> powersOf3;

public:
    // constructor
    BigInteger(){
        nums.reserve(100);
        nums.push_back(1);
        powersOf3.reserve(100);
        powersOf3.push_back(1);
    }

    // multiply BigInteger by the given multiplier
    void multiply(int multiplier){
        for(int i = 0, len = nums.size(), carry = 0, value = 0; i < len; i++){
            value = multiplier * nums[i];
```

```

        nums[i] = (value + carry) % MODFACTOR;
        carry = (value + carry) / MODFACTOR;

        if(i == len - 1 && carry > 0){
            nums.push_back(carry);
        }
    }
}

// increase power of 3 by 1 everytime this method is called and store it
void preCalculatePowerOf3(){
    for(int i = 0, len = powersOf3.size(), carry = 0, value = 0; i < len; i++){
        value = 3 * powersOf3[i];

        powersOf3[i] = (value + carry) % MODFACTOR;
        carry = (value + carry) / MODFACTOR;

        if(i == len - 1 && carry > 0){
            powersOf3.push_back(carry);
        }
    }
}

// add the precalculated power of 3 to the current BigInteger
void addPowerOf3(){
    int numsCounter = nums.size();
    int powerOf3Counter = powersOf3.size();
    int maxsize = (numsCounter > powerOf3Counter) ? numsCounter : powerOf3Counter;

    int num1 = 0, num2 = 0;
    for(int walker = 0, carry = 0; walker < maxsize; walker++){
        if(walker < numsCounter)
            num1 = nums[walker];
        else
            num1 = 0;

        if(walker < powerOf3Counter)
            num2 = powersOf3[walker];
        else
            num2 = 0;

        int value = num1 + num2;
        int result = (value + carry) % MODFACTOR;
        carry = (value + carry) / MODFACTOR;

        if(walker < numsCounter)
            nums[walker] = result;
        else
            nums.push_back(result);

        if(walker == maxsize - 1 && carry > 0){
            if(walker < numsCounter)
                nums[walker] = carry;
            else
                nums.push_back(carry);
        }
    }
}

// string representation of the numbers
void display(){
    int len = nums.size();
    printf("%d", nums[len - 1]);
    for(int i = len - 2; i >= 0; i--){
        printf("%08d", nums[i]);
    }
}

};

class Setnja {
public:
    static void main() {
        // get the input data
        string inputSet;
    }
}

```

```

cin >> inputSet;

int len = inputSet.size();
// initialize totalWalkPath to 1
BigInteger totalWalkPath;

// As per the formula we have derived
// when a '*' is encountered the sum is 5x + 3^(starCount - 1), where x is the input value
// when a 'L' is encountered the sum is 2x, where x is the input value
// when a 'R' is encountered the sum is 2x + 3^starCount, where x is the input value
// when a 'P' is encountered the sum remains unchanged
for(int i = 0, starCount = 0; i < len; i++){
    switch(inputSet[i]){
        case '*':
            ++starCount;
            totalWalkPath.multiply(5);
            totalWalkPath.addPowerOf3();
            totalWalkPath.preCalculatePowerOf3();
            break;
        case 'L':
            totalWalkPath.multiply(2);
            break;
        case 'R':
            totalWalkPath.multiply(2);
            totalWalkPath.addPowerOf3();
            break;
        default:
            break;
    }
}
totalWalkPath.display();
printf("\n");
}
};

int main() {
    Setnja::main();
    getch();
    return 0;
}

```

Source Code (Java):

```

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.math.BigInteger;

public class Setnja {

    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        // get the input data
        String inputSet = br.readLine();
        int len = inputSet.length();

        // initialize totalWalkPath to 1
        BigInteger totalWalkPath = BigInteger.ONE;

        // As per the formula we have derived
        // when a '*' is encountered the sum is 5x + 3^(starCount - 1), where x is the input value
        // when a 'L' is encountered the sum is 2x, where x is the input value
        // when a 'R' is encountered the sum is 2x + 3^starCount, where x is the input value
        // when a 'P' is encountered the sum remains unchanged
        for(int i = 0, starCount = 0; i < len; i++){
            switch(inputSet.charAt(i)){
                case '*':
                    ++starCount;
                    totalWalkPath = totalWalkPath.multiply(new BigInteger("5"));
                    totalWalkPath = totalWalkPath.add(new BigInteger("3").pow(starCount - 1));
                    break;
                case 'L':
                    totalWalkPath = totalWalkPath.multiply(new BigInteger("2"));
                    break;
            }
        }
    }
}

```



```
        case 'R':
            totalWalkPath = totalWalkPath.multiply(new BigInteger("2"));
            totalWalkPath = totalWalkPath.add(new BigInteger("3").pow(starCount));
            break;
        default:
            break;
    }
}
System.out.println(totalWalkPath);
}
```

26. Harry Potter

Harry Potter has n mixtures in front of him, arranged in a row. Each mixture has one of 100 different colors (colors have numbers from 0 to 99). He wants to mix all these mixtures together. At each step, he is going to take two mixtures that stand next to each other and mix them together, and put the resulting mixture in their place. When mixing two mixtures of colors a and b , the resulting mixture will have the color $(a+b) \bmod 100$. Also, there will be some smoke in the process. The amount of smoke generated when mixing two mixtures of colors a and b is $a*b$.

Find out what is the minimum amount of smoke that Harry can get when mixing all the mixtures together.

Input

There will be a number of test cases in the input. The first line of each test case will contain n , the number of mixtures, $1 \leq n \leq 100$.

The second line will contain n integers between 0 and 99 - the initial colors of the mixtures.

Output

For each test case, output the minimum amount of smoke.

Example

Input:

```
2
18 19
3
40 60 20
```

Output:

```
342
2400
```

In the second test case, there are two possibilities:

first mix 40 and 60 (smoke: 2400), getting 0, then mix 0 and 20 (smoke: 0); total amount of smoke is 2400.

first mix 60 and 20 (smoke: 1200), getting 80, then mix 40 and 80 (smoke: 3200); total amount of smoke is 4400

The first scenario is a much better way to proceed.

Time limit: 1s

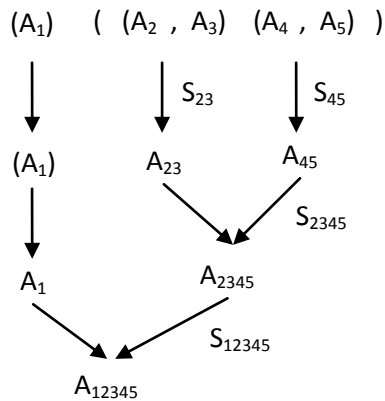
Source limit: 50000B

Solution:

This is a perfect case of Dynamic Programming problem. Dynamic programming is an algorithm in which an optimization problem is solved by saving the optimal scores for the solution of every subproblem instead of recalculating them. The main problem is to minimize the amount of smoke. The bigger problem of minimizing the smoke can be further sub-divided into smaller sub problems of minimizing smoke for the smaller combinations of colors.

Let us take an example of five colors,

A_1, A_2, A_3, A_4, A_5 . Now these colors can be grouped as $(A_1, A_2, A_3, A_4) (A_5)$ or $(A_1, A_2, A_3) (A_4, A_5)$ or $(A_1, A_2) (A_4, A_5)$ or $(A_1) (A_2, A_3, A_4, A_5)$. So, these combinations can be extended to any number of colors from A_1 to A_n . Actually for any given n , the total number of such combinations will be 2^{n-1} . The problem here asks here for which is the best possible combination of colors that we must make so that the total smoke generated (obtained by summing up smoke in each step) is minimum. So, our goal should be to minimize smoke in each such sub steps, so that the total smoke at the end is also minimum. This is also illustrated by the diagram below.



The smoke generated by the given combination is $S = S_{23} + S_{45} + S_{2345} + S_{12345}$.

We know, that smoke generated for mixing color a and b = $a * b$

Resultant color generated after mixing color a and b = $(a + b) \% 100$

Let, us take two arrays smoke[][] and mixedcolor[][] to denote the resultant smoke and color after mixing colors from A_i to A_j . Let, the colors are stored in color[].

Now, smoke[i][j] = 0 for all i = j, because a color cannot be mixed with itself.

Also, mixedColor[i][j] = color[i] for all i = j, because a color cannot be mixed with itself.

Now, let's say we have a sequence of colors, $A_i, A_{i+1}, A_{i+2}, \dots, A_k, \dots, A_j$

and we plan to subgroup at k, where $i \leq k < j$. So, the two groups now become $(A_i, A_{i+1}, A_{i+2}, \dots, A_k)$ $(A_{k+1}, A_{k+2}, \dots, A_j)$.

Hence, we can find the recursive relation,

$$smoke[i, j] = \begin{cases} 0, & \text{for } i = j \\ \min(smoke[i, k] + smoke[k + 1, j] + mixedcolor[i, k] * mixedcolor[k + 1, j]), & \text{for } i \leq k < j \end{cases}$$

$$mixedColor[i, j] = \begin{cases} color[i], & \text{for } i = j \\ (mixedColor[i, k] + mixedcolor[k + 1][j]) \% 100 & \text{for } i \leq k < j \end{cases}$$

Source Code(C++):

```

#include <iostream>
#define MAX 100
#define INF 100000000
using namespace std;
/*
 * smoke generated for mixing color a and b = a * b
 * color generated for mixing color a and b = (a + b) % 100
 * smoke[i][j] = min amount of smoke generated when color[i] to color[j] are mixed
 * mixedcolor[i][j] = resultant color produced when color[i] to color[j] are mixed
 * smoke[i][j] = 0 for all i = j
 * mixedcolor[i][j] = color[i] for all i = j
 * smoke[i][j] = min(smoke[i][k] + smoke[k + 1][j] + mixedcolor[i][k] * mixedcolor[k + 1][j]) for all i <= k < j
 * mixedcolor[i][j] = (mixedcolor[i][k] + mixedcolor[k + 1][j]) % 100
 */
class Mixtures {
public:
    static void main() {
        int smoke[MAX + 1][MAX + 1], mixedcolor[MAX + 1][MAX + 1], color[MAX + 1], n;

        while(scanf("%d", &n) != EOF){

            for(int i = 1; i <= n; i++){
                scanf("%d", &color[i]);
            }
            for(int i = 1; i <= MAX; i++){
                for(int j = 1; j <= MAX; j++){
                    smoke[i][j] = INF;
                }
            }
            for(int i = 1; i <= n; i++){
                smoke[i][i] = 0;
                mixedcolor[i][i] = color[i];
            }
            // we start with length = 2, because that's the least possible grouping
            for(int length = 2; length <= n; length++){
                for(int i = 1; i <= n - length + 1; i++){
                    int j = i + length - 1;
                    for(int k = i; k < j; k++){
                        // find the minimum smoke
                        int temp = smoke[i][k] + smoke[k + 1][j] + mixedcolor[i][k] * mixedcolor[k + 1][j];
                        if(smoke[i][j] > temp){
                            smoke[i][j] = temp;
                        }
                    }
                }
            }
        }
    }
};

```

