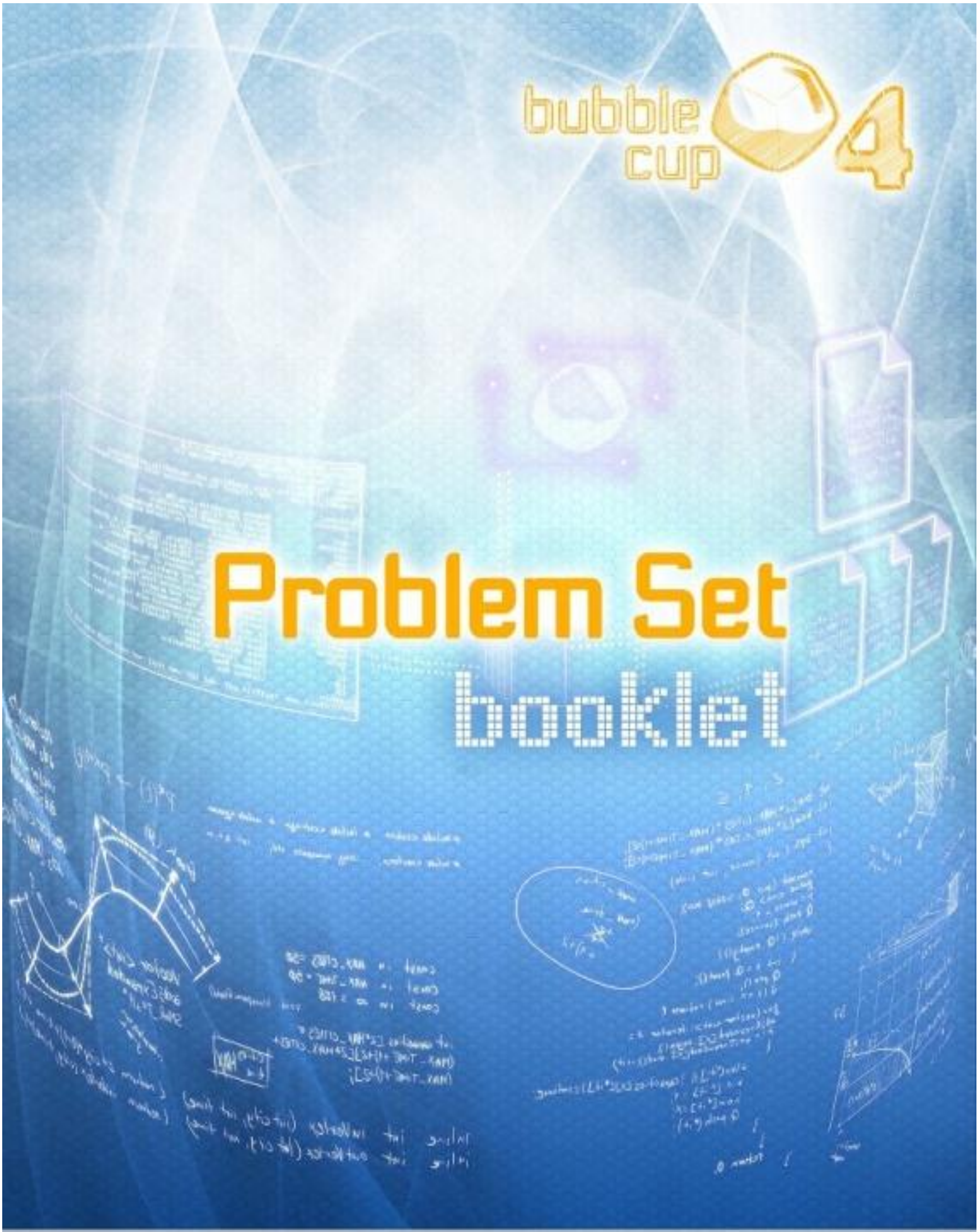
bubble cup 4

Problem Set booklet



Microsoft | Development Center Serbia

student programming contest

BUBBLE CUP 2011

Student programming contest

Microsoft Development Center Serbia

Problem set & Analysis
from Finals and Qualifications

Belgrade, 2011

Scientific committee:

Andreja Ilić
Andrija Jovanović
Milan Vugdelija
Milan Novaković
Stevan Jončić
Dimitrije Filipović
Dražen Žarić
Miloš Lazarević
Miloš Milovanović

Qualification analyses:

Alexander Georgiev
Nikola Milosavljević
Aleksandar Ilić
Gustav Matula
Boris Grubić
Vanja Petrović Tanković
Dušan Zdravković
Dimitrije Dimić
Stefan Stojanović
Yordan Chaparov
Rajko Nenadov
Demjan Grubić
Mladen Radojević
Slobodan Mitrović
Andrija Jovanović
Andreja Ilić

Cover:

Sava Čajetinac

Typesetting:

Andreja Ilić

Proofreader:

Andrija Jovanović

Volume editor:

Dragan Tomić

Contents

Preface

Dear Participant,

Thank you for participating in the fourth edition of the Bubble Cup. On behalf of Microsoft Development Center Serbia (MDCS), I wish you a warm welcome to Belgrade and I hope that you will enjoy yourself.

MDCS has a keen interest in putting together a world class event. Most of our team members participated in similar competitions in the past and have passion for solving difficult technical problems.

This edition of the Bubble Cup is special. It is the most international event that we had so far. This year, teams from Serbia, Bosnia, Croatia and Bulgaria are competing in the finals. In our evaluation of participants, we observed that at least a dozen IOI Olympians from the region will participate in the Bubble Cup finals this year. You folks are the best that this region can offer!

Given that we live in a world where technological innovation will shape the future, your potential future impact on humankind will be great. Take this opportunity to advance your technical knowledge and to build relationships that could last you a lifetime.

Thanks,
Dragan Tomić
MDCS Group Manager/Director

About Bubble Cup and MDCS

BubbleCup is a coding contest started by Microsoft Development Center Serbia in 2008 with a purpose of creating a local competition similar to the ACM Collegiate Contest, but soon that idea was outgrown and the vision was expanded to attracting talented programmers from the entire region and promoting the values of communication, companionship and teamwork.

The contest has been growing in popularity with each new iteration. In its first year close to 100 participants took part and this year, 2011, it attracted more than 500 participants from 8 different countries.

This year the emphasis was on keeping intact all of the things that made BubbleCup work in previous years but taking every opportunity to tweak and subtly improve the format of the contest. Among others, the novelties include complex, serious qualification problems that are chosen to take advantage of the long period of time given to the contestants, as well as problems in the final which make teams think about some constraints that are rarely tested in this type of competition.

Microsoft Development Center Serbia (MDCS) was created with a mission to take an active part in conception of novel Microsoft technologies by hiring unique local talent from Serbia and the region. Our teams contribute components to some of Microsoft's premier and most innovative products such as SQL Server, Office & Bing. The whole effort started in 2005, and during the last 6 years a number of products came out as a result of great team work and effort.

Our development center is becoming widely recognized across Microsoft as a center of excellence for the following domains: computational algebra engines, pattern recognition, object classification, computational geometry and core database systems. The common theme uniting all of the efforts within the development center is applied mathematics. MDCS teams maintain collaboration with engineers from various Microsoft development centers around the world (Redmond, Israel, India, Ireland and China), and Microsoft researchers from Redmond, Cambridge and Asia.

Bubble Cup Finals 2011

The Bubble Cup Finals, like the previous years, were held at the School of Electrical Engineering in Belgrade. The competitors had five hours for eight problems. In this booklet you will find nine problems - problem I (generalization of problem E) was removed from the actual competition because of its difficulty. For ties, the same rules applied as in previous years: if two or more teams solved the same number of problems, the one who needed the least time was ranked best. Additionally, teams received bonus points depending on their qualification results, but for each problem there were time penalties if a team had incorrect submissions before managing to solve it (Problem A turned out to be very good at making teams collect penalty points). Programming style is not considered in this contest – contestants are free to code in whatever style they prefer and documentation is not required.

This year, problems from the finals are slightly easier than the last year. The emphasis was on stimulating students' creativity - some of the problems were not so standard for programming competitions. The idea behind this was to test contestants in some areas for which they were not very well-prepared. The Scientific Committee is pleasantly surprised with the skill the competitors have shown. Three problems were solved by all teams, while on the other side there was only one problem that no team managed to solve.

Team **Suit Up!** won the competition (improving on last year, when they were second). The second place went to **wehmuma**. They managed to solve six problems and edge out **ex1t** thanks to a smaller time penalty.

This year the Scientific Committee decided to give some special awards:

- Award: **Silver lightning**
  Team **wehmuma** - Rumen Hristov, Georgi Georgiev and Alex Ivanov, for the first accepted solution to a problem.
- Award: **system("pause");**
  Team **Gastartbubblers** - Rajko Nenadov, Slobodan Mitrovic and Nikola Skoric, for lifetime achievement in programming excellence and spreading the BubbleCup spirit.
- Award: **Hardcoding Expert**
  Team **v.haralampiev** - Vladislav Haralampiev, for being the first to solve LR Primes despite a lack of manpower.

The Scientific Committee would like to congratulate all of you, teams and individuals, for the hard work you put in solving the problems we selected, and for your enthusiasm and interest in the Bubble Cup competition.

The final scoreboard

Place	Team	Team crew	Score	Penalty
1.	Suit Up!	Gustav Matula, XV. Gimnazija Zagreb Ivan Katanić, Gimnazija Pozega Ivica Kičić, V. gimnazija Zagreb	7	824
2.	wehmuma	Rumen Hristov, High School of Natural Science and Mathematics Georgi Georgiev, SMG Alex Ivanov, Nancho Popovich Maths and Science High School	6	760
3.	ex1t	Alexander Georgiev, Sofia University George Acev, Sofia University Preslav Le, Sofia University	6	1019
4.	Gari	Demjan Grubic, Gimn. Jovan Jovanovic Zmaj Boris Grubic, Gimnazija Jovan Jovanovic Zmaj Mario Cekic, Gimnazija Jovan Jovanovic Zmaj	5	480
5.	Gastarbubblers	Nemanja Skoric, ETH Zurich Slobodan Mitrovic Rajko Nenadov, ETH Zurich	5	538
6.	Tim Raketa	Viktor Braut, FER Zagreb Frane Kurtović, FER Zagreb Anton Grbin, FER Zagreb	5	589
7.	Drišlje	Nikola Stojiljkovic, Gimnazija Svetozar Markovic, Nis Nikola Smiljkovic, Gimnazija Svetozar Markovic, Nis Nikola Stevanovic, Gimnazija Svetozar Markovic, Nis	5	734
8.	Magic 3	Maja Kabiljo, Racunarski Fakultet Miroslav Bogdanović, Racunarski Fakultet Milos Stankovic, Racunarski Fakultet	5	734
9.	kikiriki i pivo	Mladen Radojevic, ETF Beograd Ugljesa Stojanovic, RAF/ETF Beograd Aleksandar Tomic, ETF Beograd	5	844
10.	I like it RAF	Nenad Božidarević, Računarski fakultet, Beograd Vanja Petrović Tanković, Računarski fakultet, Beograd Aleksandar Milovanovic, Računarski fakultet, Beograd	5	949
11.	S-Force	Dusan Zdravkovic, Gimnazija Svetozar Markovic Nis Dimitrije Dimic, Gimnazija Svetozar Markovic Nis Stefan Stojanovic, gimnazija Svetozar Markovic Nis	4	358
12.	The Ninjas	Nikola Milosavljevic, PMF Nis Marija Cvetkovic, PMF Nis Aleksandar Trokicic, PMF Nis	4	408
13.	doktori	Andrija Milicevic, University of Zagreb - School of Medicine Marin Smiljanic, FER Zagreb Goran Gasic, FER Zagreb	4	431
14.	v.haralampiev	Vladislav Haralampiev, SMG	4	474
15.	[BG] Coders	Vladimir Vladimirov Yordan Chaparov, Atanas Radev Yasen Trigonov, OMG	4	478
16.	TPPH	Dominik Gleich, XV. Gimnazija Zvonimir Medić, XV. Gimnazija Drago Plecko, XV. Gimnazija	4	515
17.	Royal Randoms	Nina Radojicic, Matematički fakultet, Beograd Stefan Miskovic, Matematički fakultet, Beograd Stefan Jankovic, Matematički Fakultet, Beograd	4	902
18.	Firewall	Damir Ferizovic, MSS Bosanski Petrovac Daniel Ferizovic, MSS Bosanski Petrovac Aleksandar Ivanovic, Prva kragujevačka gimnazija	3	213
19.	Gimnazija Sombor	Predrag Ilkic, Gimnazija Veljko Petrovic Slobodan Ilkic, Gimnazija Sombor Dejan Pekter, Gimnazija Veljko Petrovic	3	414

Statistics from finals

ID	Problem name	Number of teams with correct solutions	Number of teams with at least one submission attempt	Total percentage of accepted submissions
A	Card	13	19	09.77%
B	Rook	19	19	50.00%
C	Tree game	19	19	82.60%
D	Transformations	9	11	20.93%
E	LIS	0	3	00.00%
F	Padlock	19	19	67.86%
G	LR primes	6	8	22.22%
H	Hashed strings	3	6	15.00%

Table 1. Problem statistics

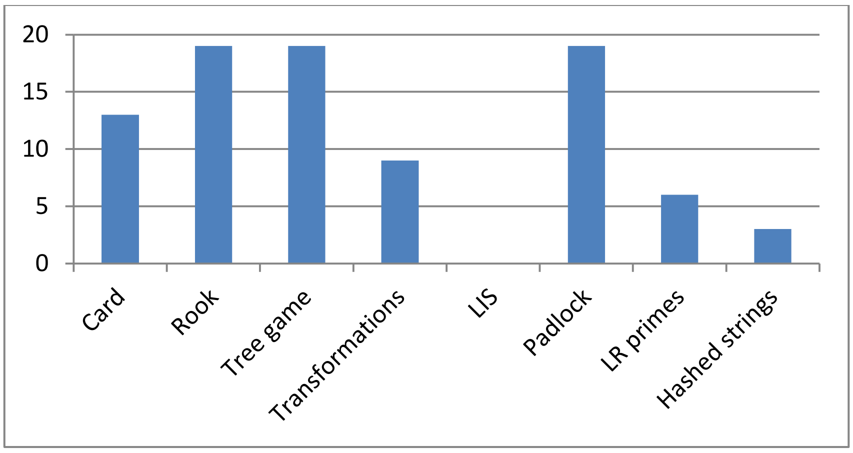
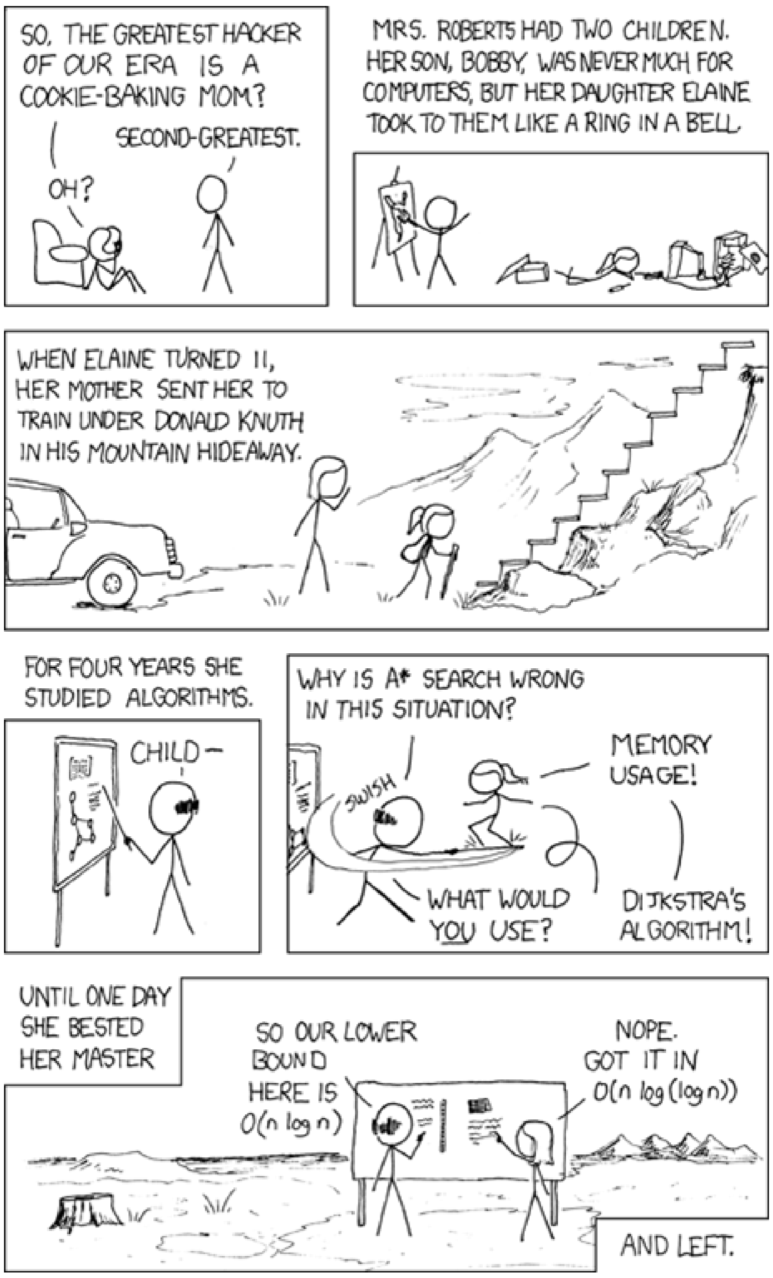