```

        mixedcolor[i][j] = (mixedcolor[i][k] + mixedcolor[k + 1][j]) % MAX;
    }
}
}
}
printf("%d\n", smoke[1][n]);
}
}
};

int main() {
    Mixtures::main();
    return 0;
}

```

Source Code(Java):

```

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.StringTokenizer;
/*
 * smoke generated for mixing color a and b = a * b
 * color generated for mixing color a and b = (a + b) % 100
 * smoke[i][j] = min amount of smoke generated when color[i] to color[j] are mixed
 * mixedcolor[i][j] = resultant color produced when color[i] to color[j] are mixed
 * smoke[i][j] = 0 for all i = j
 * mixedcolor[i][j] = color[i] for all i = j
 * smoke[i][j] = min(smoke[i][k] + smoke[k + 1][j] + mixedcolor[i][k] * mixedcolor[k + 1][j]) for all i <= k < j
 * mixedcolor[i][j] = (mixedcolor[i][k] + mixedcolor[k + 1][j]) % 100
 */
public class Mixtures {

    public static final int MAX = 100;
    public static final int INF = 100000000;

    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        int smoke[][] = new int[MAX + 1][MAX + 1];
        int mixedcolor[][] = new int[MAX + 1][MAX + 1], color[] = new int[MAX + 1];

        String in = null;
        while ((in = br.readLine()) != null) {
            int n = Integer.parseInt(in);

            StringTokenizer st = new StringTokenizer(br.readLine(), " ");
            for (int i = 1; i <= n; i++) {
                color[i] = Integer.parseInt(st.nextToken());
            }

            // initialize smoke and mixedcolor
            for (int i = 1; i <= MAX; i++) {
                for (int j = 1; j <= MAX; j++) {
                    smoke[i][j] = INF;
                }
            }
            for (int i = 1; i <= n; i++) {
                smoke[i][i] = 0;
                mixedcolor[i][i] = color[i];
            }

            // we start with length = 2, because that's the least possible grouping
            for (int length = 2; length <= n; length++) {
                for (int i = 1; i <= n - length + 1; i++) {
                    int j = i + length - 1;
                    for (int k = i; k < j; k++) {
                        // find the minimum smoke
                        int temp = smoke[i][k] + smoke[k + 1][j] + mixedcolor[i][k] * mixedcolor[k + 1][j];
                        if (smoke[i][j] > temp) {
                            smoke[i][j] = temp;
                            mixedcolor[i][j] = (mixedcolor[i][k] + mixedcolor[k + 1][j]) % MAX;
                        }
                    }
                }
            }
        }
    }
}

```

```
        }  
        System.out.println(smoke[1][n]);  
    }  
}
```

27. Lisa

Young people spend a lot of money on things like sweets, music CDs, mobile phones and so on. But most young girls/boys have one problem: Their pocket money is not enough for all these jolly things. Little Lisa Listig is one of these poor girls with a small pocket money budget. Last month her pocket money lasted only one week. So she decided to enter into negotiations with her father. Her father Tomm - a mathematician - had an incredibly ingenious idea: He wrote down some fancy digits with operators (+,*) in between them on a sheet of paper and allowed Lisa to insert brackets. Then he declared that the result of that arithmetic expression is Lisa's new pocket money. Now it's Lisa's task to maximize her pocket money. As her father was surprised what a huge sum of money Lisa got for her result, he decided to minimize the result of the expression for his son Manfred. Now it's your task to calculate the results obtained by Lisa and her father.

Input

The first line of input contains the number of testcases k ($k < 5000$). Each of the following k lines consists of an arithmetic expression. This expression consists of numbers (0-9) separated by one of the two operators '*' and '+'. There are no spaces between the characters. Each line contains less than 100 characters.

Output

For each expression output the result obtained by Lisa and the result obtained by her father separated by one space. The results of the calculations are smaller than 2^{64} .

Example

Input:

```
1
1+2*3+4*5
```

Output:

```
105 27
```

Two possible expressions for the first testcase:

$$105 = (1+2)*(3+4)*5$$

$$27 = 1+2*3+4*5$$

Time limit: 32s

Source limit: 8082B

Solution:

	1	2	3	4	5	6	7	8	9
1	1		3		9		21		105
2									
3			2		6		14		70
4									
5					3		7		35
6									
7							4		20
8									
9									5

This is another classic case of 'Dynamic Programming'. The objective here is to minimize and maximize the expression. We maintain here an array $value[MAX + 1][MAX + 1][2]$ to store the maximum and minimum value of the expression for different location of brackets, $value[MAX + 1][MAX + 1][0]$ is for storing the maximum values and $value[MAX + 1][MAX + 1][1]$ is for storing the minimum values. On the left we show the $value[MAX + 1][MAX + 1][0]$ for the expression $1+2*3+4*5$.

To find the initial state variables we observe that when a bracket is wrapped around a single integer, then the resultant value is the integer itself, i.e. no change, Hence for

Length = 1

$value[1, 1] = 1$, $value[3, 3] = 2$, $value[5, 5] = 5$, $value[7, 7] = 7$, $value[9, 9] = 9$. Note: only the odd numbered rows and columns

have digits while all the even numbered rows and columns are empty because the expression contains '*' and '+' in between the digits.

Now, we have to break the bigger problem into sub problems by taking 2, 3, 4 and 5 digits at a time and place the brackets.

Length = 2

Lets say if we had taken two inetegers at a time and tried to put brackets, the sub expressions would have been, (1+2), (2*3), (3+4), (4*5). Now, (1+2) can be further sub divided into (1) + (2), which we can get from the above step. Similarly for the other cases. So, taking all the combinations, we have

value[1, 3] = value[1, 1] + value[3, 3] = 3
value[3, 5] = value[3, 3] x value[5, 5] = 6
value[5, 7] = value[5, 5] + value[7, 7] = 7
value[7, 9] = value[7, 7] x value[9, 9] = 20

Length = 3

Lets say if we had taken three inetegers at a time and tried to put brackets, the sub expressions would have been, (1+2*3), (2*3+4), (3+4*5). Now, (1+2*3) can be further sub divided into (1+2) * (3) or (1) + (2*3), which we can get from the above steps. So the maximum value for (1+2*3) at this step should be the maximum of all the possible sub combinations. Similarly for the other cases. So, taking all the combinations, we have

value[1, 5] = max(value[1, 1] + value[3, 5], value[1, 3] x value[5, 5]) = max(7, 9) = 9
value[3, 7] = max(value[3, 3] x value[5, 7], value[3, 5] + value[7, 7]) = max(14, 10) = 14
value[5, 9] = max(value[5, 5] + value[7, 9], value[5, 7] x value[9, 9]) = max(23, 35) = 35

Length = 4

Similar logic can be extended when we take 4 integers at a time. So, taking all the combinations, we have

value[1, 7] = max(value[1, 1] + value[3, 7], value[1, 3] x value[5, 7], value[1, 5] + value[7, 7]) = max(15, 21, 13) = 21
value[3, 9] = max(value[3, 3] x value[5, 9], value[3, 5] + value[7, 9], value[3, 7] x value[9, 9]) = max(70, 26, 70) = 70

Length = 5

Similar logic can be extended when we take 5 integers at a time. So, taking all the combinations, we have

value[1, 9] = max(value[1, 1] + value[3, 9], value[1, 3] x value[5, 9], value[1, 5] + value[7, 9], value[1, 7] x value[9, 9])
= max(71, 105, 29, 105) = 105

Hence, in general, we can form a recursive relation as:

$$value[i, j] = \begin{cases} value[i], & \text{for all } i = j \\ value[i, i] + value[j, j], & \text{for all } j = i + 2 \\ \max(value[i, k] \text{ operand } [k + 1] \text{ value}[k + 1][j]) & \text{for all } k, \text{ where } i \leq k < j \end{cases}$$

Source Code(C++):

```
#include <iostream>
#include <string.h>
#include <limits.h>
#define MAX 100
using namespace std;

class Lisa {
private:
    static long long int max(long long int a, long long int b){
        return a > b ? a : b;
    }
    static long long int min(long long int a, long long int b){
        return a < b ? a : b;
    }
    static long partialValue(long long int a, long long int b, char op){
        long partialValue = 0;

        switch(op){
            case '*':
                partialValue = a * b;
                break;
            case '+':
                partialValue = a + b;
                break;
        }

        return partialValue;
    }
public:
    static void main() {
        int T;

        for(scanf("%d",&T); T; --T){

            char input[MAX];
            scanf("%s",input);
            int N = strlen(input);

            // store the value of the expressio with brackets the array has size 2 for the last dimension
            // we use value[i][j][0] for max value and value[i][j][1] for min value
            long long int value[MAX + 1][MAX + 1][2];

            for(int i = 1; i <= N; i+=2){
                for(int j = 1; j <= N; j+=2){
                    value[i][j][0] = 0;
                    value[i][j][1] = LONG_MAX;
                }
            }

            for(int i = 1; i <= N; i+=2){
                value[i][i][0] = value[i][i][1] = input[i - 1] - '0';
            }

            // recursively calculate the expression value with different bracket combinations
            for(int len = 2, L = N / 2 + 1; len <= L; len++){
                for(int i = 1; i <= N - 2 * (len - 1); i+=2){
                    int j = i + 2 * (len - 1);
                    for(int k = i; k < j; k+=2){
                        value[i][j][0] = max(value[i][j][0], partialValue(value[i][k][0], value[k+2][j][0], input[k]));
                        value[i][j][1] = min(value[i][j][1], partialValue(value[i][k][1], value[k+2][j][1], input[k]));
                    }
                }
            }
            printf("%lld %lld\n", value[1][N][0], value[1][N][1]);
        }
    }
};

int main() {
    Lisa::main();
    return 0;
}
```



```
}
```

Source Code(Java):

```
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class Lisa {
    private static long max(long a, long b){
        return a > b ? a : b;
    }
    private static long min(long a, long b){
        return a < b ? a : b;
    }
    private static long partialValue(long a, long b, char op){
        long partialValue = 0;

        switch(op){
            case '*':
                partialValue = a * b;
                break;
            case '+':
                partialValue = a + b;
                break;
        }

        return partialValue;
    }
    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        int T = Integer.parseInt(br.readLine());

        while(T-- > 0){

            char input[] = br.readLine().toCharArray();
            int N = input.length;

            // store the value of the expressio with brackets the array has size 2 for the last dimension
            // we use value[i][j][0] for max value and value[i][j][1] for min value
            long value[][][] = new long[N + 1][N + 1][2];

            for(int i = 1; i <= N; i+=2){
                for(int j = 1; j <= N; j+=2){
                    value[i][j][0] = 0;
                    value[i][j][1] = Long.MAX_VALUE;
                }
            }

            for(int i = 1; i <= N; i+=2){
                value[i][i][0] = value[i][i][1] = Long.parseLong(Character.toString(input[i - 1]));
            }