Chart 1. Number of correct solutions

ID	Task name	Elapsed time for the first accepted submission	Average elapsed time for accepted submission
A	Card	1:53	3:08
B	Rook	0:09	1:03
C	Tree game	0:20	1:11
D	Transformations	1:42	2:56
E	LIS	/	/
F	Padlock	0:38	1:40
G	LR primes	2:14	4:00
H	Hashed strings	2:56	3:36

Table 2. Time statistics

Problem set & Analysis from Finals


SO, THE GREATEST HACKER OF OUR ERA IS A COOKIE-BAKING MOM?
OH?
SECOND-GREATEST.
MRS. ROBERTS HAD TWO CHILDREN. HER SON, BOBBY, WAS NEVER MUCH FOR COMPUTERS, BUT HER DAUGHTER ELAINE TOOK TO THEM LIKE A RING IN A BELL.
WHEN ELAINE TURNED 11, HER MOTHER SENT HER TO TRAIN UNDER DONALD KNUTH IN HIS MOUNTAIN HIDEAWAY.
FOR FOUR YEARS SHE STUDIED ALGORITHMS.
CHILD—
WHY IS A* SEARCH WRONG IN THIS SITUATION?
SWISH
MEMORY USAGE!
WHAT WOULD YOU USE?
DIJKSTRA'S ALGORITHM!
UNTIL ONE DAY SHE BESTED HER MASTER
SO OUR LOWER BOUND HERE IS O(n log n)
NOPE. GOT IT IN O(n log(log n))
AND LEFT.
Taken from xkcd.com – A web comic of Romance, Sarcasm, Math, and Language

Problem A: Card

Author: ***Milan Vugdelija*** *Implementation and analysis:* ***Milan Vugdelija***

Statement:

Mike often needs to know if he could place a rectangular card of size $a \times b$ into an envelope of size $c \times d$. In order to be faster, Mike doesn't really try to put a card into an envelope, he just places a card on the table and then tries to cover it with an envelope. Of course, both the card and the envelope can be rotated, but they cannot be folded.

Now, Mike wants to be even faster. He decided to find the answers for all sizes of cards and envelopes he operates with. That's where you jump in. Your program should compute the answer for one particular case. The program should work the same way Mike does his tests, so in boundary cases the answer is "yes".

Input:

The first line contains four integers a, b c, and d delimited by a space. All values are less than 2×10^9.

Output:

The output contains only one string: "yes" or "no" (without quotes).

Example input:

```
2 3 3 4
```

Example output:

```
yes
```

Time and memory limit: 0.5s / 64 MB

Solution and analysis:

All we need to do is to distinguish between several cases. To simplify the analysis, let's first sort pairs (a, b) and (c, d) so that $a \leq b, c \leq d$.

Case 1: $\boldsymbol{a > c}$

In this case the answer is clearly **no**, since any y-projection of the card is bigger than c.

Case 2: $\boldsymbol{a \leq c,\ b \leq d}$

In this case card is easily covered with the envelope, for example by matching centers and aligning card and envelope axes, so the answer is **yes**;

Case 3: $\boldsymbol{a \leq c,\ b > d,\ a^2 + b^2 > c^2 + d^2}$

In this case the card diagonal $D_C = \sqrt{\boldsymbol{a^2 + b^2}}$ cannot be covered with the envelope, because the envelope diagonal $D_E = \sqrt{\boldsymbol{c}^2 + \boldsymbol{d}^2}$ is shorter than D_C. Therefore, the answer is **no**.

Case 4: $\boldsymbol{a \leq c,\ b > d,\ a^2 + b^2 \leq c^2 + d^2}$

This is the remaining case. Now we have $d < b < D_C \leq D_E$ and we need to try to put the envelope's diagonal over the card. Consider the circle centered at envelope center and having radius $\frac{D_C}{2}$. It intersects all four sides of the envelope and we need to check if the distance between the nearest two intersection points is bigger or equal to a. If so, the answer is **yes**, otherwise **no**.

Time complexity of this algorithm is constant - $O(1)$.

Test data:

Test corpus for this problem contains 10 test cases constructed with following methods

- several tests with different orders of side sizes
- tests with boundary conditions (for example card and envelope being of equal size)
- test in which the card tightly fits into the envelope diagonally
- test in which the card doesn't fit diagonally, but it would if it was just a bit smaller

Problem B: Rook

Author: ***Milan Vugdelija*** *Implementation and analysis:* ***Milan Vugdelija***

Statement:

There is a generalized chess board of size (n, n). A rook should move from square $(1, 1)$ to square (n, n). In every move, exactly one coordinate must increase by 1 or more. There are also m occupied squares on the board, so the rook cannot be placed on any of them and cannot jump over them. Squares $(1, 1)$ and (n, n) are not occupied.

In how many ways can the rook reach the square (n, n)?

Input:

The first line contains two positive integers n and m delimited by a space, $n \le 5000, m \le 100000$. In each of the next m lines there are two positive integers, x_i and y_i, $1 \le x_i, y_i \le n$, coordinates of i^{th} occupied square, $i = 1, 2, \dots m$.

Output:

The output contains number of different rook paths, as described above. If this number is 1 million or greater, you should only output its last 6 digits.

Example input:

```
4 2
3 3
4 1
```

Example output:

```
48
```

Time and memory limit: 2s / 64 MB

Solution and analysis:

Let's denote $w_{x,y}$ the number of ways in which the rook can reach the square (x, y). Then $w_{1,1} = 1$ and

$$w_{x,y} = \sum_{i=x_0}^{x-1} w_{i,y} + \sum_{j=y_0}^{y-1} w_{x,j}$$

where

$$x_0 = \begin{cases} \max\{i : 1 \le i < x, and\ \ square\ (i, y)\ is\ occupied\} & if\ there\ is\ such\ square, \\ 1 & otherwise \end{cases}$$

$$y_0 = \begin{cases} \max\{j : 1 \le j < y, and\ \ square\ (x, j)\ is\ occupied\} & if\ there\ is\ such\ square, \\ 1 & otherwise \end{cases}$$

Using the formula for each square directly gives an algorithm that works in $O(n^3)$ time, which is too slow for limitations given in the problem statement.

Introducing two new matrices,

$$a_{x,y} = \sum_{i=x_0}^{x-1} w_{i,y} \qquad\qquad b_{x,y} = \sum_{j=y_0}^{y-1} w_{x,j}$$

for each square (x, y) we can compute $a_{x,y}, b_{x,y}, w_{x,y}$ in $O(1)$ time, which gives the following $O(n^2)$ algorithm:

```
==================================================================================
w[1][1] = 1
for  i = 1..n
   for j = 1..n
      if (square (I, j) is occupied
          a[i][j] = 0;
          b[i][j] = 0;
          w[i][j] = 0;
      else
          a[i][j] = (a[i-1][j] + w[i-1][j]) mod 1000000;
          b[i][j] = (b[i][j-1] + w[i][j-1] ) mod 1000000;
          w[i][j] += (a[i][j] + b[i][j]) mod 1000000;
==================================================================================
```

We are assuming here that all elements with at least one zero coordinate are initialized to 0.

Complexity:

For this solution, there are a couple of variations regarding time and space complexity:

a) We can put info about occupied squares into a matrix (for example *w*), and use *a*, *b*, *w* as matrices. In that case both time and space complexity is $O(n^2)$.

b) We could also put info about occupied squares into a separate array of length m and sort it in order in which the squares are being visited. Also, instead of matrices a, b, w, it is enough to use the last two rows of each of them. That gives us time complexity $O(m \log m + n^2)$, and space complexity $O(m + n)$.

Test data:

Test cases should include:

- An example where it is not possible to move by the rules and reach the square (n, n);
- A big example with a large table and lots of occupied squares (up to the limit).

Problem C: Tree game

Author: ***Stevan Jončić*** *Implementation and analysis:* ***Stevan Jončić***

Statement:

You are playing a simple game. You are given an undirected connected graph which does not have cycles. There is also one coin with is in the beginning located at vertex x. One step consists of moving the coin from the vertex at which it is currently located to any adjacent vertex (two vertexes are adjacent if there is an edge connecting them). Every edge has an associated number of points you gain if you move the coin from one of its vertexes to another. Your task is to calculate the maximal number of points you can gain in k steps. You can move the coin along some edges more than once.

Input:

The first line contains number n, which is the number of vertexes of the tree (number of vertexes $2 \leq n \leq 100000$). The following $n-1$ lines contain information for $n-1$ edges of the tree. Each of the following $n-1$ lines has three numbers (i-th of these lines describes i-th edge) – the first two numbers are vertexes connected by the edge and the third number is the number of points that you gain if you move the coin along that edge. The number of points associated with an edge is less or equal to 1000. The vertexes are labeled with numbers from 1 to n.

The next line contains the number k, $1 \leq k \leq 100000$.

The last line contains the vertex x, vertex at which the coin is located in the beginning.

Output:

You should output one number which is the maximal number of points you can gain in k steps with the coin located in the beginning at vertex x.

Example input:

```
6
1 2 3
4 3 5
4 1 2
3 6 6
5 1 9
3
4
```

Example output:

```
20
```

Time and memory limit: 1s / 64 MB

Solution and analysis:

This is a graph problem. On first sight, it looks like this problem requires the standard dynamic programming approach for trees - bottom-up from leaves to the root. But if we play a little bit with this problem, we will see that the greedy approach will find an optimal path.

Assume that we use following edges in k steps path: $p = e_1, e_2, \dots, e_k$. Edges can be used more than once so e_i and e_j can be the same edge for some $1 \leq i < j \leq k$. If there is an edge e_m $(1 \leq m < k)$ that has more points than every edge in the path p used after e_m, then the path p can't be optimal. Namely, in that case we can use the first part $e_1, e_2, \dots, e_m - 1$ of the path p and after that we just use edge e_m for the remaining $n - i + 1$ steps. This way we will get more points than in the original path.

Because of this, in the optimal path the edge with the maximal number of points among the edges that constitute the optimal path must be the edge which was used last (or in a last couple of steps) in the optimal path. This is the main idea for the algorithm.

Let's say that in the optimal path the edge e is used last in a couple of steps. We can see that the number of edges we used prior to using edge e should be as small as possible; otherwise the path would not be optimal because we can make a path which uses less edges prior to using edge e, and this path will then get us more points.

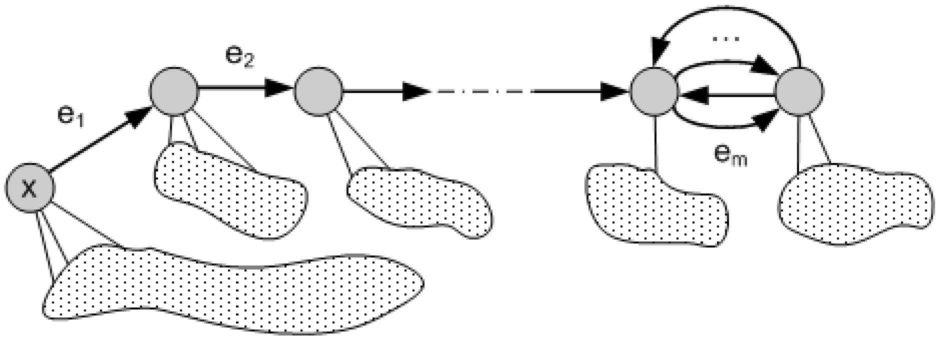


Figure 1. Optimal path

The solution consists of the following: for every edge of the tree we try to go to that edge using the minimal possible number of edges and then use that edge for every available step left and we choose among those paths the path with the maximal number of points. Of course, we try this for every edge to which the minimal number of edges used is less than the available number of steps. Because this is a tree, we can accomplish all this with one traversal using some standard graph traversal algorithms – DFS or BFS.

Time and memory complexity of this solution are both $O(n)$.

Problem D: Transformations

Author: ***Stevan Jončić***

Implementation and analysis: ***Stevan Jončić***
Andreja Ilić

Statement:

You are given n different transformations of integers $1, 2, \dots, n$, one for each of these n numbers. Using the first transformation you can transform number 1 to some group of numbers, using the second transformation you can transform number 2 to some other group of numbers etc. Numbers that can be derived using given transformations are also integers between 1 and n.

If you have a group of elements, which are numbers from 1 through n (there can be multiple instances of the same number in the group), in one step can you can transform any element of the group to new elements that are produced using the transformation of the selected element. You start with a group which has only one element, which is a number between 1 and n, and you can choose which number is the starting element of the group. Your goal is to have after s steps a group with as much elements as possible.

Input:

The first line contains one positive integer n, $1 \leq n \leq 1000$.

The following n lines contain information for transformations of numbers from 1 to n. Each of the following n lines consists of the following integers (i-th of these lines describes transformation of number i) – the first number, denote it with e_i ($1 \leq e_i \leq 30$) is the number of elements to which number i is transformed and the following e_i numbers are the numbers to which number i is transformed.

The last line contains number s ($1 \leq s \leq 50$) which is number of available steps.

Output:

You should output one number which is the maximal number of elements your group can possibly have after s steps.

Example input:

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

Example output:

```
10
```

Explanation:

There are 4 numbers. The transformations are:

$$1 \rightarrow 1\ 1\ 4$$

$$2 \rightarrow 4\ 4\ 1\ 3\ 1$$

$$3 \rightarrow 4$$

$$4 \rightarrow 2\ 1$$

The optimal solution is choosing the initial element of the group to be 2, then after transforming it the group will have elements 1 1 3 4 4, after that one instance of number 4 is transformed and the group will have elements 1 1 1 2 3 4. Finally, the number 2 is transformed and the group has 10 elements after 3 steps.

Time and memory limit: 1s / 64 MB

Solution and analysis:

We can solve the task using dynamic programming. This is a very nice problem, because we have some kind of two-step dynamic programming where these steps communicate with each other.

Firstly, let us introduce labels that we are going to use:

- $k \rightarrow \big(t[k][1], t[k][2], \dots, t[k]\big[num[k]\big]\big) \equiv T[k]$, $k \in [1, n]$, for the transformations. Number k can be transformed in the above group, where $e[k]$ represents cardinality of this list.
- $F(numStep, (a_1, a_2, \dots a_m))$ – maximal number of elements that can be obtained starting from the group $(a_1, a_2, \dots a_m)$ and performing $numStep$ transformations in some order.
- s - number of steps (transformations)

The final solution can be computed as:

$$solution = max\{F(s,(1)), F(s,(2)), \dots, F(s,(n))\}$$

The main observation for this problem is following: when we perform transformation $a_k \rightarrow T[a_k]$ on the group $(a_1, a_2, \dots, a_m)$ we obtain a new group

$$newGroup = (a_1, a_2, .., a_{k-1}, a_{k+1}, \dots, a_m) + \big(t[k][1], t[k][2], \dots, t[k]\big[e[k]num[k]\big]\big) = A + T[k]$$

Pay attention that the plus sign in the above formula is not a union. From here, we can look at these two groups independently. The only question is to find how many transformations to “give” to each group - partition of the number of steps. Without loss of generality, we can calculate the value $F(numStep, (a_1, a_2, \dots a_m))$ by checking all possible number of steps for transformation $a_m \rightarrow T[a_m]$. Formally:

$$F\big(numStep, (a_1, a_2, \dots a_m)\big) = \max_{k \in [0, numStep]} F(numStep - k, (a_1, .., a_{m-1}) + F(k, a_m)\ \}$$

We can think of these transformations and groups as some kind of tree of deep s. Basically, we start from any group with one element – which is going to represent a root of this tree. We want to find a leaf which holds the set with maximal cardinality.