            // recursively calculate the expression value with different bracket combinations
            for(int len = 2, L = N / 2 + 1; len <= L; len++){
                for(int i = 1; i <= N - 2 * (len - 1); i+=2){
                    int j = i + 2 * (len - 1);
                    for(int k = i; k < j; k+=2){
                        value[i][j][0] = max(value[i][j][0], partialValue(value[i][k][0], value[k+2][j][0], input[k]));
                        value[i][j][1] = min(value[i][j][1], partialValue(value[i][k][1], value[k+2][j][1], input[k]));
                    }
                }
            }
            System.out.println(value[1][N][0] + " " + value[1][N][1]);
        }
    }
}
```

29. Temptation Island

On Monday, the number of frosh were reduced in half. To further reduce the number of engineers to a manageable number, the following challenge was devised for the second day. Each of the students would have to take this challenge individually.

Each student would be placed at a vertex of perimeter fence of Waterloo (oh yeah, some background: to keep UofT's engineering Lady Godiva band out of Waterloo, a fence was erected surrounding the university. The fence just happens to be an N-gon). At some other vertex along the fence would be located a temptation so seductive that no Waterloo student could resist - an extra-credit assignment.

The challenge of each student is to go from his starting vertex to the vertex with the prize. There are however 3 rules:

- The student can only travel from vertex to vertex (backwards or forwards) along the polygonal fence.
- The student has to make contact with exactly K vertices (the vertex he starts at doesn't count unless he returns to it). The K vertices need not be unique. The final vertex has to be the one with the prize.
- If the student cannot reach the prize and make contact with exactly K vertices, he fails the test and is kicked out of the university.

Of course, no Waterloo student is satisfied with only 1 solution to any problem. Therefore, inevitably, each student determines all ways that he/she can win. Note that there may be no solution to the problem (the astute student has figured out that this will result in a class size of 0 - this is entirely allowable as the variable used to quantify enrollment was incorrectly defined as a whole number instead of a natural number).

Input

N K (N, K <= 50)
A B (A = the starting vertex number, B = destination vertex number)
-1 -1 terminates input

Output

The total number of ways of reaching the destination from the starting point by following the above rules. The total number of ways will be less than $2^{63} - 1$. Output 0 if there are no solution.

Example

Input:

```
8 5
1 4
3 2
1 1
3 2
1 2
1 1
1 1
50 25
10 49
50 1
1 49
50 49
25 26
-1 -1
```

Output:

6
2
1
0
480700
0
63205303218877

Time limit: 0.5s-1s

Source limit: 50000B

Solution:

This problem can be solved using recursive as well as Dynamic programming approach. Let us take an example as

3 2
2 2

So, $N = 3$, $K = 2$ and $A = 2$, $B = 2$. So, we have a triangle here, and the prize is at vertex 2 which is also the starting point. But, from starting point to end point (which are same in this case, we must visit 2 vertices in total)

Dynamic Programming

		Vertex(v)		
		1	2	3
k	0	1	②	1
	1	1	0	1
	2	0	0	0

We create a possibility matrix as shown in the left and traverse backwards from destination to start.

When $k = 2$, there is no possibility of reaching vertex 2, because we have already reached there. So, all cells are filled with 0.

When $k = 1$, we can reach vertex 2 from either vertex 1 or 3. So we have 1 route from each of 1 and 3. So, $M[1, 1]$ and $M[1, 3]$ are populated with 1.

The rest of the table can be filled up dynamically.

When $k = 0$, we can reach vertex 4 of $k = 2$ by two paths. $v=2, k=0 \rightarrow v=1, k=1 \rightarrow v=4, k=2$ or $v=2, k=0 \rightarrow v=3, k=1 \rightarrow v=4, k=2$. So, essentially it turns out that, total possibility at a cell is sum of the possibilities of cells diagonally below it. This gives the required DP recursive algorithm. The final result should be stored in $M[0, startVertex]$.

Now if the startVertex on the left or rightmost column, to go diagonally to the next row one must shift to the other side of the matrix, e.g. diagonally right cell of $M[0, 3]$ is $M[1, 1]$ and diagonally left cell of $M[0, 1]$ is $M[1, 3]$.

So, the dp relation can be written as

$$dp[i][j] = dp[i + 1][(j - 1) \% N] + dp[i + 1][(j + 1) \% N]$$

Recursive Programming

2 k = 0	1 k = 1	2 k = 2
		3 k = 2
	3 k = 1	1 k = 2
		2 k = 2

Let us start from stage $k = 0$. Since the start vertex is 2, there are two possibilities from there, we can go to vertex 1 or vertex 3 in stage $k = 1$. At stage, $k = 1$, we again have two possibilities for each of the stages. Hence, the total number of possible paths at the end of stage $k = 2$ is 4. However, we see that not all possible 4 paths lead us to vertex 2 at end of search i.e. $k = 2$. Only, two of them does.

Recursively calculate the path starting from one index and ending at the other. If we can reach the end vertex at the given K , then we can say that a path is possible. However, if we have not reached the end vertex at the given K then we say there is no path possible. In the intermediate turns, k where $k < K$, we again

need to calculate recursively for previous and next vertex from the current vertex for the next turn, because each vertex has two possibilities.

Source Code(C++):

```
#include <iostream>
using namespace std;

class TemptationIsland {
public:
    static void main() {
        while (true) {
            int N, K;
            scanf("%d %d", &N, &K);

            if (N == -1 && K == -1)
                break;

            int A, B;
            scanf("%d %d", &A, &B);

            // special case, if start and end vertex are not same
            if ((A == B) && (K == 0 || K == 1)){
                printf("0\n");
                continue;
            }

            // initialize the possibility dynamic table
            long long int **possibility = new long long int*[K + 1];
            for(int k = 0; k <= K; k++){
                possibility[k] = new long long int[N + 1];
                for(int v = 0; v <= N; v++){
                    possibility[k][v] = 0;
                }
            }

            // initilize the base cases. One can reach the destination if and only if one is
            // either 1 vertex before or after the destination vertex in the previous turn
            possibility[K - 1][B == 1 ? N : B - 1] = 1;
            possibility[K - 1][B == N ? 1 : B + 1] = 1;

            // fill up the table dynamically
            // dp relation is dp[i][j] = dp[i + 1][(j - 1) % N] + dp[i + 1][(j + 1) % N]
            for(int k = K - 2; k >= 0; k--){
                for(int v = 1; v <= N; v++){
                    possibility[k][v] =
                        possibility[k + 1][v == 1 ? N : v - 1] +
                        possibility[k + 1][v == N ? 1 : v + 1];
                }
            }
            printf("%lld\n", possibility[0][A]);
        }
    }
};

int main() {
    TemptationIsland::main();
    return 0;
}
```

Source Code(java):

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.StringTokenizer;

public class TemptationIsland {

    /* Recursively calculate the path starting from one index and ending at the other.
     * If we can reach the end vertex at the given K, then we can say that a path is possible.
     * However, if we have not reached the end vertex at the given K then we say there
     * is no path possible. In the intermediate turns, k where k < K, we again need to calculate
     * recursively for previous and next vertex from the current vertex for the next turn,
     * because each vertex has two possibilities.
     */
}
```

```
private static int possibility(int start, int end, int N, int k, int K){
    if(k == K){
        return start == end ? 1 : 0;
    }else{
        return possibility(((start - 1) + N) % N,end,N,k + 1,K) + possibility((start + 1) % N,end,N,k + 1,K);
    }
}
public static void main(String[] args) throws Exception {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

    while (true) {
        StringTokenizer st = new StringTokenizer(br.readLine(), " ");
        int N = Integer.parseInt(st.nextToken());
        int K = Integer.parseInt(st.nextToken());

        if (N == -1 && K == -1)
            break;

        st = new StringTokenizer(br.readLine(), " ");
        int A = Integer.parseInt(st.nextToken());
        int B = Integer.parseInt(st.nextToken());

        // special case, if start and end vertex are not same
        if((A == B) && (K == 0 || K == 1)){
            System.out.println("0");
            continue;
        }

        System.out.println(possibility(A, B, N, 0, K));
    }
}
```

30. Black Or White

You have a sequence of integers. You can paint each of the integers black or white, or leave it unpainted. The black integers must appear in ascending order and the white integers must appear in descending order. The ascending/descending order must be strict, that is, two integers painted with the same color cannot be equal. Paint the sequence so as to minimize the number of unpainted integers.

Input

The input contains several test cases, each one described in exactly two lines. The first line contains an integer N indicating the number of elements in the sequence ($1 \leq N \leq 200$). The second line contains N integers X_i separated by single spaces, representing the sequence to paint ($1 \leq X_i \leq 10^6$ for $1 \leq i \leq N$). The last line of the input contains a single -1 and should not be processed as a test case.

Output

For each test case output a single line with an integer representing the minimum number of unpainted elements of the sequence, when the sequence is painted optimally following the rules described above.

Example

Input:

```
1
1
2
6 7
3
1 2 4
8
1 4 2 3 3 2 4 1
12
7 8 1 2 4 6 3 5 2 1 8 7
3
1 3 2
5
5 1 6 2 1
7
7 8 1 4 1 8 7
32
7 8 11 2 4 6 31 5 2 1 8 7 9 8 6 7 5 2 3 4 8 7 2 7 9 87 45 23 7 89 3 3
-1
```

Output:

```
0
0
0
0
2
0
0
2
17
```

Time limit: 120s
Source limit: 50000B

Solution:

This is a hard 'Dynamic Programing' problem. As usual this problem can be solved by two approaches, (i) Bottom Up Dynamic Programing or (ii) Recursive Approach

Let us take an example number set:

5 1 6 2 1

The table below shows all the possible path combinations for the above set, and the best path is highlighted in red.

5	1	6	2	1
w	b	b	x	x
				w
		w	x	x
				w
		x	b	x
				w
	w	x	w	x
				w
		b	x	x
				x
		x	b	x
				x

5	1	6	2	1
b	x	b	x	x
				w
		w	x	x
				w
		w	x	x
				w
	w	b	x	x
				x
		x	x	x
				x
		x	x	x
				x

So, if we observe carefully, we can infer that, if there are 'n' numbers, then the total number of paths will be 2^n . For, the given problem, the maximum number of numbers can be 200, hence there will be 2^{200} paths, which is a HUGE number. So, a recursive solution is not the right choice for large number of 'n'. For completeness, we have also included the recursive approach in the Java solution.

To solve this we have to take help of 'Dynamic Programing' and that can be bottom up. We have used the state $dp[prev_b][prev_w][curIdx]$. $dp[prev_b][prev_w][curIdx]$ is the maximum number of painted numbers if last black index is prev_b, last white index is prev_w and the index in question is curIdx. We can move from one state ($dp[prev_b][prev_w][curIdx]$) to other ($dp[prev_b][prev_w][k]$) by iterating over the numbers, and the state k can be painted black if $value[k] > value[prev_b]$ or the state k can be painted white if $value[k] < value[prev_w]$. It might also happen that the number at state k does not satisfy any of the before mentioned cases and cannot be painted. We, take a bottom up approach here because we do not know the maximum number of numbers that are painted in the current state without knowing the maximum number of numbers that can be painted in the next/child state. Again for the child/next state it is not possible to determine the number of numbers that can be painted without knowing the index of previous black and previous white from the parent state. It is a cyclical dependency. So, at each state we have to take maximum of (current state cannot be painted; hence 0 and add the number of numbers that can be painted in next/child state, current state can be painted black; hence 1 and add the number of numbers that can be painted in next/child state, current state can be painted white; hence 1 and add the number of numbers that can be painted in next/child state).