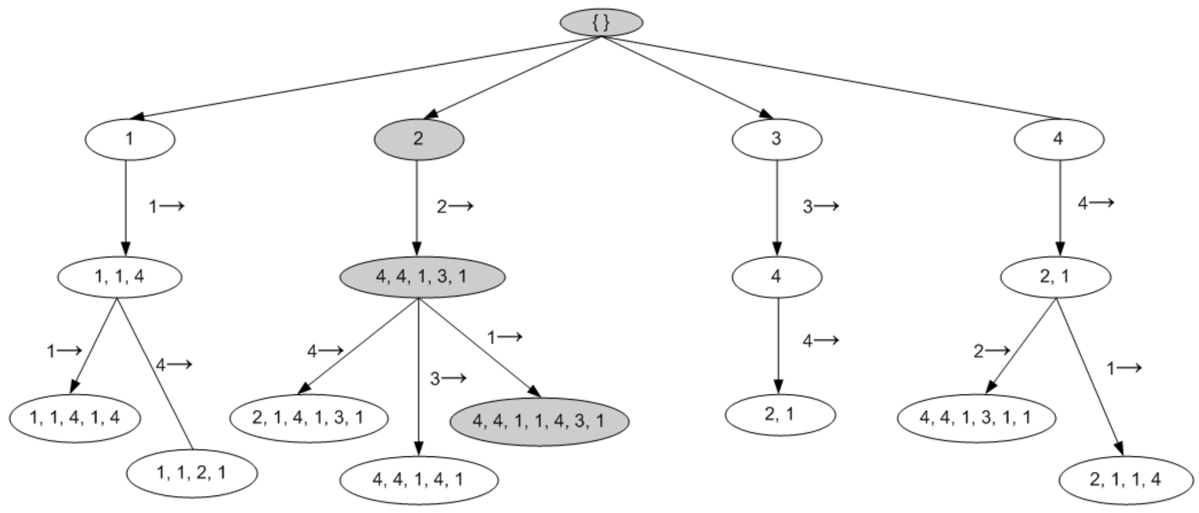


Figure 1. Example of the tree mentioned in the problem analyses from example in the problem statement with changed condition $s = 3$

Implementation:

This can be implemented in many ways. We will describe one of them. First let us define matrix d as

$$d[numStep, x] = \text{maximal number of elements that can be obtained starting from only one element } x \text{ in } numStep \text{ steps}$$

When we are computing some particular element $d[k, x]$, we are going to use:

$$q[numStep, v] = \text{maximal number of elements that can be obtained starting from the group } (t[x][1], \dots, t[x][v]) \text{ in } numStep \text{ steps}$$

From here we have that $d[numStep, x] = q[numStep - 1, e[x]]$ (here we have -1 because we used one transformation $x \to T[x]$). We can play with elements of the matrix q with following relation:

$$q[numStep, 1] = e[x]$$

$$q[numStep, v] = max_{k\in[0,numStep]}\{q[k, v-1] + d[numStep - k, t[x][v]]\}, \text{ for } v \in [2, e[x]]$$

The complexity of this solution is $O(n \cdot s^2 \cdot m)$, where m represents maximal group cardinality in the given transformations.

Problem E & I: LIS

Author: ***Andreja Ilić*** *Implementation and analysis:* ***Andreja Ilić***

This document contains the problem statements for problems E and I. You will see that the only difference is in one constraint. These are separate problems and will be tested on different test cases.

Scientific committee only has a solution for problem E.

Statement:

You are given an integer sequence a of length n and an integer w, $1 \le w \le n$. Let us denote with L_k the length of the longest increasing subsequence (LLIS) for subarray: $a_k, a_{k+1}, \dots, a_{k+w-1}$. You have to write a program that computes values L_k for every k, $1 \le k \le n - w + 1$.

- **Problem E**: Assume that the sum of values L_k does not exceed $3 \cdot n\sqrt{n}$.
- **Problem I**: There are no constraints for the sum of values L_k.

The longest increasing subsequence of a given sequence a is the subsequence of strictly increasing elements containing the largest number of elements. Elements of the subsequence do not need to be consecutive.

Input:

The first line contains two positive integers n and w ($1 \le n \le 100.000$ and $1 \le w \le n$), where n is the number of elements in the given array and w is the width of subarray that have to be examined. Next line contains n integers, separated with one space, which represents the elements of array a.

The elements are in range $[0, 2 \cdot 10^9]$.

Output:

The output contains $n - w + 1$ numbers, one per line. The number in the k-th line is the length of the longest increasing subsequence for $a_k, a_{k+1}, \dots, a_{k+w-1}$.

Example input:

```
6 4
1 4 2 5 6 7
```

Example output:

```
3
3
4
```

Explanation:

For this example we have three subsequences of width 4 in given array a:

- $(1, 4, 2, 5)$, where LLIS is equal to 3; one possible LIS is $(1,2,5)$
- $(4, 2, 5, 6)$, where LLIS is equal to 3; one possible LIS is $(4,5,6)$

- $(2, 5, 6, 7)$, where LLIS is equal to 4; LIS is the whole subsequence

Time and memory limit: 2s / 64 MB

Implementation and analysis:

This problem considers finding the length of the longest increasing subsequence in a sliding window (of width w), over a given sequence a. In the problem statement it is noted that the sum of lengths does not exceed $n\sqrt{n}$. This is a very interesting fact and it might be confusing. Here we are going to present an output-sensitive data structure that solves this problem with time complexity $O(n \log n + OUPUT)$ or in our case $O(n \log n + n\sqrt{n})$.

Within this framework, several related questions can be posed regarding this problem, each with potentially different time complexity.

- *Local Max Value* - For each window report the length of the longest increasing subsequence in that window.
- *Local Max Sequence* - Explicitly list a longest increasing subsequence for each window.
- *Global Max Sequence* - Find the window with the longest increasing subsequence among all windows.

Here we deal with the *Local Max Value*. This algorithm solves the other two versions of the problem described above. Its optimality in our case is an open question and left for contestants to improve it ☺

A naïve approach is to consider finding LIS for every window separately. The standard dynamic programming algorithm for finding LIS has time complexity of $O(n^2)$, which will lead to complexity of $O(n \cdot w^2)$ for our problem. This approach can be sped up with algorithms which date back to Robinson [1] and Schensted [2] with a generalization due to Knuth [3]. These algorithms have time complexity $O(n \log n)$, which is optimal in the comparison model. Hunt and Szmanski [4] gave an algorithm with time complexity $O(n \ \log \log n)$ using the *van Emde Boas data structure* [5]. In any case, this naïve approach has time complexity $O(n \cdot w \log \log w)$ in the best case.

Without loss of generality we can assume that a given array a is a permutation of the set $\{1, 2, \dots, n\}$ (if not we can simply sort the array and rename the numbers in it with corresponding index). As we have seen in the previous paragraph, we have to find some way to use the LIS (or some other information) from the previous window when examining the current one. For this purpose, we will use *Young tableaux* or the *Robinson–Schensted–Knuth algorithm*. We will not explain these algorithms in detail, because only a part of them will be needed here.

Above we have stated that the length of LIS for a given array can be found in $O(n \log n)$ time. How can we do this? Let us introduce a new list d. Initially this list will be empty. We will insert elements from array a one at a time into the list d. When inserting number $value$ into d we have two cases:

a) $value$ is greater than all elements from the list d - In this case we add $value$ to the end of list
b) $value$ is not greater than all elements from the list d - In this case there exists an element that is greater than $value$. Let us denote with t the first one from the left. Remove the element t from the list d and put $value$ in its place.

With this algorithm list d will be monotonically increasing. It can be shown (how?) that the length of list d is the length of the longest increasing subsequence. It should be noted that list d is not a LIS for array a, because it may not be a subsequence (see example on Figure 1). The main idea behind this method is that the element $d[k]$ is the smallest element from array a for which there exists an increasing subsequence in a of length k ending with that element. We will call d the principal row of array a and denote it with $R(a)$.

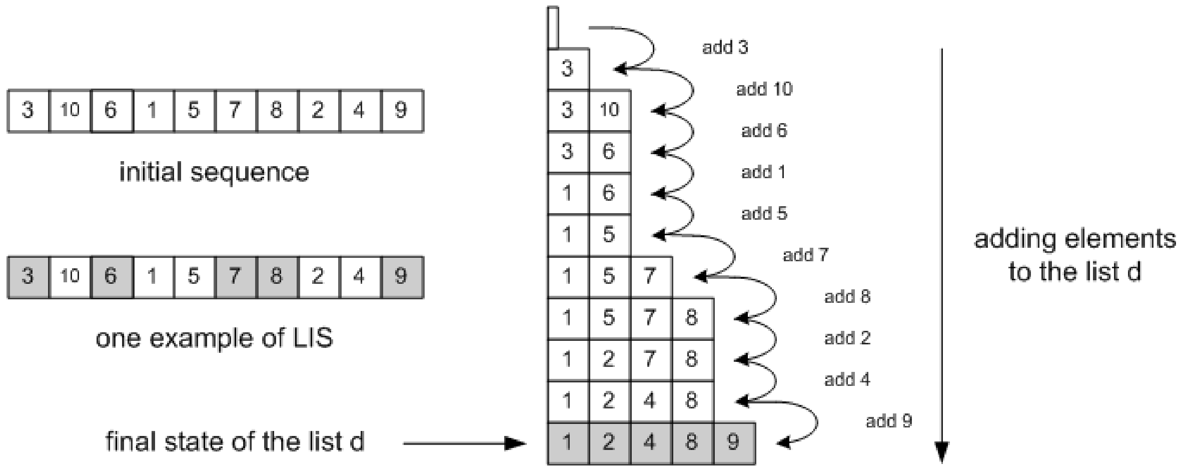


Figure 1. Example of algorithm for finding the LIS in array $\{3,10,6,1,5,7,8,2,4,9\}$.

In order to deal with the problem, we will consider a slightly more general question. We want to define some kind of structure that will maintain information about the LIS of a sequence in such a way that it supports the following operations:

- adding a new element at the end of a sequence
- removing the first element from a sequence
- querying the data structure for the LLIS

For this purpose we are going to store the principal row for every suffix of the current sequence. If we denote with a^k the suffix $a_k a_{k+1} \dots a_n$, our structure will maintain $R(a^1), R(a^2), \dots, R(a^n)$ (note that in our case this sequence has length w). This collection of principal rows is called a row tower.

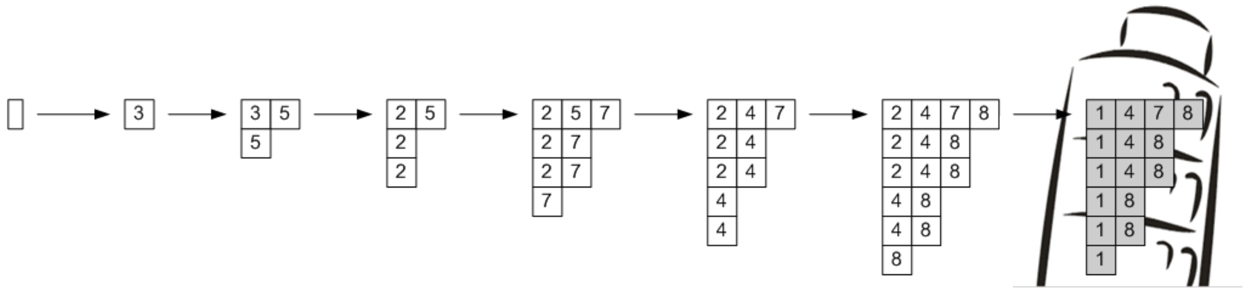


Figure 2. Example of a row tower for the array $(3,5,2,7,4,8,1)$ and how it is generated.

Removing the first element from a sequence can be implemented easily – delete the first principal row $R(a^1)$. The length of the first principal row is the length of LIS. Adding a new element corresponds to inserting it in every row and adding a new row containing only this element. A naïve implementation of this method will also lead to time complexity of $O(n \cdot w \log w)$. If we want to speed this up, we must store this tower in some compressed way.

Something that we can notice in Figure 2 is that $R(a^k)$ is either the same as $R(a^{k-1})$ or can be obtained from it by deleting a single element. This can be proven by induction (how?). From this we can state a generalization:

Lemma Let sequence A be a suffix of sequence B. Then $R(A)$ is a subsequence of $R(B)$ and

$$|R(B)| - |R(A)| \leq |B| - |A|.$$

Because of this nice property, we can store the whole row tower in the following way:

- $R = R(a) = R(a^1)$ – the principal row of whole sequence
- Drop out sequence d with the length $|R(A)|$. Element $d[index]$ represents the suffix at which the element $R(a)[index]$ drops out of the principal row.

For our example of Figure 2 we have $d = (6,3,1,5)$. With these two sequences we can reconstruct the whole tower. The main problem here is to see how we can efficiently update this representation of the row tower. The expire operation simply subtracts one from each element of d and deletes the element with expiry time 0 (if there is one) from R. The add operation for an element b requires that b should bump an element out of each row of the row tower (unless it is appended to all of them). Since the rows form an inclusion chain, if b bumps a certain element s out of a row, then it bumps the element s out of all further rows to which s belongs. In other words, the drop out time for s changes to the index of the first row from which it is bumped out by b. Now consider the next row of the tower (if one exists) after s has dropped out. In this row there may or may not be elements larger than s. If there are such elements then b bumps out the smallest one. If not, then b is appended to the end of this and all subsequent rows. We can find a sequence of indices $i_1 < i_2 < \cdots < i_k$ for the sequence d such that:

- i_1 is the least index of an element in the principal row R which is larger than b
- i_{x+1} is the least index larger than i_x for which $d(i_{x+1}) > d(i_x)$ (the element is larger than the prior one and it is still in the current principal row).

Now we can simply update the drop out sequence d according to:

- $d(i_1) = w$
- $d(i_{x+1}) = d(i_x)$, for $x \in [1, k-1]$

Implementation of this algorithm is pretty straightforward and we will leave it to the reader.

Complexity

In this way we managed to implement operations for adding and removing one element in linear time of the LLIS problem (querying is still in constant time). In the problem statement we denoted the length of LIS in k-th window with L_k. From this the overall time complexity of our algorithm is $O(\sum L_k)$. The described algorithm computes the lengths of LIS in the sliding window in total time of

$$O(n \log n + \sum L_k) = O(n \log n + OUTPUT) = O(n \log n + n\sqrt{n})$$

Test data

The test corpus for this problem consists of 15 test cases.

Test cases were generated with a couple of algorithms which (except those for special cases) were based on random sequences and following theorem [9]:

Theorem Let π_n be an uniform random permutation of set the $\{1,2,3,\dots,n\}$ and L_n an integer-valued random variable $L_n = LLIS(\pi_n)$. As $n \to \infty$ we have

$$E[L_n] \approx 2\sqrt{n} \quad \text{and} \quad \sigma[L_n] = o(\sqrt{n}\,)$$

A short description of test cases is given in Table 1.

ID	n	w	min LLIS	max LLIS	solution sum	Description
01	10	5	2	3	16	By hand
02	100	10	3	7	395	Random
03	1000	100	12	21	15.333	Random
04	1000	900	54	57	5.675	Random
05	10000	100	70	91	802.603	Increasing sequence
06	99000	1000	2	825	39.315.222	"Saw" sequence
07	100000	50000	427	446	21.829.042	"Saw" sequence
08	100000	90000	587	597	5.908.135	Random
09	100000	100	12	25	1.671.330	Random
10	100000	1	1	1	100.000	Special case - Random
11	1	1	1	1	1	By hand
12	99999	99999	618	618	618	Special case - Random
13	99888	65432	1	1	34.457	Decreasing sequence
14	99999	1000	23	61	4.159.326	Random, $P_d = 95\%$
15	99999	77777	3024	3101	67.945.385	Random, $P_d = 95\%$

Table 1. Description of the test data

References

[1] G. de B. Robinson, *On representations of the symmetric group*, Am. J. Math. 60 (1938) 745–760.

[2] C. Schensted, *Longest increasing and decreasing subsequences*, Can. J. Math. 13 (1961) 179–191.

[3] D. E. Knuth, *Permutations, matrices, and generalized Young tableaux*, Pacific J. Math. 34 (1970) 709–727.

[4] J. Hunt, T. Szymanski, *A fast algorithm for computing longest common subsequences*, Comm. ACM 20 (1977) 350–353.

[5] P. van Emde Boas, R. Kaas, E. Zijlstra, *Design and implementation of an efficient priority queue*, Math. Systems Theory 10 (2) (1976/77) 99–127.

[6] M. H. Albert at al., *Longest increasing subsequences in sliding windows,* Theor. Comp. Sci. 321 (2004) 405 – 414.

[7] D. E. Knuth, *The Art of Computer Programming, Vol. 3, Sorting and Searching*, Addison–Wesley, Reading, Mass, 1973.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to algorithms*, MIT Press (2009)

Problem F: Padlock

Authors: ***Andreja Ilić*** *Implementation and analysis:* ***Miloš Lazarević***
Dražen Žarić

Statement:

You are stuck in a room with n doors. On every door there is a padlock with a 10-digit rolling lock combination. You can roll any digit either up or down, where rolling up at digit 9 will make the digit 0, and rolling down at digit 0 will make the digit 9. The padlock will be open when the combination is matched with the key for that padlock. The goal is to open all doors with the minimal number of rolling operations.

Initially all padlocks are set to 0000000000. Doors can be opened in any order. Besides rolling digits there is one very cool button on the padlocks. This button can turn the digits on padlock to the same combination as a different padlock that is already open (you cannot jump to a combination of the padlock for some door that is not open yet). This transformation does not count as a rolling operation.

Input:

The first line contains one positive integer n $(1 \leq n \leq 1000)$, where n is the number of doors. The next n lines contain 10-digit integers (some of them can have leading zeros), which represent the keys for padlocks.

Output:

The output should contain only one integer – minimal number of rolling necessary to open all doors.

Example input:

```
2
0000000003
0000000001
```

Example output:

```
3
```

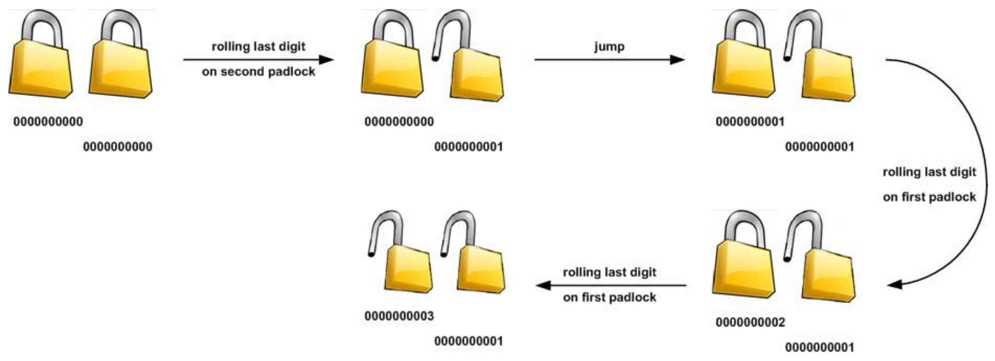


Figure. Explanation of the given example

Time and memory limit: 1s / 64 MB

Solution and analysis:

We will first give the algorithm description, and then prove its correctness. We can use a simple greedy strategy:

1. Find a lock that needs the least number of rollings, from the initial state, to open. Add that number to the overall cost, and put that lock into the set of open locks.
2. Repeat until all locks are open:
   - Among locks that are still closed, find the one that requires the least number of rollings to unlock, considering we can set it to state of any of the locks already open using zero rollings, or we can roll the numbers from the initial state.
   - Update the overall cost, and put the minimal lock into set of open locks.

To show that this approach does indeed yield the minimal overall number of rollings, we can consider a graph whose vertices are locks, and weight of each edge (u, v) is equal to number of rollings needed to open lock v once it is set to the combination of lock u. We can extend this graph with a lock z, whose key is all zeros, so that weights of edges (z, i) represent numbers of rollings necessary to open lock i from its initial state. We also notice that weights of edges (u, v) and (v, u) must be equal, thus we have a complete undirected graph.

When opening lock v, we can either set it to a key of a previously open lock u and then roll the numbers to get the right key, or roll the numbers from the initial position to v's key. So unlocking v increases the overall cost either by weight of edge (z, v)*)* or by weight of edge (u, v). If we consider the subgraph with only these *used* edges, we see that it is actually a spanning tree of the original graph. So in order to find the least number of rollings necessary to open all locks, we need to find a **minimum spanning tree** of our graph.

The proposed greedy approach is actually **Prim's algorithm** for finding minimum spanning trees of graphs and is easily implemented to run in $O(n^2)$ time. We can also precalculate numbers of rollings between all pairs of locks, and store the graph in matrix form, which requires additional $O(n^2)$ time and memory.

Problem G: LR primes

Author: ***Andreja Ilić*** *Implementation and analysis:* ***Milan Novaković***
Andreja Ilić

Statement:

A number $a = \overline{c_n c_{n-1} \dots c_2 c_1}$ is called L prime if its every non-empty suffix is a prime number and all its digits are different from zero. In other words, numbers $\overline{c_1}, \overline{c_2 c_1}, \dots, \overline{c_{n-1} \dots c_2 c_1}$ and $\overline{c_n c_{n-1} \dots c_2 c_1}$ must be primes. For example the number 113 is L prime number.

A number $a = \overline{c_n c_{n-1} \dots c_2 c_1}$ is called R prime if its every non-empty prefix is a prime number. In other words, numbers $\overline{c_n}, \overline{c_n c_{n-1}}, \dots, \overline{c_n c_{n-1} \dots c_2}$ and $\overline{c_n c_{n-1} \dots c_2 c_1}$ must be primes. For example number 311 is R prime.

You are given an integer segment $[a, b]$. How many integers from this segment are L or R prime numbers (including numbers a and b)?

Input:

The first line contains two positive integers a and b ($1 \le a \le b \le 10^{18}$), which represent the given segment.

Output:

The output contains only one integer – the number of integers from given segment that are L or R primes.

Example input:

```
10 30
```

Example output:

```
4
```

Explanation

From the segment $[10, 30]$ L primes are: $13, 17, 23$; R primes are $23, 29$. Number 23 is both L and R prime, so we are going to count it only once.

Time and memory limit: 0.5s / 64 MB

Problem analysis:

L and R primes are also known as **left-truncated** and **right-truncated primes**. Codes of their sequences in the *On-Line Encyclopedia of Integer Sequences* are A024785 and A024770. We found them interesting for a programming problem because of two facts:

- they are finite
- they have some kind of recursive property

We need to find a method for generating consecutive right and left primes. Here we are going to explain the algorithm for right primes. The same algorithm, with small modifications because of the special digit 0, can be used for the left primes.

As we mentioned, these numbers have some kind of recursive structure: every right prime number having at least two digits is an extension of another right prime number (i.e. the least significant digit is added). This is the main fact on which we are going to base our iterative algorithm.

Let Q_R be an empty queue, which will store the right primes. We start by inserting the one-digit right primes (just primes). Then in every step we extract the first element s from the queue and check if any of the numbers $10 \cdot s + k$, $k \in [1,3,7,9]$ is prime. We excluded the digits $\{0,2,4,5,6,8\}$, because if the last digit is from this set, the new number will not be prime. If this number is also a right prime - put it at the end of the queue.

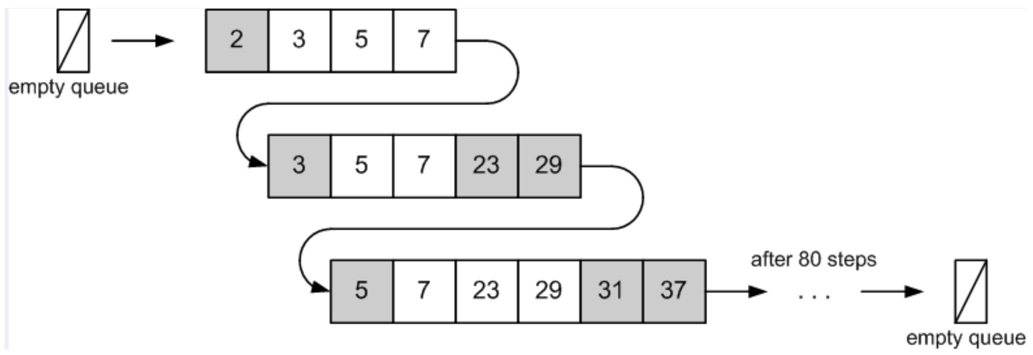


Figure 1. The queue states in the right prime construction process.

Complexity and implementation:

An interesting feature that we need to address for this algorithm is its time complexity. The complexity is $O(k\sqrt{m})$, where k is the number of the right prime numbers and m is the greatest among them (the same thing holds for the left primes). This fact is left for contestants to find out. Namely, these are finite sequences and after running this algorithm it appears that the algorithm terminates with an empty queue. There are only 83 right prime numbers and only 4260 left prime numbers.

The largest of them are $73.939.133$ and $357.686.312.646.216.567.629.137$, respectively.

Another interesting fact is that if zeros are permitted, the sequence of left primes is infinite.

Because of this fact, the described algorithm with a reasonable implementation works very fast. For this we must use some other technique for the primality testing. In our case the Fermat test will do the work. Of course some other algorithm, like the Miller-Rabin test, would also work. Here we will briefly describe the Fermat test.

Firstly, recall Fermat's little theorem: if p is a prime number and a is an integer relatively prime to p, then

$$a^{p-1} \equiv_p 1$$

Experimentation shows that this typically fails when p is composite. This is the fact which is going to be the core of our test. Complexity of this algorithm is $O(k \cdot \log^2 n)$, where k is the number of times we test a random number a with above theorem.

```
=====================================================================================
              Function: Fermat's primality test
              Input:  n – a value to test for primality
              Output: false if n is composite
                      true if n is probably prime
 ------------------------------------------------------------------------------------

              repeat k times
                 pick random integer a from set {2,3,…,n-1}
                 d = gcd(a,n);
                 if (d != 1)
                   return false;
                 tmp = a^(n-1) mod n;
                 if (tmp != 1)
                    return false;

              return true;
=====================================================================================
```

Pseudo code for the second algorithm

Another option is to hardcode all left and right primes in the code. Such solution works in linear time. Here we have to pay attention to the size of the file. If we hardcode this in a relatively smart way, we will get the source file of the size ~ 60KB, and 64KB is the maximum allowed size for source files on the finals.

Problem H: Hashed strings

Author: ***Dimitrije Filipović*** *Implementation and analysis:* ***Andrija Jovanović*** ***Dimitrije Filipović***

Statement:

You are an evil hacker and your current evil mission is to impersonate your target by sending messages that look like they came from them but that are actually from you. You have worked out the entire operation except for one small detail: every string that your target sends is followed by a 32-bit hash value, which is used for error checking. You know the algorithm, and it goes like this:

The strings are composed of lowercase letters of the English alphabet, and every letter corresponds to a unique 16-bit code. All 32 bits of the hash value are initialized to zero. The hash is then calculated by passing through every character of the string in order and performing the following steps:

1. Do a binary left rotation of the entire hash value (by one place)
2. Take the code for the character and the least significant 16 bits of the hash and do a binary XOR of these two values
3. Write the result from the previous step to the least significant 16 bits of the hash

Unfortunately, to implement the algorithm you need to know the 16-bit codes for letters of the alphabet, and those codes are secret. Not all is lost though! You have already intercepted many pairs of strings with their hash values. Now all you have to do is find some way to use that information to crack the codes.

Input:

The first line contains one positive integer n $(1 \leq n \leq 400)$, the number of strings. Each of the following n lines contains one string consisting exclusively of letters 'a'-'z' and one integer in the range $[0,\ 2^{32}-1]$. Writing this integer in 32-bit binary gives the hash value of the string. Each string is at most 100 characters long.

Output:

The first line of output should be one of three words: "IMPOSSIBLE", "UNIQUE" or "MULTIPLE" (without quotes), if there are respectively no solutions, exactly one solution and more than one solution. If the first line is "IMPOSSIBLE" or "MULTIPLE", nothing else should be written to output. If the first line is "UNIQUE", each following line of output should contain exactly one letter and one number, separated by a space. Every number is in the range $[0,\ 2^{16}-1]$, and when written in 16-bit binary represents the code for the letter. There should be as many lines as there are different letters that appear in input. The lines should be sorted alphabetically by letter.

Example input:

```
2
a 4
ab 12
```

Example output:

```
UNIQUE
a 4
b 4
```

Time and memory limit: 3s / 64 MB

Implementation and analysis:

This task is based on a problem that the author actually had to solve for his real-life job (it didn't involve any hackers though – that part is made up ☺), and we thought it was interesting enough to be used for competitive purposes.

We have n strings. Let's denote them with $s_i = c_{i,l_i-1}c_{i,k_i-2} \dots c_{i,0}$ ($i \in \{0,1 \dots n-1\}$;l_i is the length of s_i). (We'll use indexes that increase right to left throughout the text, so don't say you weren't warned.) A 32-bit integer corresponds to each string: $h_i = b_{i,31}b_{i,30} \dots b_{i,0}$ ($b_{i,j} \in \{0,1\}$). And finally, each character c corresponds to a code, which is the 16-bit integer $x_c = x_{c,15}x_{c,14} \dots x_{c,0}$.

Let's observe the rightmost (index 0) bit of the hash value h_i. How is it calculated? Obviously, the rightmost bit of the rightmost character of string s_i (which we have denoted with $x_{c_{i,0},0}$) can change it in the last step. But before that, the bit $x_{c_{i,17},15}$ was initially added on index 15 and then made half a circle to get to our bit $b_{i,0}$. And before that, bits $x_{c_{i,18},14}$, $x_{c_{i,19},13}$,… $x_{c_{i,32},0}$ also ended up turning around and contributing to $b_{i,0}$. And another half-circle before that, bit $x_{c_{i,49},15}$, and so on. Obviously, this goes on until we run out of characters in s_i. The formula is

$$b_{i,0} = \sum_{0 \le j < l_i, 0 \le k < 16}^{32|(j+k)} x_{c_{i,j},k}$$

where the sum is an XOR sum (or, mathematically speaking, everything happens in $\mathbb{Z}_2$). Now let's try to generalize this observation. We have the q-th bit (from the right, zero-based – as above) of hash value h_i. Which bits of the original codes are important for this bit? The same line of thinking as in the previous paragraph leaves us with the formula

$$b_{i,q} = \sum_{0 \le j < l_i, 0 \le k < 16}^{j+k \equiv q \pmod{32}} x_{c_{i,j},k}$$

This means that the problem reduces to a **system of linear equations**. We have one equation per every bit of every hash sum, which is a total of $32n$. The number of variables is 16 times the number of letters that appear in the input.

Solving systems of linear equations is a well-known problem, and here it is made even easier by the fact that we are working in $\mathbb{Z}_2$ so the only values are 0 and 1 and there are no problems with precision. For example, we can solve the system in time $O(u^2 \cdot v)$, where u is the number of variables and v the number of equations, by the standard Gaussian method of eliminating the variables one by one. Of course, this algorithm is able to determine whether the system has a solution, whether it is unique and, if it is, to find it. After this, assembling the solution bit-by-bit into codes for every letter and sorting them alphabetically should present no trouble at all.

Complexity

It is easy to see that the most expensive part of our algorithm is solving the system of equations, so the time complexity will be $O(u^2 \cdot v)$ per above. We have $u = 32n$ and, since we have a finite alphabet of known size, we could say that v is a constant but that would be slightly disingenuous as this constant is quite large. If w is the number of letters that appear in the input, we have $v = 16w$, and finally our complexity is $O(2^{13} \cdot w^2 \cdot n)$. It is interesting to note that time complexity is independent from the length of the strings.

We need $O(u \cdot v) = O(2^9 \cdot w \cdot n)$ space to store the equations, which gives us the memory complexity of this solution.

Test data:

ID	Description
01	Easy test (example from the problem statement)
02	1 string with 1 letter with valid hash value (result code for the letter is equal to the hash value)
03	1 string - 31 times one letter. Code invalid/valid (IMPOSSIBLE/UNIQUE)
04	
05	1 string - 32 times one letter (each bit from the code influences each bit of the hash value, so all bits of the hash value need to be equal) (IMPOSSIBLE/MULTIPLE)
06	
07	1 string - 33 times one letter. Code valid (UNIQUE)\
08	1 string - 64 times one letter (hash value doesn't depend on the code of the letter/hash value is always 0) (IMPOSSIBLE/MULTIPLE)
09	
10	Invalid hash value (larger than it could be calculated with given string) (IMPOSSIBLE)
11	
12	Contradiction (last bit of the code for a letter should be both 0 and 1)
13	Large test with a small number of letters
14	Less strings than the number of used letters but still UNIQUE solution
15	Large strings but not enough equations to calculate UNIQUE solution (MULTIPLE)
16	Large test. One bit changed so IMPOSSIBLE.
17	Large tests to calculate UNIQUE solution
18	
19	

Qualifications

This is the fourth birthday of Bubble Cup and we are very pleased to see that the number of participating teams keeps increasing. This year 71 teams managed to solve at least one problem from the qualification rounds. We are especially proud of the fact that the competition can now truly be called regional, with more and more teams from countries such as Croatia, Bulgaria, Romania and Macedonia not only participating but also achieving notable results.