Hence the recursive relation can be stated as:

```

painted = max(dp[prev_b][prev_w][curIdx], findMaxPainted(prev_b, prev_w, curIdx + 1);

if(prev_b == 0 || (sequence[prev_b] < sequence[curIdx])){
    painted = max(painted, 1 + findMaxPainted(curIdx, prev_w, curIdx + 1));
}
if(prev_w == 0 || (sequence[prev_w] > (sequence[curIdx])){
    painted = max(painted, 1 + findMaxPainted(prev_b, curIdx, curIdx + 1));
}

```

Source Code(C++):

```

#include <iostream>
#include <cstring>
#define MAX 200
using namespace std;

int dp[MAX + 1][MAX + 1][MAX + 1];

class BlackOrWhite {
private:
    /*
     * Bottom up recursive dynamic programming approach to find the maximum painted numbers
     * for the curIdx, given the previous black and white positions.
     */
    static int findMaxPainted(int seq[], int prev_b, int prev_w, int curIdx, int N)
    {
        if(dp[prev_b][prev_w][curIdx] != -1){
            return dp[prev_b][prev_w][curIdx];
        }

        // this is the base case at the last index
        if(curIdx == N){
            dp[prev_b][prev_w][curIdx] = 0; // in the worst case, we cannot paint the number

            // check if we can paint it black
            if(prev_b == 0 || seq[prev_b] < seq[curIdx]){
                dp[prev_b][prev_w][curIdx] = max(dp[prev_b][prev_w][curIdx], 1);
            }

            // check if we can paint it white
            if(prev_w == 0 || seq[prev_w] > seq[curIdx]){
                dp[prev_b][prev_w][curIdx] = max(dp[prev_b][prev_w][curIdx], 1);
            }

            return dp[prev_b][prev_w][curIdx];
        }

        //check the next step first to be safe, whether we get a better result
        // if we do not paint this number
        int ret = max(dp[prev_b][prev_w][curIdx], findMaxPainted(seq, prev_b, prev_w, curIdx + 1, N));

        // check if we can paint it black
        if(prev_b == 0 || seq[prev_b] < seq[curIdx]){
            ret = max(ret, 1 + findMaxPainted(seq, curIdx, prev_w, curIdx + 1, N));
        }
        // check if we can paint it white
        if(prev_w == 0 || seq[prev_w] > seq[curIdx]){
            ret = max(ret, 1 + findMaxPainted(seq, prev_b, curIdx, curIdx + 1, N));
        }

        return dp[prev_b][prev_w][curIdx] = ret;
    }
public:
    static void main() {

        int N;

        while (true) {
            scanf("%d", &N);

```



```

        if(N == -1)
            break;

        // get the numbers
        int seq[MAX + 1];
        for(int i = 1; i <= N; i++)
            scanf("%d",&seq[i]);

        memset(dp, -1, sizeof dp);

        // if N = 1 or N = 2, then all numbers can be coloured, irrespective of sequence
        if(N < 3) {
            printf("0\n");
        }else{
            printf("%d\n", N - findMaxPainted(seq, 0, 0, 1, N));
        }
    }
};

int main() {
    BlackOrWhite::main();
    return 0;
}

```

Further Optimizations:

We can avoid memorizing current index if we follow a bottom up approach. This idea works if in practice we see that not memoising a variable is not affecting the dp calculation. That is not true for all solutions though. If we observe carefully then we can infer that a pair of black and white for a parent call are never calculated without calling the same for the child. In this case, the curIdx always decreases by one and no two curIdx will be same ever, hence the dp matrix will always have the state matrix for previously calculated child i.e. curIdx + 1. So, a 2D dp matrix is enough, however if we maintain a 3D dp matrix with the last dimension as curIdx then the third dimension will always be used sparsely and no cell of curIdx and curIdx + 1 will ever overlap, it will be a waste of space and time too. Also instead of a recursive dp we can use iterative dp to make it fast.

```

#include <iostream>
#include <cstring>
#define MAX 200
using namespace std;

int dp[MAX + 1][MAX + 1];

class BlackOrWhiteIterativeDP {
public:
    static void main() {

        int N;

        while (true) {
            scanf("%d",&N);

            if(N == -1)
                break;

            int seq[MAX + 1];
            for(int i = 1; i <= N; i++)
                scanf("%d",&seq[i]);

            memset(dp, 0, sizeof dp);

            // if N = 1 or N = 2, then all numbers can be coloured, irrespective of sequence
            if(N < 3) {
                printf("0\n");
            }else{
                /*
                 * Bottom up iterative dynamic programming approach to find the maximum painted numbers
                 * for the curIdx, given the previous black and white positions.
                 */
                for(int curIdx = N; curIdx >= 1; curIdx--){
                    for(int prev_b = 0; prev_b < curIdx; prev_b++){

```

```

        for(int prev_w = 0; prev_w < curIdx; prev_w++){
            if(prev_w == prev_b && prev_w != 0 && prev_b != 0 ){
                continue;
            }

            // this is the base case at the last index
            if(curIdx == N){

                // check if we can paint it black
                if(prev_b == 0 || seq[prev_b] < seq[curIdx]){
                    dp[prev_b][prev_w] = 1;
                }

                // check if we can paint it white
                if(prev_w == 0 || seq[prev_w] > seq[curIdx]){
                    dp[prev_b][prev_w] = 1;
                }
            }
            else{
                //check the next step first to be safe, whether we get a better result
                // if we do not paint this number
                int ret = dp[prev_b][prev_w];

                // check if we can paint it black
                if(prev_b == 0 || seq[prev_b] < seq[curIdx]){
                    ret = max(ret, 1 + dp[curIdx][prev_w]);
                }
                // check if we can paint it white
                if(prev_w == 0 || seq[prev_w] > seq[curIdx]){
                    ret = max(ret, 1 + dp[prev_b][curIdx]);
                }
                dp[prev_b][prev_w] = ret;
            }
        }
    }
}

printf("%d\n", N - dp[0][0]);
}
}
}
};

int main() {
    BlackOrWhiteIterativeDP::main();
    return 0;
}

```

Source Code(Java):

```

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.StringTokenizer;

public class BlackOrWhite {
    /*
     * Calculate iteratively for each index/stage the maximum number of number that can be
     * painted. For each stage there can be two ways to go to the next stage and the
     * number in the present stage can either be coloured or not coloured based on the
     * previous index of black or white painted numbers.
     */
    private static int findMaxPainted(int seq[], int prev_b, int prev_w, int curIdx, int N){
        int rb = 0, lb = 0;    // lb = left branch, rb = right branch
        // find whether we can assign 'B' or 'W' to the current Index
        if(prev_w == -1 || seq[prev_w] > seq[curIdx]){
            lb++;
        }
        if(prev_b == -1 || seq[prev_b] < seq[curIdx]){
            rb++;
        }

        // When we are at the last stage, the return should be max value of the two branches
        if(curIdx == N - 1){
            return Math.max(lb, rb);
        }
        else {

```

```

        // for each given index the next index can be filled up in max two ways
        // either 'B' or 'W'. So, there can be two paths to the next stage. Calculate the path
        // lengths for each of the ways and take the max of them
        return Math.max(lb + findMaxPainted(seq, prev_b, lb == 1? curIdx : prev_w, curIdx + 1, N),
            rb + findMaxPainted(seq, rb == 1? curIdx : prev_b, prev_w, curIdx + 1, N));
    }
}

public static void main(String[] args) throws Exception {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

    int N;

    while((N = Integer.parseInt(br.readLine())) != -1){

        int seq[] = new int[N];
        StringTokenizer st = new StringTokenizer(br.readLine(), " ");
        for(int i = 0; i < N; i++){
            seq[i] = Integer.parseInt(st.nextToken());
        }

        // if N = 1 or N = 2, then all numbers can be coloured, irrespective of sequence
        if(N < 3) {
            System.out.println(0);
        }else{

            int painted = 0;
            // for each index find the maximum possible number of integers that can be coloured
            // and then take the max of them
            for(int i = 0; i < N - 1 && painted != N - i; i++){
                painted = Math.max(painted, findMaxPainted(seq, -1, -1, i, N));
            }

            System.out.println(N - painted);
        }
    }
}
}
}

```

31. Round Table

We have a round table with $2*N$ seats ($1 \leq N \leq 2000$). The seats are numbered with the numbers from 1 to $2*N$ in order. This round table is in a round room with two doors. Door 1 is between seats 1 and $2N$ on the table, and door 2 is between seats N and $N+1$. We have $2N$ guests that we want to seat. Each guest has a id from 1 to $2N$ and we want to seat him/her exactly at the seat corresponding to his id. All the guest are split into two groups of N and each group is waiting in front of a door in a line. This room unfortunately has the problem that if you let a person go from door 1 and he/she has to seep at position p , to get to his seat he has to make either all people seating on seats from 1 to $p-1$ or all people seating on seats $p+1$ to $2N$ to get up to let him in. This presents you with the problem of how to let the guests in to cause minimum number of getting ups. The only thing you control is from which door to let the next guest. The guests that are waiting in front of door 1 can only get in from door 1 the same is true with door 2.

Input

On the first line there will one integer - N . Two more lines of input follow each with N integers in the range $[1, 2N]$ - the guests waiting in front of door 1 and door 2 respectively. The first people in the list is the first to enter the room.

Output

Single integer the total min number of stand-ups that will happen if we choose the best sequence of letting guest in.

Example

Input:

3
4 5 3
6 2 1

Output:

3

Input:

5
4 7 1 6 9
3 8 5 10 2

Output:

9

Input:

8
2 4 6 8 10 12 14 16
1 3 5 7 9 11 13 15

Output:

12

Time limit: 120s

Source limit: 50000B

Solution:

This is a hard 'Dynamic Programing' problem. Here we have two varying parameters, people in team 1 as well as people in team 2. Moreover, all the people can either sit clockwise or anticlockwise. So, we have 4 different cases. So, consider the 4 different cases separately and then find the minimum of all. Let us take the given test case

3 No of members in each team
4 5 3 Team 1
6 2 1 Team 2

Let us consider that people from first team are now moving in and they are going into their seats in either clockwise or anticlockwise direction. Now, before the i^{th} person from first team moves in, it is sure that $i - 1$ persons from first team have already bean seated (because we can let people from any door only sequentially) and it is possible that j people from second team have already bean seated. Depending on the clockwise or anticlockwise direction, the number of stand ups can be determined by merely comparing two values $\text{firstTeam}[i - 1]$ and $\text{firstTeam}[i]$ and $\text{secondTeam}[j]$ and $\text{firstTeam}[i]$. Similarly this can be applied for second team also when we consider that people from second team are now moving in and they are going into their seats int either clockwise or anticlockwise direction. Now, before the i^{th} person from second team moves in, $i - 1$ persons from second team have already bean seated (because we can let people from any door only sequentially) and it is possible that j

people from first team have already been seated. So we have used two arrays `standUpWhenXVaries[i][j][k]` and `standUpWhenYVaries[i][j][k]`.