The qualifications were split into two rounds, with ten problems in each round and 25 days for the contestants to solve them. The first round lasted throughout April, and teams earned one point for each successfully solved problem. The second round was in May, and problems in this round were worth two points each. The problems for both rounds were chosen from the publicly available archives at the Timus Online Judge site.

The qualification rounds, especially the second one, were a little bit advanced. Some of the problems (like Expert Flea) were pretty unusual for competition problems. Namely, these problems required spending a good couple of days thinking, or reading and analyzing scientific papers on the subject. We are very delighted to see that competitors managed to deal with this type of tasks.

Unlike last year, every problem from qualifications has been solved by at least one team. The nineteen teams with the highest number of points qualified for the finals. One of these teams was not eligible for awards, but they were nevertheless allowed to compete.

Num	Problem name	ID	Accepted solutions
01	Triathlon	1062	36
02	Archer's Travel	1459	30
03	Caves and Tunnels	1553	39
04	Cactuses	1610	28
05	Salary for Robots	1696	37
06	Visits	1726	78
07	Ministry of Truth	1732	57
08	Old Ural Legend	1769	88
09	Barber of the Army of Mages	1744	51
10	Space Bowling	1775	41

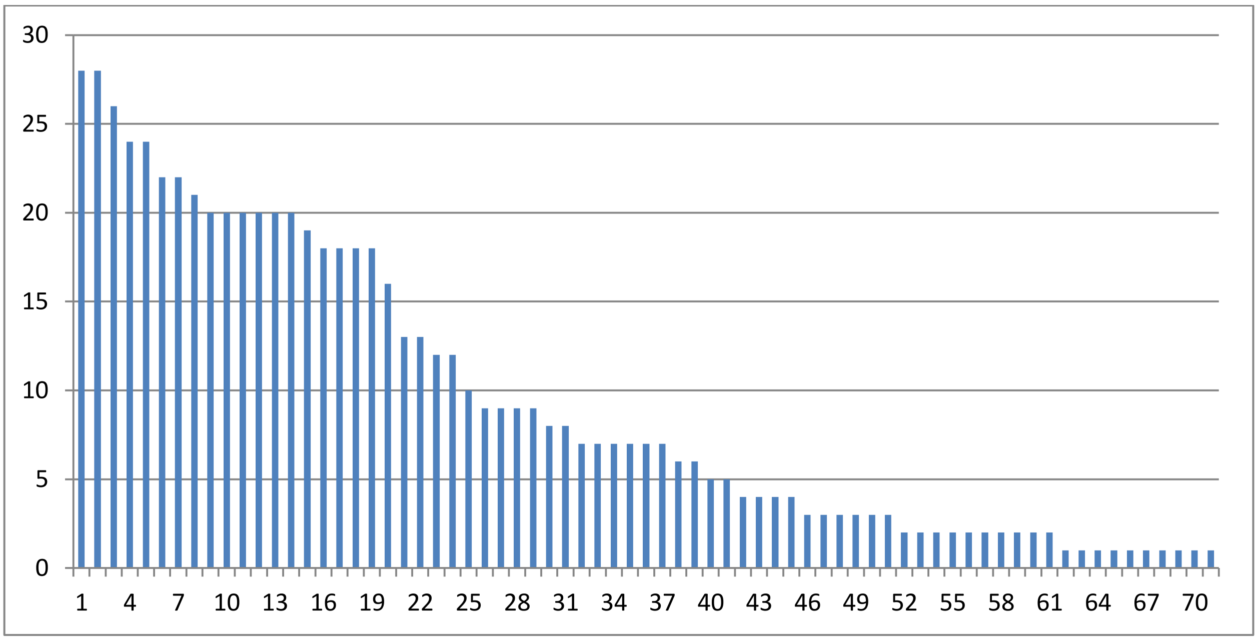
Table 1. Statistics for Round 1

Num	Problem name	ID	Accepted solutions
01	Funny Card Game	1166	18
02	Shots at Walls	1390	5
03	Wires	1460	9
04	Spy Satellites	1478	4
05	Square Country 3	1667	25
06	Monkey at the Keyboard	1677	17
07	Mnemonics and Palindromes 2	1714	18
08	Expert Flea	1763	5
09	Fair Fishermen	1818	23
10	Professional Approach	1819	1

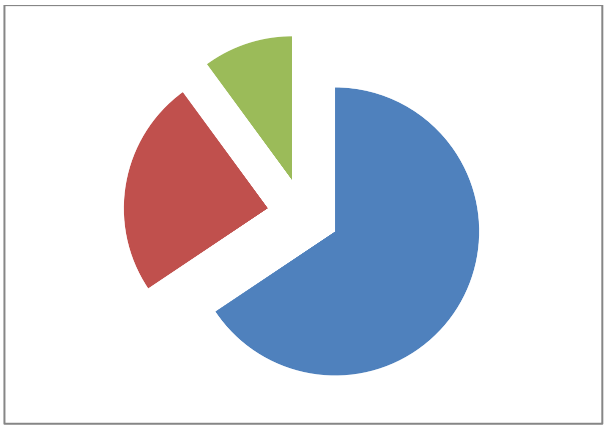
Table 2. Statistics for Round 2

The explanations of the solutions for all 20 problems are provided in this booklet. They were written by a number of different people, some by contestants and some by MDCS Bubble Crew, and you should note that they are not official - we cannot guarantee that all of them are accurate in general. (Still, a correct implementation should pass all of the test cases on the Timus site.)

The organizers would like to express their gratitude to everyone who participated in writing the solutions.



Team results chart [overall]



Average percent of points won by team members

Problem R1 01: Triathlon (ID: 1062)

Time Limit: 2.0 second
Memory Limit: 16 MB

Triathlon is an athletic contest consisting of three consecutive sections that should be completed as fast as possible as a whole. The first section is swimming, the second section is riding bicycle and the third one is running.

The speed of each contestant in all three sections is known. The judge can choose the length of each section arbitrarily provided that no section has zero length. As a result sometimes she could choose their lengths in such a way that some particular contestant would win the competition.

Input

The first line of the input contains integer number N ($1 \leq N \leq 100$), denoting the number of contestants. Then N lines follow, each line contains three integers V_i, U_i and W_i ($1 \leq V_i, U_i, W_i \leq 10.000$), separated by spaces, denoting the speed of ith contestant in each section.

Output

For every contestant write to the output one line, that contains word "Yes" if the judge could choose the lengths of the sections in such a way that this particular contestant would win (i.e. she is the only one who would come first), or word "No" if this is impossible

Sample

input	output

```
9
10 2 6
10 7 3
5 6 7
3 2 7
6 2 6
3 5 7
8 4 6
10 4 2
1 8 7
```
	```
	Yes
	Yes
	Yes
	No
	No
	No
	Yes
	No
	Yes
	```

Solution:

The constraints for this problem were relatively relaxed (in terms of both size and properties of the input) so it was possible to solve the problem in several different ways and BubbleCup contestants have come up with some very creative solutions. One cool way to solve it, which we will explain now, is to treat it as a geometry problem.

Let's choose any of our N contestants – without loss of generality we will assume we have chosen the first one. Now, for all other contestants $i = 2..N$, we make expressions $x \cdot \left(\frac{1}{V_i} - \frac{1}{V_1}\right) + y \cdot \left(\frac{1}{U_i} - \frac{1}{U_1}\right) + z \cdot \left(\frac{1}{W_i} - \frac{1}{W_1}\right)$. If we can choose a vector (x, y, z) such that the value of this expression is greater than zero it means that when the judge picks x, y and z as respective distances for the three disciplines the first contestant beats contestant i, and if we can choose (x, y, z) such that all $N - 1$ expressions have values greater than zero it means that the first contestant can win the race.

Of course, an equation of the type $x \cdot a + y \cdot b + z \cdot c > 0$ defines an open half-space in Euclidean 3D space. So, geometrically speaking, determining whether a chosen player can win reduces to determining whether the **intersection of half-spaces** is nonempty. There is a bug hiding here, however – we have to make sure that our result makes physical sense! In addition to $N - 1$ half-spaces defined by the expressions above, we have to add half-spaces $x > 0$, $y > 0$ and $z > 0$ to make sure our solutions are positive.

It is known that the problem of finding the intersection of half-spaces is the dual of the problem of finding a **convex hull** of a set of points in the same number of dimensions. Finding a 3D convex hull is tricky but there exists a variety of well-known algorithms– we will not go into detail for any of them, but the reader is encouraged to refer to [1] for a description of a randomized incremental algorithm that works in expected $O(N \log N)$ time (worst-case performance: $O(N^2)$) and $O(N \log N)$ space. You can also find the proof of the duality property (and explanation of the duality concept itself) in [1].

Since we have to do N iterations of the algorithm (one for each contestant), and the running time of one iteration is dominated by convex hull computation, we conclude that the solution for the whole task has expected time complexity of $O(N^2 \log N)$ and worst-case time complexity of $O(N^3)$. (It is nearly impossible to actually achieve $O(N^3)$ running time, but due to the low constraints the solution should pass even if that happens). The space complexity is $O(N \log N)$.

References:

[1] Mark de Berg, Marc van Kreveld, Mark Overmars and Otfried Schwartzkopf, *Computational Geometry: Algorithms and Applications*, 2nd, revised edition, Springer, 2000.

Solution by:
Name: ***Andrija Jovanović***
School: School of Computing, Belgrade
E-mail: ja.andrija@gmail.com

Problem R1 02: Archer's Travel (ID: 1459)

Time Limit: 1.0 second

Memory Limit: 32 MB

Let an archer be a chessman that can move one square forward, back, to the left, or to the right. An archer is situated at the square (1, 1) of an *N* × *M* chessboard (the upper right square of the board is coded as (*N*, *M*)). The archer's goal is to travel through the board and return to the initial position in such a way that each square of the board is visited exactly once (the travel starts with the first move of the archer). It is required to determine the number of different ways to perform such a travel.

Input

Integers *N* and *M* separated with a space. 2 ≤ *N* ≤ 5; 2 ≤ *M* ≤ 10^9.

Output

You should output the number of ways to travel through the board calculated modulo 10^9.

Sample

input	output
2 3	2

Solution:

In this task we need to find the number of directed Hamiltonian cycles in the grid matrix $N \times M$, where $2 \le N \le 5$ and $M \le 10^9$. This task requires dynamic programming with bitmasks as well as fast computation of matrix powers. We will calculate the number of undirected Hamiltonian cycles and at the end just multiply this number by 2.

For $N = 2$ or $M = 2$ it is obvious that there is exactly one such cycle.

For $N = 3$, we will give some mathematical arguments. By coloring the grid in chessboard pattern, it follows that after each step a color of a cell is changed. Therefore, if the number of white and black cells is not equal – there are no Hamiltonian cycles. So assume that M is even. Since cells with coordinates $(1, 1)$ and $(1, 3)$ have only two neighbors, we are forced to have the cycle as shown on Figure 1. Using a symmetry one can easily conclude that there are only two possibilities for next moves (see Figure 1). If $d[M]$ denotes the number of Hamiltonian cycles in table $3 \times M$, we have the recurrence $d[M + 2] = 2 \cdot d[M]$ with starting value $d[2] = 1$. Finally, $d[M] = 2^{\frac{M}{2}}$ for M being even.

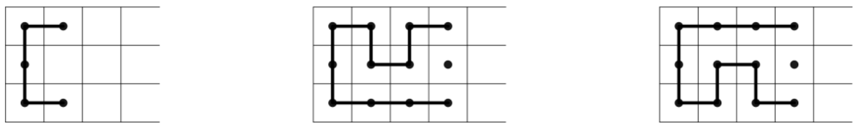


Figure 1. Example for $N = 3$.

For $N = 4$ we can extend the above argument and establish a similar, although much more complicated, recurrence formula. But here we will present a more general method for $N = 4$ and $N = 5$.

In order to solve these linear homogenous recurrence relations, we will use the matrix multiplication method. To illustrate this, consider Fibonacci numbers, defined as $F_1 = F_2 = 1$ and $F_{n+1} = F_n + F_{n-1}$. In order to the calculate n-th Fibonacci number, we can consider the matrix $A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ and verify the following identity $[F_n \ \ F_{n-1}] \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = [F_{n+1} \ \ F_n]$. Therefore, by taking n-th power of matrix A, one can easily calculate F_n with the starting row vector $[1 \ \ 1]$.

We can calculate the n-th power of a matrix A in time complexity $O(m^3 \cdot \log n)$, where m is the dimension of a square matrix A. This can be done using a general principle called exponentiation by squaring:

```
==================================================================================
             Function: Exponential squaring
 ---------------------------------------------------------------------------------
             result = 1;
             while exponent > 0 do
                if (exponent & 1) == 1 then
                   result = (result * base) mod modulus;
                   exponent = exponent >> 1;
                   base = (base * base) mod modulus;
             return result;
==================================================================================
```

For $N = 4$ and $N = 5$, we will code one column state with numbers 0, 1 and 2 by taking the cross section between any two neighboring columns: 0 means that this cell is not an end of some path, while 1 and 2 represent ends of two possible paths. It can be easily seen that there must be exactly two 1's, or two 1's, and two 2's in each state (the rest are zeros). Furthermore, we need to be careful when there are two paths – these are either independent (one above the other) or nested (one inside the other). Using a symmetry, for $N = 4$ there are exactly five possible states $1100, 1010, 1001, 0110$ and 1122.

For example consider the position 12201 for $N = 5$ as shown in Figure 2. In order to continue to the next column and include the empty cell, one has three possibilities: the lower end of the second path will include the empty cell, the lower end of the upper path will include the empty cell and the second path will join the first path (in the last case again lower end of the upper path will include empty cell). We can establish similar relations for other states.

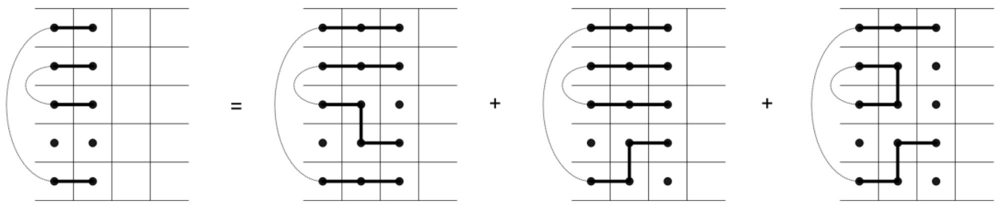


Figure 2. Example state 12201 for *N*=5.

For N=5, there are eleven possible states:

$$11000, 10100, 10010, 10001, 01100, 11220, 11202, 11022, 12210, 12201 \text{ and } 12021$$

and the corresponding matrix is

```
{0, 0, 0, 1, 0, 0, 0, 0, 2, 0, 0}
{0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1}
{0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0}
{2, 0, 0, 0, 2, 2, 0, 0, 0, 1, 0}
{0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0}
{0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0}
{0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0}
{2, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0}
{0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1}
{0, 2, 0, 0, 0, 0, 0, 2, 0, 0, 0}
{0, 0, 0, 1, 0, 0, 0, 0, 2, 0, 0}
```

We use starting row vector $[0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0]$ and the final solution is the sum of the third and twice the eighth element of the product (these are the only possible ending positions).

Solution by:

*Name: **Aleksandar Ilić***

Organization: Facebook Inc.

E-mail: aleksandari@gmail.com

Problem R1 03: Caves and Tunnels (ID: 1553)

Time Limit: 3.0 second
Memory Limit: 64 MB

After landing on Mars surface, scientists found a strange system of caves connected by tunnels. So they began to research it using remote controlled robots. It was found out that there exists exactly one route between every pair of caves. But then scientists faced a particular problem. Sometimes in the caves faint explosions happen. They cause emission of radioactive isotopes and increase radiation level in the cave. Unfortunately robots don't stand radiation well. But for the research purposes they must travel from one cave to another. So scientists placed sensors in every cave to monitor radiation level in the caves. And now every time they move robots they want to know the maximal radiation level the robot will have to face during its relocation. So they asked you to write a program that will solve their problem.