`standUpWhenXVaries[i][j][0]` means the number of times to stand up when the first $i - 1$ people in the first team and the first j people in the second team have been seated, and the i^{th} person from the first team is moving clockwise into his seat. `standUpWhenXVaries[i][j][1]` is the same but anticlockwise. $K = 0$ signifies clockwise and $k = 1$ signifies anticlockwise. Similarly we have `standUpWhenYVaries[i][j][k]` for second team.

standUpWhenXVaries (Clockwise)					standUpWhenXVaries (AntiClockwise)					standUpWhenYVaries (Clockwise)					standUpWhenYVaries (AntiClockwise)				
0 1 2 3					0 1 2 3					0 1 2 3					0 1 2 3				
0	0	0	0	0	0	0	0	0	0	0	0	1	1		0	0	0	1	
1	0	0	1	2	1	0	1	1	1	1	0	1	2	2	1	0	0	0	1
2	1	1	2	3	2	0	1	1	1	2	0	2	3	3	2	0	0	0	1
3	0	0	1	2	3	2	3	3	3	3	0	2	3	3	3	0	1	1	2

Once we are able to find the stand ups for all the combinations, now we calculate the minimum stand up required for all the combinations and use array `minStanding[i][j]` to denote the minimum number of times to stand up when the first i people in the first team and the first j people in the second team have been seated.

Minimum Standup

0 1 2 3				
0	0	0	0	1
1	0	0	0	1
2	0	0	0	1
3	0	0	1	3

Source Code(C++):

```
#include <iostream>
#define MAX 2001
using namespace std;

short int min(short int a, short int b){
    return a < b ? a : b;
}
// store the first team and second team
short int x[MAX], y[MAX];
// standUpWhenXVaries[i][j][0] means the number of times to stand up when the first i-1 people
// in the first team and the first j people in the second team have been seated, and the ith person
// from the first team is moving clockwise into his seat.
// standUpWhenXVaries[i][j][1] is the same but anticlockwise
short int standUpWhenXVaries[MAX][MAX][2];
// standUpWhenYVaries[i][j][0] means the number of times to stand up when the first i-1 people
// in the second team and the first j people in the first team have been seated, and the ith person
// from the second team is moving clockwise into his seat.
// standUpWhenYVaries[i][j][1] is the same but anticlockwise
short int standUpWhenYVaries[MAX][MAX][2];
// minStanding[i][j] means the minimum number of times to stand up when the first i people in the
// first team and the first j people in the second team have been seated.
short int minStanding[MAX][MAX];
```

```
class Roundt{
public:
    static void main(){

        // get the inputs
        int N;
        scanf("%d",&N);

        for(int i = 1; i <= N; i++){
            scanf("%d",&x[i]);
        }

        for(int i = 1; i <= N; i++){
            scanf("%d",&y[i]);
        }

        // calculate the minimum number of times to stand up when the first i-1 people in the first team
        // and the first j people in the second team have been seated, and the ith person from the first
        // team is moving into his seat, both clockwise and anticlockwise is considered.
        for(int i = 1; i <= N; i++){
            for(int j = 1; j < i; j++){
                standUpWhenXVaries[i][0][0] = standUpWhenXVaries[i][0][0] + (x[j] < x[i] ? 1 : 0);
                standUpWhenXVaries[i][0][1] = standUpWhenXVaries[i][0][1] + (x[j] > x[i] ? 1 : 0);
            }
            for(int j = 1; j <= N; j++){
                standUpWhenXVaries[i][j][0] = standUpWhenXVaries[i][j - 1][0] + (y[j] < x[i] ? 1 : 0);
                standUpWhenXVaries[i][j][1] = standUpWhenXVaries[i][j - 1][1] + (y[j] > x[i] ? 1 : 0);
            }
        }

        // to make the calculation easy for the second team, we do a small tweak here, as if the entire
        // arrangement is flipped along the vertical and horizontal edge, so the second team orientation
        // is like of the first team and first team is like of second team
        for(int i = 1; i <= N; i++){
            x[i] = x[i] > N ? x[i] - N : x[i] + N;
            y[i] = y[i] > N ? y[i] - N : y[i] + N;
        }

        // calculate the minimum number of times to stand up when the first i-1 people in the second team
        // and the first j people in the first team have been seated, and the ith person from the second
        // team is moving into his seat, both clockwise and anticlockwise is considered.
        for(int i = 1; i <= N; i++){
            for(int j = 1; j < i; j++){
                standUpWhenYVaries[0][i][0] = standUpWhenYVaries[0][i][0] + (y[j] < y[i] ? 1 : 0);
                standUpWhenYVaries[0][i][1] = standUpWhenYVaries[0][i][1] + (y[j] > y[i] ? 1 : 0);
            }
            for(int j = 1; j <= N; j++){
                standUpWhenYVaries[j][i][0] = standUpWhenYVaries[j - 1][i][0] + (x[j] < y[i] ? 1 : 0);
                standUpWhenYVaries[j][i][1] = standUpWhenYVaries[j - 1][i][1] + (x[j] > y[i] ? 1 : 0);
            }
        }

        // Now both X and Y are varying, take all the four combinations above and try to find the minimum of all
        for(int i = 1; i <= N; i++){
            minStanding[i][0] = minStanding[i - 1][0] + min(standUpWhenXVaries[i][0][0], standUpWhenXVaries[i][0][1]);
        }
        for(int i = 1; i <= N; i++){
            minStanding[0][i] = minStanding[0][i - 1] + min(standUpWhenYVaries[0][i][0], standUpWhenYVaries[0][i][1]);
        }

        for(int i = 1; i <= N; i++){
            for(int j = 1; j <= N; j++){
                minStanding[i][j] = min(
                    minStanding[i][j - 1] + min(standUpWhenYVaries[i][j][0], standUpWhenYVaries[i][j][1]),
                    minStanding[i - 1][j] + min(standUpWhenXVaries[i][j][0], standUpWhenXVaries[i][j][1])
                );
            }
        }

        printf("%d",minStanding[N][N]);
    }
}
```

```
};

int main(){
    Roundt::main();
    return 0;
}
```

Source Code(Java):

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.StringTokenizer;

public class Roundt {

    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        int N = Integer.parseInt(br.readLine());
        int x[] = new int[N + 1];
        int y[] = new int[N + 1];

        StringTokenizer st = new StringTokenizer(br.readLine());
        for(int i = 1; i <= N; i++){
            x[i] = Integer.parseInt(st.nextToken());
        }

        st = new StringTokenizer(br.readLine());
        for(int i = 1; i <= N; i++){
            y[i] = Integer.parseInt(st.nextToken());
        }

        int standUpWhenXVaries[][][] = new int[N + 1][N + 1][2];
        int standUpWhenYVaries[][][] = new int[N + 1][N + 1][2];

        // calculate the number of times to stand up when the first i-1 people in the first team
        // and the first j people in the second team have been seated, and the ith person from the first
        // team is moving into his seat, both clockwise and anticlockwise is considered.
        for(int i = 1; i <= N; i++){
            for(int j = 1; j < i; j++){
                standUpWhenXVaries[i][j][0] = standUpWhenXVaries[i][j-1][0] + (x[j] < x[i] ? 1 : 0);
                standUpWhenXVaries[i][j][1] = standUpWhenXVaries[i][j-1][1] + (x[j] > x[i] ? 1 : 0);
            }
            for(int j = 1; j <= N; j++){
                standUpWhenXVaries[i][j][0] = standUpWhenXVaries[i][j-1][0] + (y[j] < x[i] ? 1 : 0);
                standUpWhenXVaries[i][j][1] = standUpWhenXVaries[i][j-1][1] + (y[j] > x[i] ? 1 : 0);
            }
        }

        // to make the calculation easy for the second team, we do a small tweak here, as if the entire
        // arrangement is flipped along the vertical and horizontal edge, so the second team orientation
        // is like of the first team and first team is like of second team
        for(int i = 1; i <= N; i++){
            x[i] = x[i] > N ? x[i] - N : x[i] + N;
            y[i] = y[i] > N ? y[i] - N : y[i] + N;
        }

        // calculate the number of times to stand up when the first i-1 people in the second team
        // and the first j people in the first team have been seated, and the ith person from the second
        // team is moving into his seat, both clockwise and anticlockwise is considered.
        for(int i = 1; i <= N; i++){
            for(int j = 1; j < i; j++){
                standUpWhenYVaries[0][i][0] = standUpWhenYVaries[0][i-1][0] + (y[j] < y[i] ? 1 : 0);
                standUpWhenYVaries[0][i][1] = standUpWhenYVaries[0][i-1][1] + (y[j] > y[i] ? 1 : 0);
            }
            for(int j = 1; j <= N; j++){
                standUpWhenYVaries[j][i][0] = standUpWhenYVaries[j-1][i][0] + (x[j] < y[i] ? 1 : 0);
                standUpWhenYVaries[j][i][1] = standUpWhenYVaries[j-1][i][1] + (x[j] > y[i] ? 1 : 0);
            }
        }

        int minStanding[][] = new int[N + 1][N + 1];
        // Now both X and Y are varying, take all the four combinations above and try to find the minimum of all
```

```

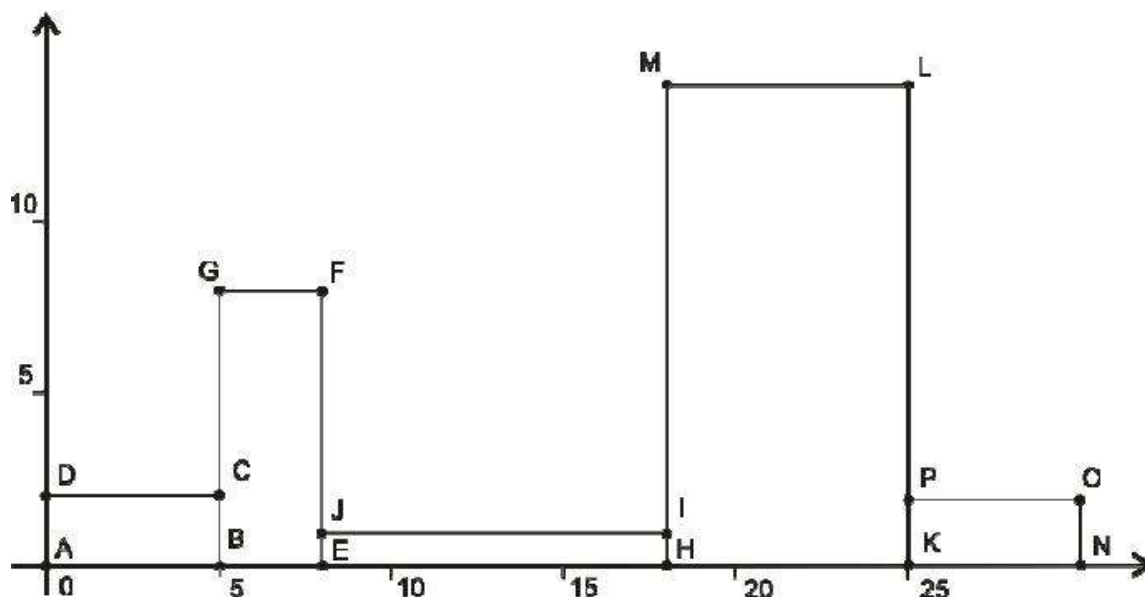
    for(int i = 1; i <= N; i++){
        minStanding[i][0] = minStanding[i - 1][0] +
            Math.min(standUpWhenXVaries[i][0][0], standUpWhenXVaries[i][0][1]);
    }
    for(int i = 1; i <= N; i++){
        minStanding[0][i] = minStanding[0][i - 1] +
            Math.min(standUpWhenYVaries[0][i][0], standUpWhenYVaries[0][i][1]);
    }