Input

The first line of the input contains one integer N $(1 \le N \le 100.000)$ — the number of caves. Next $N-1$ lines describe tunnels. Each of these lines contains a pair of integers a_i, b_i $(1 \le a_i, b_i \le N)$ specifying the numbers of the caves connected by corresponding tunnel. The next line has an integer Q $(Q \le 100000)$ representing the number of queries. The Q queries follow on a single line each. Every query has a form of "$C\ U\ V$", where C is a single character and can be either $'I'$ or $'G'$ representing the type of the query (quotes for clarity only). In the case of an $'I'$ query radiation level in U-th cave $(1 \le U \le N)$ is incremented by V $(0 \le V \le 10000)$. In the case of a $'G'$ query your program must output the maximal level of radiation on the way between caves with numbers U and V $(1 \le U, V \le N)$ after all increases of radiation ($'I'$ queries) specified before current query. It is assumed that initially radiation level is 0 in all caves, and it never decreases with time (because isotopes' half-life time is much larger than the time of observations).

Output

For every 'G' query output one line containing the maximal radiation level by itself.

Sample

input	output
4 1 2 2 3 2 4 6 I 1 1 G 1 1 G 3 4 I 2 3 G 1 1 G 3 4	1 0 1 3

Solution:

First, let's write the problem statement in graph theory language: We are given a tree (connected acyclic graph) where every node has some value. In a query we are either asked to find the maximum value on a path between two given nodes (and because this is a tree this path is unique) or to change the value of a given node.

We could do the first query type, G query, using any graph search algorithm (*BFS* for example) and find this value in time complexity of $O(N)$ and the second query type, I query, in constant time. Unfortunately, this naïve approach would be too slow for this problem.

Without loss of generality, let us assume that the tree is rooted in node 1. This way we have a father – son relationship between nodes. Let's forget about the second query for a moment. Then we could solve it using a preprocessed matrix $f_{x,y}$ which contains the maximum value on the path between node x and its 2^y-th father. This could be easily preprocessed in $O(N \log N)$ using simple dynamic programming. This would help in finding the asked value in $O(\log N)$ – first we will find the **lowest common ancestor** (LCA) and then maximal value on paths between LCA and given nodes using $f_{x,y}$. But with the second type of query this solution doesn't work.

Let's introduce a relation R between nodes. Nodes v and u are in relation R iff u is a direct son of v and it has the maximal number of nodes in its subtree among other direct sons of node v. If there are several maxima, u has to be the son with the minimal index. This way relation R decomposes the tree into paths (every node is in relation with at most one son). A useful property of this decomposition is the number of distinct paths on path from some node to any other node in its subtree. How can we determine this number?

Let $F(v)$ represent the number of nodes in the subtree rooted at node v. Let's assume we are in node v with $F(v) = X$. Now if we continue going down the tree we can either:

- go to node u which is in relation R with v - but then we do not change path and $F(u) \leq X$
- go to node u which isn't in relation R with v – then we do change path, but $F(u) \leq X/2$ (if $F(u) > X / 2$ then u would be in relation R with v)

We can see that the number of distinct paths on a path from some node to any other node in its subtree is limited by logarithm of the number of nodes in the tree. This is very convenient for our needs.

This decomposition is known as **heavy-light decomposition of a tree**. We can achieve it using this simple algorithm:

- Put every leaf in other paths.
- Using BFS from leaves, put every next node in the same path as its son with maximum number of nodes in its subtree, or if there are several maxima, choose the minimal-indexed among them (please note that you should put a new node in queue only if all of its sons have been visited, that's because we have rooted our tree around node 1).

It's clear that this algorithm takes linear time.

Now, how does this decomposition help us (this is a very good question)? Let's assume we are asked to find maximum value on the path between nodes u and v. We can find LCA in $O(\log N)$ time, find maximum value from LCA to u and from LCA to v and combine these results. So, now it is left to solve the following problem: Find the maximum value from node v to node u if we know that node u is in v's subtree.

We know that the number of distinct paths from v to u is at most $\log N$, so we could go to every path and find the maximum value in it. The problem is that it isn't always the whole path that we are looking at, so we should find maximum value in some part of the path. We can do this in $O(\log N)$ using a well-known structure called the **segment tree** (we could do it even faster, but because of the second query type segment tree is the optimal choice).

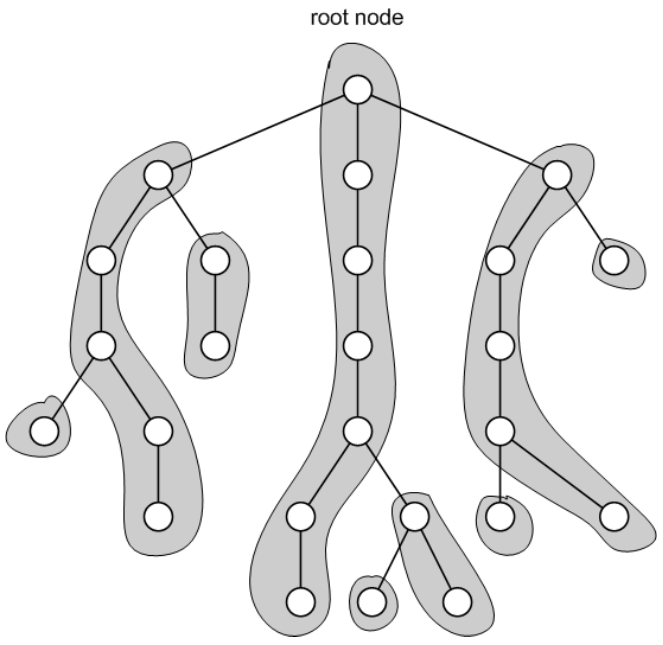


Figure 1. Example of the paths in heavy-light decomposition of a given tree.

This completes the solution for this task. Let's summarize:

- Find decomposition of tree – $O(N)$.
- On every path in decomposition construct segment tree - overall $O(N \log N)$.
- Initialize the matrix G where $G_{u,x}$ is 2^x-th father of u (we need this for finding LCA) – $O(N \log N)$
- Read queries:
    - If query is to find maximum value on path between nodes u and v then:
        - Find LCA of u and v – $O(\log N)$
        - Find maximum value on path from LCA to u – $O(\log^2 N)$
        - Find maximum value on path from LCA to v – $O(\log^2 N)$
        - Combine results
    - If query is to change value of node
        - Update its value in segment tree of the path for this node – $O(\log N)$

Overall time complexity is $O(N \log N + numQuery \cdot \log^2 N)$. Memory complexity for this approach is $O(N \log N)$.

The idea of decomposing a tree into paths and applying some “array” data structure over them is, more or less, well-known. For those who want to test this algorithm on a slight generalization of this problem, we would recommend the problem Otoci from Croatian Open Competition in Informatics 2009. Another variant of this problem can be found on SPOJ – problem QTREE3.

References:

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to algorithms*, MIT Press (2009)
[2] Michael A. Bender, Martín Farach-Colton, *The level ancestor problem simplified*, Theoretical Computer Science, 321 (2004) 5 – 12
[3] http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor
[4] http://www.hsin.hr/coci/archive/2008_2009/
[5] http://courses.csail.mit.edu/6.897/spring05/lec/lec05.pdf

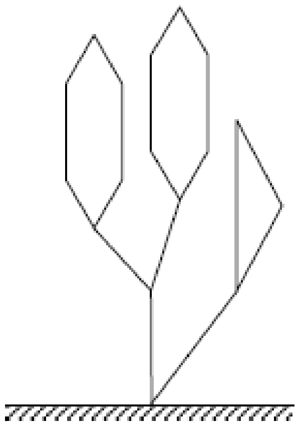
Solution by:
Name: ***Boris Grubić***
School: “Jovan Jovanović Zmaj” Grammar School
E-mail: borisgrubic@gmail.com

Problem R1 04: Cactuses (ID: 1610)

Time Limit: 1.0 second
Memory Limit: 64 MB

There is no doubt that Yekaterinburg trams are the best in the world. Nevertheless, it is Saint-Petersburg that has the largest tram network in Russia. Not long ago, the Saint-Petersburg tram network was included into the Guinness Book of Records as the largest in the world.

Two fans of the tram forum from Yekaterinburg decided to make a trip to Saint-Petersburg to visit the centenary celebration of the tram launch in that city. From their Saint-Petersburg friends they learned that in the previous 15 years the amount of tram service had been constantly decreasing. In many avenues, tram lines were dismantled. Tram service in the city center was minimized, and the city tram network was divided into three fragments, so that it was no longer possible to get by tram from any part of Saint-Petersburg to any other part.



Another thing the travelers learned was that cactuses were in fashion in Saint-Petersburg. Upon their return to Yekaterinburg, they decided to plant a cactus at their office. A cactus is a connected undirected graph such that each of its edges belongs to at most one simple cycle. One vertex of a cactus touches the ground and is called its root.

However, it soon turned out that cactuses became too popular, and all fans of the tram forum already had them. Then the friends decided to get rid of their cactus by a very unusual method: they by turns choose some edge of the cactus and chop it up. This edge is removed, and if the cactus breaks into two parts, then the part that is not connected to the root anymore is thrown out.

The friends have bet a monthly tram ticket on who will chop the last edge growing from the root. Determine who will win if they both play optimally

Input

Along with the vogue of cactuses, the friends follow the Saint-Petersburg vogue to describe the set of edges of a cactus by a family of paths such that in each path all edges are different. The first line contains the amount n of vertices of the cactus, the amount m of paths, and the number r of the root vertex; $1 \leq r \leq n \leq 50.000$.

lines describes a path in the form of a sequence of its vertices. Each description starts with the length of the sequence n_i $(2 \leq n_i \leq 100.000)$. Then there are n_i integers, which are the numbers of vertices of the path, in the order in which they are on the path. Adjacent vertices of any path are different. There can be at most one edge between any two vertices of the cactus. Each edge of the cactus is given in the input only once.

Output

Output "First" if the person who makes the first move wins a monthly ticket assuming that both play optimally. Otherwise, output "Second".

Sample

input	output

`17 2 1`
`15 3 4 5 6 7 8 3 2 9 10 11 12 13 14 9`
`6 2 1 15 16 17 15`
	`First`

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

Solution:

This task requires graph and game theory.

A **rooted graph** is an undirected graph with every edge connected by some path to a special vertex called the root. A **cactus graph** is a connected graph in which every edge belongs to at most one simple cycle. It can be easily proven that the number of edges in a cactus graph is less than 2*n*-1, where *n* denotes the number of vertices in a graph (what is the maximal number of edges of a cactus graph on *n* vertices?). As the input graph is given by an edge disjoint partition of paths, we can store the rooted cactus in O (*n*) memory using graph using an adjacency list.

Nim is a mathematical game of strategy in which two players take turns removing objects from distinct heaps. On each turn, a player must remove at least one object, and may remove any number of objects provided they all come from the same heap. The winner is a player that takes the last object.

The key to the theory of this game is the **binary digital sum** (xor) of the heap sizes. Within combinatorial game theory it is usually called the **nim sum**. The nim sum of x and y is written $x \oplus y$ to distinguish it from the ordinary sum. An example of the calculation with heaps of size 3, 4, and 5 is as follows:

$$3 \oplus 4 \oplus 5 = 011_2 \oplus 100_2 \oplus 101_2 = 010_2 = 2$$

In normal play, **the winning strategy is to finish every move with a nim sum of 0**. This is always possible if the nim sum is not zero before the move. If the nim sum is zero, then the first player will lose if the second player does not make a mistake. For the proof of this fact and other variants of Nim we refer the reader to [1, 2].

The game of Hackenbush is played by hacking away edges from a rooted graph and removing those pieces of the graph that are no longer connected to the ground. We discuss the impartial version of this game in which either player at his turn may chop any edge. Our task is to determine who has the winning strategy (the first or the second player) if both players play optimally.

The simplest case is when there are just pendent paths (also called **Bamboo stalks**) attached to the root vertex. A move consists of hacking away one of the edges, and removing that edge and all edges above it that are no longer connected to the ground. Players alternate moves and the last player to move wins. A single bamboo stalk of *n* segments can be transformed into a bamboo stalk of any smaller number of segments from *n*-1 to 0. So a single bamboo stalk of *n* segments is equivalent to a nim pile of *n* chips.

Playing a sum of games of bamboo stalks is thus equivalent to playing nim.

Consider now a more complicated case – when the game is played on rooted trees (connected graphs without cycles). Since the game is impartial, the general theory tells us that each such tree is equivalent to some nim pile (or if you will to some bamboo stalk). The problem is to find the nim value of each subtree. This may be done using the following principle, known in its more general form as

The Colon Principle: *When branches come together at a vertex, one may replace the branches by a non-branching stalk of length equal to their nim sum.*

We will illustrate this principle on the tree in Figure 1. The leftmost branching vertex has two branches of lengths three and one. The nim sum of three and one is two, so the two branches may be replaced by a single branch of length two. The rightmost branching vertex has two branches of lengths one and two whose nim sum is three, so the two branches may be replaced by a single branch of length three. Continuing in like manner we arrive at the conclusion that the tree on Figure 1 is equivalent to a nim pile of 8 chips. Since this is not zero, the first player has a winning strategy. We leave to the reader to figure out how to choose a winning move (although this is not required in the task).

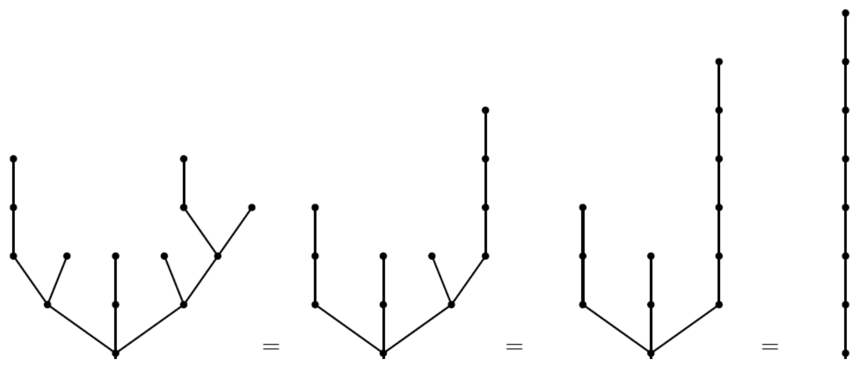


Figure 1. An example of nim sum transformations for trees.

The method of pruning trees given by the colon principle works to reduce all trees to a single bamboo stalk. One starts by depth first search from the root and for each child *v* calculates the nim sum of the subtree rooted at *v*, by recursively calculating nim sums and XOR-summing the values of all children. This gives an O (*n*) dfs algorithm for trees. For the proof of the Colon Principle see [3].

We now consider arbitrary graphs. These graphs may have circuits and loops and several segments may be attached to the ground. To find the equivalent nim pile, we look for an equivalent tree, from which the equivalent nim pile may be found. This is done using the **fusion principle**. We fuse two neighboring vertices by bringing them together into a single vertex and transforming the edge joining them into a loop (an edge joining a vertex to itself). As far as Green Hackenbush is concerned, a loop may be replaced by a leaf (an edge with one end unattached).

The Fusion Principle: *The vertices on any circuit may be fused without changing the nim sum value of the graph.*

The fusion principle allows us to reduce an arbitrary rooted graph into an equivalent tree which can be further reduced to a nim pile by the colon principle. For a proof of the fusion principle see [1]. An example of fusion and colon principles is given in Figure 2.

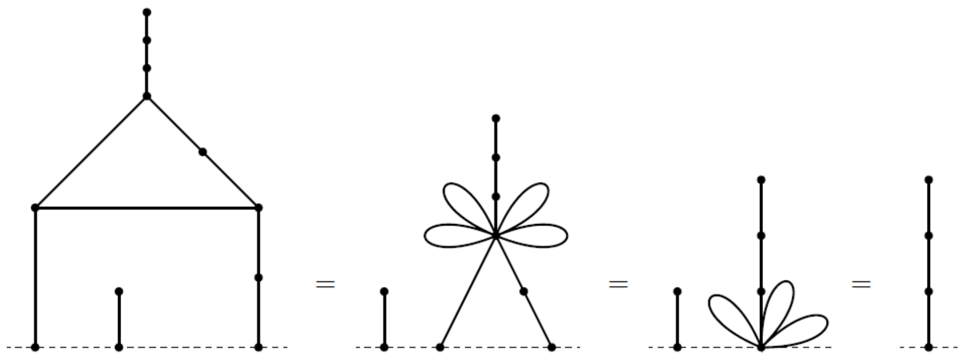


Figure 2. An example of nim sum transformations for general graphs.

We see more generally that a circuit with an odd number of edges reduces to a single edge, and a circuit with an even number of edges reduces to a single vertex. Therefore, in the cactus graph we can transform each cycle to a single vertex or single edge in one graph traversal and reduce the problem to trees. For an even easier solution, one can modify depth-first search for cactus graphs and when examining a back-edge, recursively calculate the nim sum of all subtrees rooted at vertices of a given cycle. The time complexity of this algorithm is O (*n*).

References:

[1] Elwyn R. Berlekamp, John H. Conway and Richard K. Guy: Winning Ways for your Mathematical Plays, Academic Press, Inc., 1982.
[2] http://en.wikipedia.org/wiki/Nim
[3] http://www.math.ucla.edu/~tom/Game_Theory/comb.pdf

Solution by:
Name: ***Aleksandar Ilić***
Organization: Facebook Inc.
E-mail: aleksandari@gmail.com

Problem R1 05: Salary for Robots (ID: 1696)

Time Limit: 2.0 second

Memory Limit: 16 MB

There are n robots on planet PTZZZ. Each robot has its own unique rank — an integer from 1 to n, and should execute all orders from robots with a higher rank.

Once a month all robots get their salary: a positive integer number of credits, not exceeding k. The salary is paid by an accountant-robot. Salary is so important for robots that the first month when all the robots got their salary was named the First month of the First year. There are p months in the year on PTZZZ, so the robots get their salary p times a year.

The salary paid to each robot can be different in different months. If it turns out that all the robots get exactly the same salary as in any month earlier, the accountant-robot will rust of sadness. What is more, the law doesn't allow the accountant-robot to pay salary in such a way that there will be a triple of robots (a, b, c) with rank of a higher than rank of b, rank of b higher than rank of c and the salary of a less than the salary of b and the salary of b less than the salary of c.

The accountant-robot doesn't want to rust, so since the First month of the First year he tries to pay salary in different ways. However, the accountant-robot will rust sooner or later. Your task is to calculate the month number when this will happen.

Input

The only input line contains three space-separated integers n, k and p —the number of robots on PTZZZ, the maximal possible salary and the number of months in a year, respectively ($1 \le n \le 1000$, $1 \le k \le 200$, $2 \le p \le 10^9$).

Output

Output the month number the accountant-robot will rust in. Months are numerated 1 to p.

Sample

input	output
3 3 20	7

Solution:

Here we have a pretty interesting task. Let's reformulate it a little bit. Define A as a sequence of n integers where each element is between 1 and k, inclusive. Call A "bad" if there exist three indexes i, j, k $(i < j < k)$ such that $A_i > A_j > A_k$ and call sequence A "good" otherwise. How many "good" sequences are there? Output that number modulo p.

We could just iterate over all sequences and count only "good" ones, but of course it would be too slow.

Let’s assume that we have one “good” sequence for n– 1 numbers. Suppose we add the number x, which is between 1 and k (inclusive), at the end of this sequence. Is this sequence “good”?

Because the first $n - 1$ numbers are “good”, the only possibility that this sequence is “bad” is that there are two indexes i, j $(i < j)$ such that $A_i > A_j > x$. Because we don’t care about indexes, we can just consider the maximum A_j such that there exists $A_i > A_j$ where $i < j$. If the sequence is “good”, the number x can be a new such value. When does it happen? This happens if $x > A_j$ and there exists an index k such that $A_k > x$, but because we don’t care about indexes, we can just consider maximum A_k.

Now with this we can represent sequence of integers A with three numbers:

- current length of the sequence A, denoted by $length(A)$
- the maximum number such that there is a larger number before it (with a smaller index), denoted by $F(A)$
- maximal value in the sequence, denoted by $G(A)$

We can directly see that not all sequences have $F(A)$, but let’s set $F(A) = 0$ for those sequences. Also, we have that $F(A)$ is strictly smaller then $G(A)$.

Let’s define a three-dimensional matrix d, such that

$d[n, a, b]$ represents the number of “good” sequences A

such that $length(A) = n, F(A) = a$ and $G(A) = b$

How can we calculate these values? Assuming that we have calculated all $d[m, p, q]$ for all m, p, q such that $m < n$ and that the last number in sequence is x, we have five cases:

1) Case $x < a$: this case is not possible because then we would have a “bad” sequence (the number a is bigger than x and we have a number before a which is bigger than a);
2) Case $x < a$ and x$< b$: this case is not possible either because the maximal element among the first n– 1 elements is b, but then x is $F(A)$ and $x > a$.
3) Case x$=$ a and $a \neq 0$: we should add $d[n - 1, p, b]$ for all $p \geq 0$ and $p \leq a$
4) Case $x = b$: we should add $d[n - 1, a, p]$ for all $p > a$ and $p \leq b$
5) Case $x > b$: this is not possible because $G(A)$ then would be x, but $x \neq b$

So, we have:

$$d[n, a, b] = \sum_{0 \leq p \leq a} d[n - 1, p, b] + \sum_{a < q \leq b} d[n - 1, a, q]\,, \quad for\ a > 0$$

$$d[n, a, b] = \sum_{a < q \leq b} d[n - 1, a, q], \qquad for\ a = 0$$

For the initial states we have $d[1,0, b] = 1$ for all b, $1 \leq b \leq k$. The final results is $\sum_{a<b} d[n, a, b]$.

A naive implementation of this idea runs in $O(n \cdot k^3)$, which is too slow for our constraints. We can speed up our algorithm if we return the required sums in constant time. We can achieve this by creating two matrices s_1 and s_2 defined as:

$$s_1[n, a, b] = \sum_{0 \leq p \leq a} d[n, p, b]$$

$$s_2[n,a,b] = \sum_{a<q\leq b} d[n,a,q]$$

The initialization of these matrices can be done using the following recurrent relations:

$$s_1[n,a,b] = s_1[n,a-1,b] + d[n,a,b]$$

$$s_2[n,a,b] = s_2[n,a,b-1] + d[n,a,b]$$

Finally we have that

$$d[n,a,b] = s_1[n-1,a,b] + s_2[n-1,a,b], for\ a > 0$$

$$d[n,a,b] = s_2[n-1,a,b], \qquad for\ a = 0$$

Actually, there is one more thing we should do. Memory complexity of this solution is $O(n \cdot k^2)$ which gives Memory Limit Exceeded, but we can note that we only need the last two matrices of dimension k^2 for every array. Pay attention that you should do all calculations modulo p.

This completes the solution for this task. The time complexity is $O(n \cdot k^2)$ and the memory complexity is $O(k^2)$.

Solution by:
Name: ***Boris Grubić***
School: "Jovan Jovanović Zmaj" Grammar School
E-mail: borisgrubic@gmail.com

Problem R1 06: Visits (ID: 1726)

Time Limit: 1.0 second

Memory Limit: 64 MB

The program committee of the school programming contests, which are often held at the Ural State University, is a big, joyful, and united team. In fact, they are so united that the time spent together at the university is not enough for them, so they often visit each other at their homes. In addition, they are quite athletic and like walking.

Once the guardian of the traditions of the sports programming at the Ural State University decided that the members of the program committee spent too much time walking from home to home. They could have spent that time inventing and preparing new problems instead. To prove that, he wanted to calculate the average distance that the members of the program committee walked when they visited each other. The guardian took a map of Yekaterinburg, marked the houses of all the members of the program committee there, and wrote down their coordinates. However, there were so many coordinates that he wasn't able to solve that problem and asked for your help.

The city of Yekaterinburg is a rectangle with the sides parallel to the coordinate axes. All the streets stretch from east to west or from north to south through the whole city, from one end to the other. The house of each member of the program committee is located strictly at the intersection of two orthogonal streets. It is known that all the members of the program committee walk only along the streets, because it is more pleasant to walk on sidewalks than on small courtyard paths. Of course, when walking from one house to another, they always choose the shortest way. All the members of the program committee visit each other equally often.

Input

The first line contains the number n of members of the program committee ($2 \le n \le 10^6$). The i-th of the following n lines contains space-separated coordinates x_i, y_i of the house of the i-th member of the program committee ($1 \le x_i, y_i \le 10^6$). All coordinates are integers.

Output

Output the average distance, rounded down to an integer, that a member of the program committee walks from his house to the house of his colleague.

Sample

input	output

3
10 10
20 20
10 20
	13

Solution:

In this problem we are given n points and asked to calculate the average distance between two points. Clearly the distance defined in this problem is **Manhattan distance** - since for walking from one to another point one can use only paths that are parallel to x-axis or to y-axis.

Let us denote the coordinates for the i-th point, $1 \le i \le n$, as (x_i, y_i). Let $d_M(i,j)$ denote the distance between points i and j, i.e. $d_M(i,j) = |x_i - x_j| + |y_i - y_j|$. S will be the sum of distances of all pairwise distinct points. Note that for every two points i and j both $d_M(i,j)$ and $d_M(j,i)$ will be counted for S.

Therefore, the output should be $\frac{S}{n(n-1)}$. How can we calculate S efficiently?

Obviously, it can be calculated in time $O(n^2)$, but taking into account the constraints of the problem, that would be highly inefficient. Formally,

$$S = \sum_{i=1}^{n}\sum_{j=1}^{n} d_M(i,j)$$

Since $d_M(i,j) = d_M(j,i)$ we can rewrite S as:

$$S = 2\sum_{i=1}^{n}\sum_{j=1}^{i} d_M(i,j) = 2\sum_{i=1}^{n}\sum_{j=1}^{i} |x_i - x_j| + |y_i - y_j| = 2\left(\sum_{i=1}^{n}\sum_{j=1}^{i} |x_i - x_j| + \sum_{i=1}^{n}\sum_{j=1}^{i} |y_i - y_j|\right)$$

From the above equation we conclude that in order to compute S one can split calculation into two parts - calculating x-distances and calculating y-distances.

If x values and y values were sorted, independently from each other, then in the equation we can get rid of absolute values. Therefore, from now on assume that x and y values are independently sorted in increasing order, and rewrite S in the following way:

$$S = 2\left(\sum_{i=1}^{n}\left(i \cdot x_i - \sum_{j=1}^{i} x_i\right) + \sum_{i=1}^{n}\left(i \cdot y_i - \sum_{j=1}^{i} y_i\right)\right)$$

Let $P_x[k] = \sum_{i=1}^{k} x_i$, and similarly $P_y[k] = \sum_{i=1}^{k} y_i$. Finally, we can rewrite S in the following way:

$$S = 2\left(\sum_{i=1}^{n}(i \cdot x_i - P_x[i]) + \sum_{i=1}^{n}(i \cdot y_i - P_y[i])\right)$$

In order to sort x and y values one can use quick or merge sort and achieve sorting in time $O(n \log n)$. Thus overall time complexity of the algorithm is $O(n + n\log n) = O(n \log n)$. Solution can be big number, (of the order 10^{15}), so in the implementation we must use $long\ long$ or $\_int64$ types.

We can solve this problem in another way (with same time complexity).

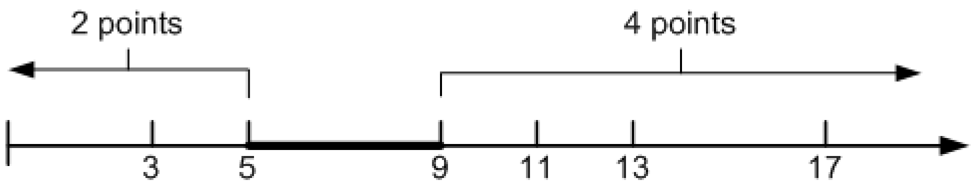


Figure 1. Example for the points on x axces with coordinates $\{3,5,9,11,13,17\}$.

The key observation is that we can calculate the sum of the distances separately for x and for y coordinates. Let's see how can we sum the distances for x coordinate (it is analogous for y). We have that the distance between the k-th and $(k+1)$-th point is $x_{k+1} - x_k$. Instead of calculating the distance between every two points, let see which pairs of points pass through this segment $[x_k, x_{k+1}]$. We can easily see that the number of such pairs is $k \cdot (n-k)$. This is because every pair with one point on the left side of the segment and one point on the right side of the segment passes through (recall that array x is sorted). See example of Figure 1, where we have that 8 pairs of point pass through the segment [5,9].

```
==================================================================================
              Function: getDistance
              Input:  n – number of points
                     X – x coordinates of points
                     Y – y coordinates of points
              Output: average distance between points
 ---------------------------------------------------------------------------------

              sort arrays x and y;
              toReturn = 0;
              for i = 1 to n - 1 do begin
                   segmentX = x [i + 1] – x [i];
                   toReturn = toReturn + segmentX * (i + 1) * (n – 1 – i);

                   segmentY = y [i + 1] – y [i];
                   toReturn = toReturn + segmentY * (i + 1) * (n – 1 – i);
              endif
              numPair = (n * (n – 1)) / 2;
              toReturn = toReturn / numPair;
              return toReturn
==================================================================================
```

Pseudo code for the second algorithm

At the end let us mention the Serbian IOI 2008 preparation where one of the problems was very similar to this one. Here we will just give the problem statement:

You are given a set S of n points in the plane. Coordinates of the given points are integers. For every given point A let us denote

$$d(A) = \sum_{B \in S} max\{|A.x - B.x|, |A.y - B.y|\}$$

In other word, $d(A)$ is sum of distance to all other points, where distance between point A and B is defined as $dist(A, B) = max\{|A.x - B.x|, |A.y - B.y|\}$.

Find a point for which function d has minimal value.

Solution by:
Name: ***Slobodan Mitrović***
School: EPFL Lausanne
E-mail: boba5555@gmail.com

Problem R1 07: Ministry of Truth (ID: 1732)

Time Limit: 1.0 second
Memory Limit: 64 MB

In whiteblack on blackwhite is written the utterance that has been censored by the Ministry of Truth. Its author has already disappeared along with his whole history, and now, while Big Brother is watching somebody else, you, as an ordinary official of the Minitrue, have to delete some letters from the utterance so that another utterance will appear, which has been approved of by the Ministry.

The Ministry of Truth defines a $word$ as a nonempty sequence of English letters and an $utterance$ as a sequence of one or more words separated with one or more spaces. There can also be spaces before the first word and after the last word of an utterance. In order to compare two utterances, one should delete all the leading and trailing spaces and replace each block of consecutive spaces with one space. If the resulting strings coincide, then the utterances are considered to be equal. When the official deletes a letter from the utterance, this letter turns into a space.

Input

The first line contains the original utterance and the second line contains the utterance that must be obtained. The length of each utterance is at most 100000 symbols. The words in both utterances are separated with exactly one space; there are no leading or trailing spaces in each line. The original and the required utterances are different.

Output

If you can't carry out your order, output "I HAVE FAILED!!!" in the only line. Otherwise, output the original utterance replacing the letters that are to be deleted with the underscore character.

Sample

input	output

`Preved to Medved`
`Preved Me`
	`Preved __ Me____`

`this is impossible`
`im possible`
	`I HAVE FAILED!!!`

Solution:

Denote the original utterance as s_1 and the utterance that must be obtained as s_2. Next algorithm checks if string s_2 can be obtained from string s_1 using the rules from the problem statement and, if the answer is 'yes', replaces the letters which should be deleted from the original utterance with the underscore character. The algorithm executes these steps in a circulary fashion:

1. Take the next word from s_2 and search for it in s_1 from the corresponding position (for the first word it is the beginning of s_1).

2. If there is no such substring in s_1 then print “I HAVE FAILED!!!” and quit.
3. Ok, a word was found. Replace all letters in s_1 before the newly found substring and after the previously found substring (or from the beginning of s_1 if it is the first word) with the underscore character.
4. If there are no more words to be searched then replace all letters in s_1 after the last found substring with the underscore character. Print s_1 and quit.
5. The next position to start searching for a new substring in s_1 is two positions to the right from the last letter in the previously found substring.

In step 1 what we actually want is to find out if some pattern appears in the text. For that some fast enough string searching algorithm should be used. Implementations of string searching algorithms from standard libraries are generally slow (for example, functions string::find() in c++ and strstr() in c, their time complexity is $O(n^2)$, where n is text length).

The two probably best-known appropriate algorithms are **KMP (Knuth-Morris-Pratt)** and **Boyer-Moore**. Both of them have time complexity $O(n)$. KMP is easier for implementation, but Boyer-Moore algorithm is in general faster, especially on large alphabets (relative to the length of the pattern). A simplified version of Boyer-Moore algorithm is often implemented in text editors for the <<search>> and <<substitute>> commands. A slightly deeper analysis of the chosen string searching algorithm with the previously described solution shows that the time complexity for the whole solution is $O(n)$, where n is length of s_1.