    for(int i = 1; i <= N; i++){
        for(int j = 1; j <= N; j++){
            minStanding[i][j] = Math.min(minStanding[i][j - 1] +
                Math.min(standUpWhenYVaries[i][j][0], standUpWhenYVaries[i][j][1]), minStanding[i - 1][j] +
                Math.min(standUpWhenXVaries[i][j][0], standUpWhenXVaries[i][j][1]));
        }
    }
    System.out.println(minStanding[N][N]);
}
}

```


32. Rectangles Perimeter

Given are n rectangles, numbered from 1 to n . We place them tightly on the axis OX , from left to right, according to rectangles' numbers. Each rectangle stays on the axis OX either by its shorter or by its longer side (see the picture below). Compute the length of the upper envelop line, i.e. perimeter's length of the obtained figure minus the length of the left, right and bottom straight line segments of the picture. Write program to find the maximum possible length of the upper envelop line.



Input

On the first line of the standard input, the value of n is written. On each of the next n lines, two integers are given – a_i and b_i – the side lengths of the i -th rectangle.

Constraints: $0 < n < 1000$; $0 < a_i < b_i < 1000$, for each $i = 1, 2, \dots, n$.

Output

On a line of the standard output, your program should write the result as a positive integer.

Sample Input:

Input:

5
2 5
3 8
1 10
7 14
2 5

Output:

68

Input:

3
1 1
100 100
100 100

Output:

300

Input:

3
100 100
1 1
100 100

Output:

399

Input:

3
10 10
20 1
100 10

Output:

148

Explanation: Test case 1

A configuration, that yields the maximum length of the upper envelop line, is presented on the picture. The upper envelop line consists of segments DC , CG , GF , FJ , JI , IM , ML , LP , and PO . The total length is $5 + 6 + 3 + 7 + 10 + 13 + 7 + 12 + 5 = 68$

Time limit: 1s

Source limit: 50000B

Solution:

This is an easy 'Dynamic Programing' problem. As usual this problem can be solved by two approaches, (i) Bottom Up Dynamic Programming or (ii) Recursive Approach

The basic underlying principle is, when we start with the given position of the first rectangle, then we can place the second rectangle in two ways; either in its given position or rotate the second rectangle in such a way that its length and height are interchanged. For each orientation of the second rectangle the third rectangle can again be oriented in two same ways as was done for second rectangle and so on. The first rectangle can again be oriented in two ways. Hence, for n rectangles we have 2^n distinct combination of rectangles. We have to find the maximum of all such combinations. For small n , a recursive approach can give our desired result within desired time limit, however, for large n this approach is not feasible. There we take a 'Dynamic Programing' approach. We take a matrix $dp[MAX][2]$ where $dp[i][0]$ or $dp[i][1]$ denotes the maximum perimeter when the i th rectangle is placed as it is or is rotated respectively. If we take a bottom up approach, i.e. start with the last rectangle, then it is ensured that before reaching state $dp[i][0]$ or $dp[i][1]$ we have already calculated the maximum perimeter when $n - i$ rectangles have already been placed.

We have included the recursive approach in the Java source code for completeness.

Source Code(C++):

```
#include <iostream>
#define MAXN 1000
using namespace std;

// stores the dimensions of the rectangles
int m[MAXN][2];
// the Dynamic Programing table
int dp[MAXN][2];

int abso(int a){
    return a < 0 ? -a : a;
}

class Mmaxper{
public:
    static void main(){
        // get the inputs
        int n;
        scanf("%d",&n);

        for(int i = 0; i < n; i++){
            scanf("%d %d",&m[i][0],&m[i][1]);
        }

        /*
         * recursively solve for a given rectangle, its next rectangle can be placed in two ways
         * k = 0 for normal position as given in input, k = 1 for rotated by 90 degrees where length
         * and breadth are interchanged. Then calculate the maximum perimeter for both cases.
         */
        for(int i = n - 1; i >= 0; i--) {
            if (i == n - 1) {
                dp[i][0] = m[i][0];
                dp[i][1] = m[i][1];
            }
            else {
                int a, b, k = 0;
                a = m[i][k] + abso(m[i][1 - k] - m[i + 1][1 - k]) + dp[i + 1][k];
                b = m[i][k] + abso(m[i][1 - k] - m[i + 1][k]) + dp[i + 1][1 - k];
                dp[i][k] = max(a, b);

                k = 1;
                a = m[i][k] + abso(m[i][1 - k] - m[i + 1][1 - k]) + dp[i + 1][k];
                b = m[i][k] + abso(m[i][1 - k] - m[i + 1][k]) + dp[i + 1][1 - k];
                dp[i][k] = max(a, b);
            }
        }
    }
}
```

```

        // find the maximum of both configurations
        printf("%d\n",max(dp[0][0], dp[0][1]));
    }
};

int main(){
    Mmaxper::main();
    return 0;
}

```

Source Code(Java):

```

import java.io.BufferedReader;
import java.io.InputStreamReader;

public class Mmaxper {

    // recursively solve for a given rectangle, its next rectangle can be placed in two ways
    // k = 0 for normal position as given in input, k = 1 for rotated positions where length
    // and breadth are interchanged
    public static int solve(int i, int k, int n, int m[][]){
        if(i == n - 1){
            return m[i][k];
        }else{
            int a = m[i][k] + Math.abs(m[i][1 - k] - m[i + 1][1 - k]) + solve(i + 1, k, n, m);
            int b = m[i][k] + Math.abs(m[i][1 - k] - m[i + 1][k]) + solve(i + 1, 1 - k, n, m);
            return Math.max(a, b);
        }
    }

    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        // get the inputs
        int n = Integer.parseInt(br.readLine());
        int m[][] = new int[n][2];

        String params[] = null;
        for(int i = 0; i < n; i++){
            params = br.readLine().split("\\s");
            m[i][0] = Integer.parseInt(params[0]);
            m[i][1] = Integer.parseInt(params[1]);
        }
        // We start with the first rectangle, first we take its given configuration,
        // and then calculate the maximum perimeter
        int a = solve(0, 0, n, m);
        // Then we rotate it by 90deg so that length is now height and height is now length
        // and then calculate the maximum perimeter
        int b = solve(0, 1, n, m);

        // find the maximum of two configurations
        System.out.println(Math.max(a, b));
    }
}

```

33.Coins Game

Asen and Boyan are playing the following game. They choose two different positive integers K and L, and start the game with a tower of N coins. Asen always plays first, Boyan – second, after that – Asen again, then Boyan, and so on. The boy in turn can take 1, K or L coins from the tower. The winner is the boy, who takes the last coin (or coins). After a long, long playing, Asen realizes that there are cases in which he could win, no matter how Boyan plays. And in all other cases Boyan being careful can win, no matter how Asen plays.

So, before the start of the game Asen is eager to know what game case they have. Write a program coins which help Asen to predict the game result for given K, L and N.

Input

The input describes m games.

The first line of the standard input contains the integers K, L and m, $1 < K < L < 10$, $3 < m < 50$. The second line contains m integers N1, N2, ..., Nm, $1 \leq N_i \leq 1\,000\,000$, $i = 1, 2, \dots, m$, representing the number of coins in each of the m towers

Output

The standard output contains a string of length m composed of letters A and B. If Asen wins the ith game (no matter how the opponent plays), the ith letter of the string has to be A. When Boyan wins the ith game (no matter how Asen plays), the ith letter of the string has to be B.

Example

Input:	Input:	Input:
2 3 5	3 8 10	4 5 1
3 12 113 25714 88888	3 12 24 79 113 25714 90 88888 5674 65342	6
Output:	Output:	Output:
ABAAB	AABBAABAAB	A

Time limit: 1s
Source limit: 50000B

Solution:

This is an easy 'Dynamic Programing' problem. This problem can be solved by Bottom Up Dynamic Programing. Let us consider an array w[i] to dynamically store who wins if Asen starts at w[i]. w[i] = A, if Asen wins when Asen starts with n[i] and w[i] = B, if Boyan wins when Asen starts with n[i].

When $i = 1$ or K or L, Asen will win if Asen starts with i. When $1 < i < K$, i.e. from w[2] to w[K - 1], Asen and Boyan will alternate starting with w[2] = B because each player can take only 1 in this range. So, if Asen starts with 2, he can take only 1 and Boyan will take the remaining 1 and win. When Asen starts with 3, Asen will take 1, Boyan will take 1 and Asen will take the last 1, so Asen wins and so on.

When $K < i < L$ i.e. from w[K + 1] to w[L - 1] each player can take either K or 1. Hence, check for both these cases, when Asen selects either of them, whether is there any possibility for Asen to win? Then Asen will obviously choose that path and ignore the other one. If both cases lead to Asen being the winner, Asen can choose either of them. Remember, we are not here to find what will Asen choose, we are just interested whether is there a possibility for Asen to win, and since each player always plays the best move possible so Asen will always select that case which makes him the winner.

Similarly, When $L < i < \text{MAX}$ i.e. from w[L + 1] to w[MAX] each player can take either L or K or 1. Hence, check for all three cases, when Asen selects any one of them, whether is there any possibility for Asen to win? Then Asen will obviously choose that path and ignore the others. If all cases lead to Asen being the winner, Asen can choose any one of them. In all the cases, when there is no possibility for Asen to win then Boyan wins irrespective of how Asen plays.

We have taken this array with respect to Asen because Asen always starts the game. Since, the game can have only one winner and it should be either of the two, so the matrix w will be a mirror if we had taken that with respect to Boyan, only the values would be reversed. So if $w[i] = A$ when Asen starts with i , it also means that $w[i] = B$ when Boyan starts with i , if $w[i] = B$ when Asen starts with i , it also means that $w[i] = A$ when Boyan starts with i . In one word if $w[i] = A$, it means that, A will win if A starts at i or B will win if B starts at i . if $w[i] = B$, it means that, Asen will loose if Asen starts at i or Boyan will loose if Boyan starts at i .

Source Code(C++):

```
#include <iostream>
#include <vector>
using namespace std;

class Mcoins {
public:
    static void main() {
        // get the inputs
        int K, L, m;
        scanf("%d %d %d", &K, &L, &m);

        // optimization, to store the max value of n[i], the dp should be of that length only
        int MAX = 0;
        vector<int> n(m + 1);
        for(int i = 1; i <= m; i++){
            scanf("%d", &n[i]);
            MAX = MAX < n[i]? n[i]: MAX;
        }
        MAX++;

        /* w is the dp array to dynamically store who wins if A starts at w[i].
         * w[i] = A, if A wins when A starts with n[j]
         * w[i] = B, if B wins when A starts with n[j]
         */
        vector<char> w(MAX, 'X');
        for(int i = 1; i < MAX; i++){
            // When i = 1 or i == K or L, Asen will win if Asen starts with i
            if(i == 1 || i == K || i == L){
                w[i] = 'A';
            }else if(i > 1 && i < K){
                // When 1 < i < K, i.e. from w[2] to w[K - 1], Asen and Boyan will alternate
                // starting with w[2] = B because each player can take only 1 in this range.
                w[i] = w[i - 1] == 'A' ? 'B' : 'A';
            }else if(i > K && i < L){
                // When K < i < L i.e. from w[K + 1] to w[L - 1] each player can take either K or 1. Hence, check
                // for both these cases, when Asen selects either of them, whether is there any possibility for
                // Asen to win? Then Asen will obviously choose that path and ignore the other one.
                if(w[i - K] != 'A' || w[i - 1] != 'A'){
                    w[i] = 'A';
                }else{
                    w[i] = 'B';
                }
            }else if(i > L){
                // Similarly, When L < i < MAX i.e. from w[L+1] to w[MAX] each player can take either L or K or 1.
                // Hence, check for all three cases, when Asen selects any one of them, whether is there any
                // possibility for Asen to win? Then Asen will obviously choose that path and ignore the others.
                if(w[i - L] != 'A' || w[i - K] != 'A' || w[i - 1] != 'A'){
                    w[i] = 'A';
                }else{
                    w[i] = 'B';
                }
            }
        }
        for(int i = 1; i <= m; i++){
            printf("%c", w[n[i]]);
        }
    }
};

int main() {
    Mcoins::main();
    return 0;
}
```

Source Code(Java):

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.Arrays;
import java.util.StringTokenizer;

public class Mcoins {
    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        // get the inputs
        StringTokenizer st = new StringTokenizer(br.readLine(), " ");
        int K = Integer.parseInt(st.nextToken());
        int L = Integer.parseInt(st.nextToken());
        int m = Integer.parseInt(st.nextToken());

        st = new StringTokenizer(br.readLine(), " ");

        // optimization, to store the max value of n[i], the dp should be of that length only
        int MAX = 0;
        int n[] = new int[m + 1];
        for(int i = 1; i <= m; i++){
            n[i] = Integer.parseInt(st.nextToken());
            MAX = MAX < n[i]? n[i]: MAX;
        }
        MAX++;

        /* w is the dp array to dynamically store who wins if A starts at w[i].
        * w[i] = A, if A wins when A starts with n[j]
        * w[i] = B, if B wins when A starts with n[j]
        */
        char w[] = new char[MAX];
        Arrays.fill(w, 'X');

        for(int i = 1; i < MAX; i++){
            // When i = 1 or K or L, Asen will win if Asen starts with i
            if(i == 1 || i == K || i == L){
                w[i] = 'A';
            } else if(i > 1 && i < K){
                // When 1 < i < K, i.e. from w[2] to w[K - 1], Asen and Boyan will alternate
                // starting with w[2] = B because each player can take only 1 in this range.
                w[i] = w[i - 1] == 'A' ? 'B' : 'A';
            } else if(i > K && i < L){
                // When K < i < L i.e. from w[K + 1] to w[L - 1] each player can take either K or 1. Hence, check
                // for both these cases, when Asen selects either of them, whether is there any possibility for
                // Asen to win? Then Asen will obviously choose that path and ignore the other one.
                if(w[i - K] != 'A' || w[i - 1] != 'A'){
                    w[i] = 'A';
                } else{
                    w[i] = 'B';
                }
            } else if(i > L){
                // Similarly, When L < i < MAX i.e. from w[L+1] to w[MAX] each player can take either L or K or 1.
                // Hence, check for all three cases, when Asen selects any one of them, whether is there any
                // possibility for Asen to win? Then Asen will obviously choose that path and ignore the others.
                if(w[i - L] != 'A' || w[i - K] != 'A' || w[i - 1] != 'A'){
                    w[i] = 'A';
                } else{
                    w[i] = 'B';
                }
            }
        }
        for(int i = 1; i <= m; i++){
            System.out.print((char)w[n[i]]);
        }
    }
}
```

34. Aligning Strings

There are given two strings, A and B. An expansion of some string X is a string created by adding or inserting any number (zero, one or more) of blanks anywhere in the string, or in the beginning or the end of the string. Eg., if the string X is 'abcbcd', then the strings 'abcb-cd', '-a-bcbcd-' and 'abcb-cd-' are expansions of the string X (blanks are denoted by the character '-').

If A1 is an expansion of the string A, and B1 is an expansion of the string B, and if A1 and B1 are of the same length, then we define the distance of the strings A1 and B1 as the sum of the distances of the characters on the same positions in these strings. We define the distance of two characters as the absolute difference of their ASCII codes, except the distance of the blank and another character, which is given (and equal for all characters).

You are to write a program which finds the expansions A1 and B1 of strings A and B, that have the smallest difference.

Input

The first line of the input file consists of the string A, and the second line of string B. They are consisted only of lower case characters of the english alphabet (a-z), and the number of characters in any of the strings is less than or equal to 2000.

The third line consists of an integer K, $1 \leq K \leq 100$, the distance of the blank and the other characters.

Output

The first only line of the input file should consist of the smallest distance as defined in the text of the task.

Example

Input:

cmc
snmn
2

Output:

10

Input:

koiv
ua
1

Output:

5

Input:

mj
jao
4

Output:

12

Time limit: 1s

Source limit: 50000B

Solution:

We will use the dynamic programming algorithm to find the optimal alignment between two strings. First, we will compute the optimal alignment for every substring and save those scores in a matrix. For two strings, A of length m and B of length n, $D[i, j]$ is defined to be the minimum difference of aligning the two substrings $A[1 \dots i]$ and $B[1 \dots j]$. The minimum score for the alignment is precisely $D[m, n]$, the last value in the table. We will compute $D[m, n]$ by computing $D[i, j]$ for all values of i and j where i ranges from 0 to m and j ranges from 0 to n.

The recurrence relation establishes a relationship between $D[i, j]$ and values of D with indices smaller than i and j. When there are no smaller values of i and j then the values of $D[i, j]$ must be stated explicitly in the base conditions. The base conditions are used to calculate the scores in the first row and column where there are no matrix elements above and to the left. This corresponds to aligning with strings of '-'. After the base conditions have been established, the recurrence relation is used to compute all values of $D[i, j]$ in the table.

The recurrence relation is:

$$D[i, j] = \max \{ D[i-1, j-1] + \text{absDiff}(A[i] - B[j]), \quad D[i-1, j] + \text{value of '-'}, \\ D[i, j-1] + \text{value of '-' } \}$$

In other words, if you have an optimal alignment up to $D[i - 1, j - 1]$ there are only three possibilities of what could happen next:

1. the characters for $A[i]$ and $B[j]$ match,
2. a '-' is introduced in B and
3. a '-' is introduced in A

It is not possible to add a gap to both substrings. The maximum of the three scores will be chosen as the optimal score and is written in matrix element $D[i, j]$. Refer to <http://www.biorecipes.com/DynProgBasic/code.html>.

Source Code(C++):

```
#include <iostream>
#include <string>
#define MAX 2001
using namespace std;

// For two strings, A of length m and B of length n, D[i, j] is defined to be the
// minimum difference of aligning the two substrings A[1 ... i] and B[1 ... j].
int dp[MAX][MAX];

int abs(int a){
    return a < 0 ? -a : a;
}

class Mblast {
public:
    static void main() {

        string A, B;
        int K;

        // get the inputs
        cin >> A;
        cin >> B;
        cin >> K;

        int m = A.size();
        int n = B.size();

        // Establish the base conditions for D[i, j] when there are no smaller values of i and j
        // The base conditions are stated in the first row and column where there are no
        // matrix elements above and to the left.
        // This corresponds to aligning with strings of '-'.
        for (int i = 1; i <= m; i++) {
            dp[i][0] = K * i;
        }
        for (int j = 1; j <= n; j++) {
            dp[0][j] = K * j;
        }

        /* For two strings, A of length m and B of length n, D[i, j] is defined to be the best score of aligning
        * the two substrings A[1 ... i] and B[1 ... j]. The best score for the alignment is precisely D[m, n],
        * the last value in the table. We will compute D[m,n] by computing D[i, j] for all values of i and j
        * where i ranges from 0 to m and j ranges from 0 to n. The recurrence relation is
        * D[i, j] = max{D[i-1, j-1] + absDiff(A[i] - B[j]), D[i-1, j] + value of '-', D[i, j-1] + value of '-'}
        */
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                dp[i][j] = min(
                    min(K + dp[i - 1][j], K + dp[i][j - 1]),
                    dp[i - 1][j - 1] + abs(A[i - 1] - B[j - 1]));
            }
        }
        printf("%d\n", dp[m][n]);
    }
};

int main() {
    Mblast::main();
    return 0;
}
```



```
}
```

Source Code(Java):

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Mblast {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        char[] st1 = br.readLine().toCharArray();
        char[] st2 = br.readLine().toCharArray();
        int K = Integer.parseInt(br.readLine());

        int m = st2.length;
        int n = st1.length;

        // For two strings, A of length m and B of length n, D[i, j] is defined to be the
        // minimum difference of aligning the two substrings A[1 ... i] and B[1 ... j].
        int[][] dp = new int[m + 1][n + 1];

        // Establish the base conditions for D[i, j] when there are no smaller values of i and j
        // The base conditions are stated in the first row and column where there are no
        // matrix elements above and to the left.
        // This corresponds to aligning with strings of '-'.
        for (int i = 1; i <= m; i++) {
            dp[i][0] = K * i;
        }
        for (int j = 1; j <= n; j++) {
            dp[0][j] = K * j;
        }

        /* For two strings, A of length m and B of length n, D[i, j] is defined to be the best score of aligning
        * the two substrings A[1 ... i] and B[1 ... j]. The best score for the alignment is precisely D[m, n],
        * the last value in the table. We will compute D[m,n] by computing D[i, j] for all values of i and j
        * where i ranges from 0 to m and j ranges from 0 to n. The recurrence relation is
        * D[i, j] = max{D[i-1, j-1] + absDiff(A[i] - B[j]), D[i-1, j] + value of '-', D[i, j-1] + value of '-'}
        */
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                dp[i][j] = Math.min(
                    Math.min(K + dp[i - 1][j], K + dp[i][j - 1]),
                    dp[i - 1][j - 1] + Math.abs(st1[j - 1] - st2[i - 1]));
            }
        }
        System.out.println(dp[m][n]);
    }
}
```

35. Card Sorting

Dave's little son Maverick likes to play card games, but being only four years old, he always lose when playing with his older friends. Also, arranging cards in his hand is quite a problem to him.

When Maverick gets his cards, he has to arrange them in groups so that all the cards in a group are of the same color. Next, he has to sort the cards in each group by their value – card with lowest value should be the leftmost in its group. Of course, he has to hold all the cards in his hand all the time.

He has to arrange his cards as quickly as possible, i.e. making as few moves as possible. A move consists of changing a position of one of his cards.

Write a program that will calculate the lowest number of moves needed to arrange cards.

Input

The first line of input file contains two integers C, number of colors ($1 < C < 4$), and N, number of cards of the same color ($1 < N < 100$), separated by a space character.

Each of the next C*N lines contains a pair of two integers X and Y, $1 < X < C$, $1 < Y < N$, separated by a space character.

Numbers in each of those lines determine a color (X) and a value (Y) of a card dealt to little Maverick. The order of lines corresponds to the order the cards were dealt to little Maverick. No two lines describe the same card.

Output

The first and only line of output file should contain the lowest number of moves needed to arrange the cards as described above.

Example

Input

2 2
2 1
1 2
1 1
2 2

Output

2

Input

4 1
2 1
3 1
1 1
4 1

Output

0

Input

3 2
3 2
2 2
1 1
3 1

Output

2

Output

2

Input

1 1
1 1

Output

0

Input

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

Output

6

Time limit: 1s
Source limit: 50000B

Solution:

This is a moderate 'Dynamic Programing' problem. Let us say we have 10 cards, numbered 1 to 10 , and say they are ordered in the following order 9 4 2 6 7 10 3 1 5 8 , from left to right. If we are asked to remove some set of cards from the set and then place them, so that finally the cards are sorted in increasing order, then we would be removing the following cards from the above list and then place them in order.

9, 2, 10, 3, 1, 5 - the cards left would be 4, 6, 7, 8 - which is LIS(Longest Increasing subsequence) in the given sequence. This would require minimum moves - 6 moves [6 misplaced cards].

But again we have cards with different colors, so the colors can have different arrangements. We try with all possible arrangement of colors.

Let's say we have three different colors - R, G, B. We will have 6 ways of ordering the colors

R < G < B

R < B < G

B < R < G

B < G < R

G < B < R - this means G - 4 [green card with no. 4 on it] will be lower than a R - 1 [red card with no. 1 on it], since G < R

G < R < B

After all possibilities choose the order with the maximum ordered cards and hence that would give the minimum moves.

Source Code(C++):

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

class Mcards {
private:
    int C, N, maxOrderedCards, totalCards;
    // all permutations of the order in which the colors should be arranged
    vector<int> colorsOrder;

    // represent a single card
    class Card {
    public:
        int color, value;
        Mcards *mcards;

        Card(){}
        Card(int c, int v, Mcards *mc){
            color = c;
            value = v;
            mcards = mc;
        }
        // Used to find whether cards are arranged in ascending order. First it checks two cards by color
        // and then if the colors are same it checks by comparing their values. The order of colors is given
        // by vector<int> colorsOrder
        bool operator > (const Card& c) const {
            if (color != c.color) return mcards->colorsOrder[color - 1] > mcards->colorsOrder[c.color - 1];
            return value > c.value;
        }
    };

    // store all the cards
    vector<Card> cards;

    // Longest Increasing subsequence.
    // This method finds how many cards are already arranged in sequence.
    int lis() {
        vector<int> q(totalCards, 0);
```

```

    q[0] = 1;
    int max = 1;
    for (int k = 1; k < totalCards; k++) {
        q[k] = 1;
        for (int j = 0; j < k; j++) {
            if (cards[k] > cards[j] && q[k] < q[j] + 1) {
                q[k] = q[j] + 1;
            }
            if (q[k] > max) {
                max = q[k];
            }
        }
    }
    return max;
}

public:
    Mcards(){
        maxOrderedCards = 0;
    }

    void main() {
        scanf("%d %d",&C, &N);

        totalCards = C * N;

        colorsOrder = vector<int>(C);
        for (int i = 1; i <= C; i++) {
            colorsOrder[i - 1] = i;
        }

        cards = vector<Card>(totalCards);
        for (int i = 0, c, v; i < totalCards; i++) {
            scanf("%d %d",&c, &v);

            Card card(c, v, this);
            cards[i] = card;
        }

        do {
            // for all permutaions of color order determine the largest number of cards which are
            // arranged sequentially. Then take max of all those sequences.
            maxOrderedCards = max(lis(), maxOrderedCards);
        } while ( next_permutation (colorsOrder.begin(), colorsOrder.end()) );

        // Leastmoves = totalCards - Max Cards which are already ordered
        int leastMoves = totalCards - maxOrderedCards;
        printf("%d\n", leastMoves);
    }
};

int main() {
    Mcards mcards;
    mcards.main();
    return 0;
}

```

Source Code(Java):

```

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.Iterator;
import java.util.StringTokenizer;

public class Mcards {

    private int C, N, maxOrderedCards = 0, totalCards;
    // all permutations of the order in which the colors should be arranged
    private int[] colorsOrder;
    // store all the cards
    private Card[] cards;

    // Longest Increasing subsequence.

```

```

// This method finds how many cards are already arranged in sequence.
private int lis() {
    int q[] = new int[totalCards];
    q[0] = 1;

    int max = 1;
    for (int k = 1; k < totalCards; k++) {
        q[k] = 1;
        for (int j = 0; j < k; j++) {
            if (cards[k].compareTo(cards[j]) == 1 && q[k] < q[j] + 1) {
                q[k] = q[j] + 1;
            }
            if (q[k] > max) {
                max = q[k];
            }
        }
    }
    return max;
}

public void solve() throws Exception {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

    StringTokenizer st = new StringTokenizer(br.readLine(), " ");
    C = Integer.parseInt(st.nextToken());
    N = Integer.parseInt(st.nextToken());

    totalCards = C * N;

    colorsOrder = new int[C];
    for (int i = 1; i <= C; i++) {
        colorsOrder[i - 1] = i;
    }

    cards = new Card[totalCards];
    for (int i = 0; i < totalCards; i++) {
        st = new StringTokenizer(br.readLine(), " ");
        cards[i] = new Card(Integer.parseInt(st.nextToken()),
            Integer.parseInt(st.nextToken()));
    }

    Permuter permuter = new Permuter(colorsOrder);
    // for all permutaions of color order determine the largest number of cards
    // which are arranged sequentially. Then take max of all those sequences.
    while (permuter.hasNext()) {
        permuter.next();
        maxOrderedCards = Math.max(lis(), maxOrderedCards);
    }

    // Leastmoves = totalCards - Max Cards which are already ordered
    int leastMoves = totalCards - maxOrderedCards;
    System.out.println(leastMoves);
}

public static void main(String[] args) throws Exception {
    Mcards mcards = new Mcards();
    mcards.solve();
}

// represent a single card
class Card implements Comparable<Card> {
    private int color, value;

    public Card(int color, int value) {
        this.color = color;
        this.value = value;
    }

    // Used to find whether cards are arranged in ascending order. First it
    // checks two cards by color and then if the colors are same it checks
    // by comparing their values. The order of colors is given by int[] colorsOrder
    @Override
    public int compareTo(Card o) {
        if (color != o.color)

```

```

        return colorsOrder[color - 1] > colorsOrder[o.color - 1] ? 1
            : -1;
        return value > o.value ? 1 : -1;
    }
}

// permutation generator
class Permuter implements Iterator<int[]>, Iterable<int[]> {

    /**
     * An integer array backing up the original one to keep track of the
     * indices.
     */
    private int[] indices;

    /**
     * The total number of variations to be computed.
     */
    final private int total;

    /**
     * The variations still to go.
     */
    private int numLeft;

    /**
     * The elements the operator works upon.
     */
    protected int[] elements;

    /**
     * Initialize a new permuter, with given array of elements to permute.
     *
     * @param elements
     *         The elements to permute.
     * @post The total number is set to the factorial of the number of
     *       elements. | new.getTotal() == factorial(elements.length)
     * @post The number of permutations left is set to the total number. |
     *       new.getNumLeft() == new.getTotal()
     */
    public Permuter(final int[] elements) {
        this.indices = new int[elements.length];
        this.elements = elements.clone();
        this.total = this.initializeTotal(elements.length);
        this.reset();
    }

    /**
     * Compute the total number of elements to return.
     *
     * @param n
     *         The number of elements the operator works on.
     * @param r
     *         The size of the arrays to return.
     * @return The total number of elements is always bigger than 0. |
     *         result >= 0
     */
    private int initializeTotal(int n) {
        return factorial(n);
    }

    /**
     * Initialize the array of indices. By default, it is initialized with
     * incrementing integers.
     */
    protected void initializeIndices() {
        for (int i = 0; i < this.indices.length; i++) {
            this.indices[i] = i;
        }
    }

    /**
     * Compute the factorial of n.
     */

```

```

* @param n
*     The number of which the factorial is to be computed.
*     Should be less than 20, otherwise integer overflow occurs.
* @return The factorial of n, <code>n!</code>.
*/
private int factorial(final int n) {
    int fact = 1;
    for (int i = n; i > 1; i--) {
        fact = fact * i;
    }
    return fact;
}

/**
 * Swap the elements at positions a and b, both from the index array and
 * from the element array.
 */
* @param a
*     The first index of the elements to be swapped.
* @param b
*     The second index of the elements to be swapped.
* @post The elements at indices a and b of the array of indices are
*     swapped. | new.indexes[a] = indexes[b] | new.indexes[b] =
*     indexes[a]
*/
private void swap(final int a, final int b) {
    final int temp = this.indexes[a];
    this.indexes[a] = this.indexes[b];
    this.indexes[b] = temp;
}

/**
 * Compute the next array of indices.
 */
private void computeNext() {
    // find the rightmost element that is smaller than the element at its right
    int i = this.indexes.length - 1;
    while (this.indexes[i - 1] >= this.indexes[i]) {
        i = i - 1;
    }
    // find the rightmost element that is bigger then the other one
    int j = this.indexes.length;
    while (this.indexes[j - 1] <= this.indexes[i - 1]) {
        j = j - 1;
    }
    // swap them (always is i < j)
    this.swap(i - 1, j - 1);
    // now the elements at the right of i are in descending order, so reverse them all
    i++;
    j = this.indexes.length;
    while (i < j) {
        this.swap(i - 1, j - 1);
        i++;
        j--;
    }
}

/**
 * Compute the result, based on the given array of indices.
 */
* @param indexes
*     An array of indices into the element array.
* @return An array consisting of the elements at positions of the given
*     array. | result[i] == elements[indexes[i]]
*/
private int[] getResult(final int[] indexes) {
    for (int i = 0; i < indexes.length; i++) {
        colorsOrder[i] = elements[indexes[i]];
    }
    return colorsOrder;
}

/**
 * Reset the iteration.

```

```

*
* @post The number of iterations to go is the same as the total number
*       of iterations. | new.getNumLeft() == getTotal()
*/
public void reset() {
    this.initializeIndices();
    this.numLeft = this.total;
}

/**
 * A combinatoric operator is itself an iterator.
 *
 * @return Itself. | result == this
 */
@Override
public Iterator<int[]> iterator() {
    return this;
}

/**
 * Returns <tt>true</tt> if the iteration has more elements. This is the
 * case if not all n! permutations have been covered.
 *
 * @return The number of permutations to go is bigger than zero. |
 *         result == getNumLeft() > 0;
 */
@Override
public boolean hasNext() {
    return this.numLeft > 0;
}

/**
 * Compute the next combination.
 */
@Override
public int[] next() {
    if (this.numLeft != this.total) {
        this.computeNext();
    }
    this.numLeft--;
    return this.getResult(this.indices);
}

/**
 * Not supported.
 */
@Override
public void remove() {
    throw new UnsupportedOperationException();
}
}
}

```