Solution by:
Name: **Mladen Radojević**
School: The Faculty of Electrical Engineering, University of Belgrade
E-mail: mladen0211@yahoo.com

Problem R1 08: Old Ural Legend (ID: 1769)

Time Limit: 1.0 second

Memory Limit: 64 MB

According to a tale, in the sacred Indian city of Benares, beneath a temple that marked the center of the world, Brahma put three diamond needles and placed 64 gold disks on them. Priests of the temple are busy transferring the disks from needle to needle. It is believed that the world will perish as soon as the task is done.

Another legend is known in Ural. It is said that a monastery is lost in woods at the boundary of Europe and Asia, where the mythical stone flower grew. The monks who live there are doomed to write positive integers on parchment until the Last Judgement. Nobody knows when and where they started this work. There is a legend among monks that when the monastery had been built its first abbot was visited in his dream by the Archangel Gabriel, who showed a stone on the slopes of the Ural mountains where a string of digits was carved. Gabriel ordered to write all the integers starting from the smallest integer that was not a substring of that string. If this legend is true, which integer was written by the monks first?

Input

The only input line contains the nonempty string consisting of decimal digits carved on the stone. The length of the string does not exceed 10^5.

Output

Output the positive integer that is the first number written by the monks.

Sample

input	output
`10123456789`	`11`

Solution:

After reading the problem statement we can shorten it to: what is the smallest positive integer that is not a substring of a given string?

The first thing we should ask ourselves is how big the answer can be. Even if each substring of the given string is a different integer there are at most $\frac{n(n+1)}{2}$ substrings, i.e. integers, where n is the length of the given string. So the answer is at most 5,000,050,001. Obviously, naïve brute-force solution that checks each integer would time out.

A common trick in such problems is "inverting" what we need to do – instead of selecting a number and checking if it is contained in the string, we should get all integers that are contained in the string and find the smallest that is not. Since the answer has no more than $\log_{10} n^2 = 10$ digits, we can iterate the string and get all integers with length less than or equal to 10 out of it in $O(n \cdot\ 10)$. Since $10n$ is not really big we can do one of the following:

a) Use a hash set and then iterate for the smallest integer that is not in the hash set. Since it contains at most $10n$ entries, we will find the answer in at most $10n + 1$ iterations.
b) Do the same as a), using a tree set (which can be slow, depending on the implementation, but STL set should do fine).
c) Store the integers in an array, then sort it afterwards and find the smallest missing positive integer in linear time.
d) Note that the upper bound we used is much bigger than the real one (try to solve the following problem: generate an input to this problem that covers the maximal number of integers) and use arrays instead of a hash set. The maximal answer in Timus' test set is smaller than 12,000,000, so the memory is quite enough.

The complexity is $O(n \cdot \log n^2) = O(n \log n)$ for finding the numbers and $O(n \log n)$ for finding the smallest missing one in a) and d), or $O(n \log n \log(n \log n))$ if we are using b) or c).

The overall complexity with the faster solution is $O(n \log n)$.

Solution by:
Name: ***Alexander Georgiev***
School: Sofia University, Sofia, Bulgaria
E-mail: espr1t.net@gmail.com

Problem R1 09: Barber of the Army of Mages (ID: 1774)

Time Limit: 0.5 second

Memory Limit: 64 MB

Petr, elected as a warlord of the army of mages, faced a challenging problem. All magicians recruited in the army had heavy beards, which were quite unacceptable for soldiers. Therefore, Petr ordered all recruits to shave their beards as soon as possible. Of course, all magicians refused to do it, referring to the fact they don't know any shaving spell. Fortunately, a magician Barberian agreed to shave all recruits.

Barberian can cast a “Fusion Power” spell which shaves beards of at most k magicians in one minute. In order to achieve full effect every magician should be shaved twice: the first spell shaves close, the second spell shaves even closer. For each recruit Petr appointed a time when he should visit Barberian. Unfortunately, the discipline in the new army is still far from perfect, so every magician will come to Barberian in time, but everyone will wait for the shave until his patience is exhausted and will disappear after that.

Determine whether Barberian will be able to shave beards of all magicians before they disappear.

Input

The first line contains two space-separated integers n and $k (1 \le n, k \le 100)$, which are the number of recruits in the army and the number of magicians Barber can shave simultaneously. The i-th of the following n lines contains space-separated integers t_i and s_i $(0 \le t_i \le 1000,\ 2 \le s_i \le 1000)$, which are the time in minutes, at which the i-th magician must come to Barberian, and the time in minutes he is ready to spend there, including shaving time.

Output

If Barberian is able to shave beards of all magicians, output “Yes” in the first line. The i-th of the following n lines should contain a pair of integers p_i, q_i which are the moments at which Barberian should cast the spell on the i-th magician $(t_i \le p_i < q_i \le t_i + s_i - 1)$. If at least one magician disappears before being completely shaved, output a single word “No”.

Sample

input	output

`3 2`
`1 3`
`1 3`
`1 3`
	`Yes`
	`1 2`
	`1 3`
	`2 3`

`2 1`
`1 3`
`1 3`
	`No`

Solution:

First we are going to solve an easier version of this task, where one magician has to be shaved only once, and Barberian can shave only one magician in one second. Let's make a graph in which each magician is represented with a node. We will denote these nodes with M, where M_i represents the i-th magician. We can notice that there is at most 2000 seconds, so we can make a node for every second. Let's call these nodes S, where S_i represents i-th second. Edges will be constructed in the following way: if the i-th magician comes to Barberian at time t and he can wait q seconds, we should connect node M_i with nodes $\{ S_t, S_{t+1}, S_{t+2}, \dots, S_{t+q-1} \}$. Now it is quite obvious that it is a **bipartite matching problem**. If there is a **perfect bipartite matching** on that graph there is a solution and we should write all edges which are in the perfect matching, otherwise there is no solution.

Now let's extend this solution for our problem. Because every magician should be shaved 2 times, we should make a global node G, which will be connected to all nodes in M with weight 2. The second condition says that Barberian can shave k magicians in one second, so we should make a second global node T, which will be connected to all nodes in S with weight k. Weight between nodes in M and nodes in S is 1, because every magician can be shaved only once in one second. Now we have a standard **maximum flow problem** between nodes G (sink) and T (target). If the maximum flow through that graph is $2 \cdot N$, there is a solution and we should write edges which are in maximum flow, otherwise there is no solution. Because there are at most $2 \cdot N$ augmenting paths, the complexity of this algorithm is only $O(N \cdot E)$. Memory complexity is also $O(N \cdot E)$.

Solution by:
Name: ***Demjan Grubić***
School: "Jovan Jovanović Zmaj" Grammar School, Novi Sad
E-mail: demjangrubic.f@gmail.com

Problem R1 10: Space Bowling (ID: 1775)

Time Limit: 1.0 second
Memory Limit: 64 MB

The inhabitants of planets orbiting around the pulsar PSR 2010+15 enjoy playing space bowling. A few cylindrical pins of unit diameter are set on a huge field. A player chooses a certain point of the field and rolls a ball from this point, trying to destroy as many pins as possible. After the ball is released, it rolls in a straight line, touching the surface all the time before rolling away from the field. If the ball touches a pin, this pin dematerializes, and the ball doesn't change direction. To score a *strike*, the player has to destroy at least k pins in one shot.

Unfortunately, aliens haven't yet invented a machine that would return the balls that rolled away from the field. Instead, they use a machine that materializes a new ball from vacuum before each shot. A player enters the diameter and in a second he obtains a ball of exactly the same diameter.

It is time for an alien Vas-Vas to roll a ball. There are n pins standing on the field at the moment. Help Vas-Vas to determine the minimal diameter of a ball, he can score a strike with.

Input

The first line contains space-separated integers n and k ($1 \le k \le n \le 200$). The i-th of following n lines contains space-separated integers x_i andy_i ($-10^5 \le x_i, y_i \le 10^5$), which are the coordinates of the centers of pins. All pins are situated at different points.

Output

Output the minimal possible diameter of a ball which can be used to score a strike, with absolute or relative error not exceeding 10^{-6}. If a strike can be scored with a ball of arbitrarily small diameter, output "0.000000".

Sample

input	output

5 4
0 4
0 6
6 4
6 6
3 0
	1.0000000000

Solution:

Let us first make a couple of observations. Suppose that the starting position of the ball of radius r is (x, y), and that we roll it in some direction d. The gray area in the Figure 1 (plus the area of the ball) represents all points that are touched by the ball. Note that moving the ball to the left, in the opposite direction from d, only increases this area, and hence we can move it „infinitely“ to the left without losing anything - Figure 2.

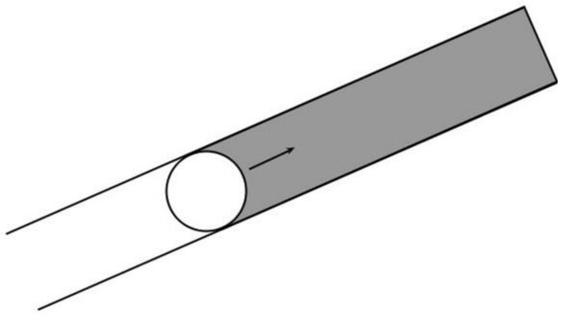


Figure 1: Rolling of the ball

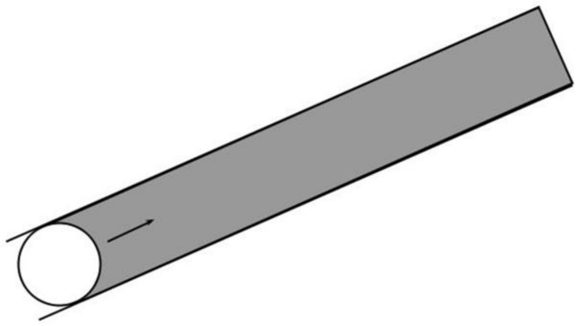


Figure 2: Moving the ball to the left

This allows us to completely ignore the starting position of the ball, and consider only the strip of width r as the area affected by the ball. With this new setting in mind, our goal is to find minimum r such that there is a strip of width r that intersects at least k balls.

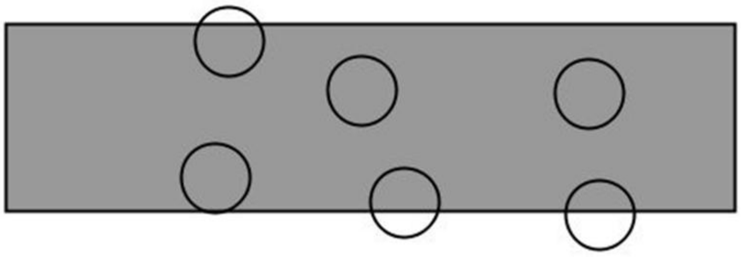


Figure 3: Pins and solution stripe

Since brute-forcing our way through all possible strips is too slow, we need an observation about what the solution looks like. Let us consider an example with the set of pins and a possible solution stripe as on the Figure 3, and suppose that k is 6.

It is easy to see that the given solution can be improved, shortened, by moving the upper boundary of the stripe down and the lower boundary of the stripe up.

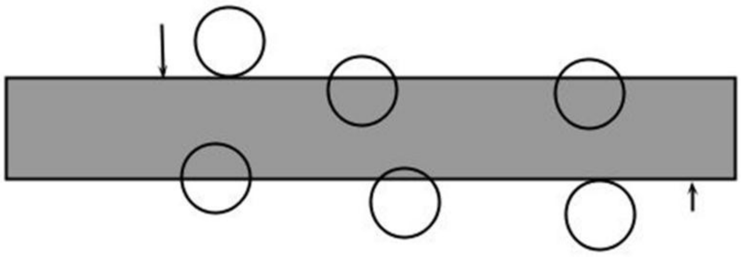


Figure 4: Improved solution

Moving the upper boundary further down causes the loss of intersection with the topmost pin, and hence it would no longer be a possible solution. Thus we can conclude that in the optimal solution there is at least one pin such that the upper boundary is tangential to it and its center lies outside of the stripe (and a similar statement holds for the lower boundary). If that is not the case, we can move the upper boundary further towards the lower one until this happens, because in order for the ball not to be intersected by the stripe anymore we need to pass the moment when its upper bound is tangential to it. Hence stopping exactly at that moment does not change the number of intersections.

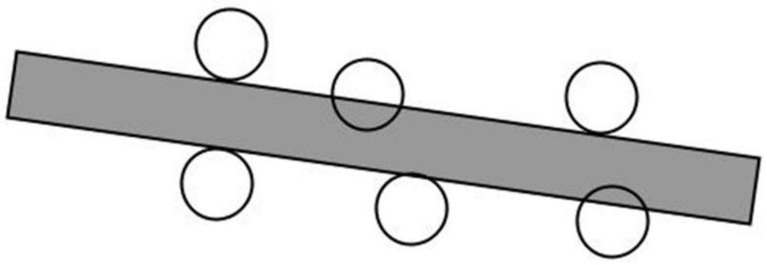


Figure 5: Rotation

However, for two fixed pins, there are infinitely many stripes that are tangential to them (one from the below and one from above). Note that only those stripes that are tangential to at least one additional pin are candidates for the optimal solution, because otherwise we could rotate the stripe in the direction that decreases its width (rotate „around“ two fixed pins) until this happens (Figure 5).

We can now easily transform these observations into a solution. Since we know that the upper and lower boundary of the solution stripe must be tangential to at least one pin, and that it further needs to be tangential to at least one additional pin, we can assume that the center of this additional pin will be above the upper boundary. Hence, we can iterate over all pairs (a, b) of pins, and fix them as pins that are tangential to the solution stripe and whose centers are above it. After that we just need to find a pin that will be tangential to the lower boundary. Note that we can do this greedily, by selecting the $(k-2)$th farthest pin from the line that goes through the centers of pins a and b (and that is also below it, since we are looking for the lower boundary). Taking any other pin gives either a strip of larger width or too few intersections. Since there are n^2 pairs, and for each pair we can find the $(k-2)$th farthest one in $O(n)$, the total running time is $O(n^3)$. Note that for an easier implementation, selecting can be done in $n \log n$ (sort and then pick) which gives $O(n^3 \log n)$, which is still good enough for $n \leq 200$.

Solution by:
Name: ***Rajko Nenadov***
School: ETH Zurich
E-mail: rajkon@gmail.com

Problem R2 01: Funny Card Game (ID: 1166)

Time Limit: 1.0 second
Memory Limit: 16 MB

Of course all of you want to know how to become ACM world champions. There is no exact answer to this question but it is well known that the champions of the last two ACM World Finals enjoyed playing the following funny card game. Two or more players can play this game simultaneously. It is played using a standard 54-card deck. At the beginning the players sit forming a circle. One of the players shuffles the deck and then he deals the cards in clockwise order starting from the neighbour on his left. He deals the top card of the deck to the current player each time. He does it until each player gets five cards. Then he takes the top card of the deck and lays it onto the table face up and he also lays the remainder of the deck nearby (these cards are laid face down preserving their original order). The card laid by the shuffler is considered as the first turn of the game (as if it was made by the shuffler to the player on his left).

The normal game flow as following: the player should cover the last laid card with the card of the same suit or value. If he has none, he takes one card from the top of the deck and again checks this condition. If still there are no matching cards, the move will go to the next player (his left neighbour). But for some cards special rules are applied:

1. If the laid card is 6, the player takes one card from the top of the deck and skips his turn
2. If the laid card is 7, the player takes two cards from the top of the deck (if there is only one card in the deck, he takes just it) and skips his turn
3. If the laid card is Ace the player skips his turn without taking any cards
4. If the player lays Queen, he himself announces the suit of the card it should be covered with
5. Eight is one of the most powerful weapons in this game. When it is laid, the next player has to cover it in any case. If he cannot cover it with his cards he has to take cards from the deck, until he is able to cover it.
6. And the most important card in the game is the King of Spades. If it's laid, the next player takes 4 cards from the top of the deck (if there is not enough cards in the deck, he takes all of them) and skips his turn.

You may assume that the deck is looped and the laid cards are immediately moving to the bottom of the deck. So it can happen that the player has to cover the card by itself. We should say some words about Jokers. Jokers can have any card value by the wish of the player who has it. If the player lays the joker, he assigns a definite card value and suit for it, so this Joker has this assigned value until another player takes it from the deck (if it ever happens). The player is free to use or not to use the Joker whenever he wants (if it is his turn to play, of course). If the player is left without any cards in his hand, he is considered a winner and the game continues without him (his left neighbour becomes the left neighbour of his right neighbour).

If there is only one player left, he is the looser, so he is called a Japanese Fool (it is a Russian name of this game). We are interested in the following situation. Consider the moment when only two players are left in the game. If one of them has a special combination of cards, it may happen that he can lay out all his cards in a some sequence so that the other player won't get a move (he'll just have to take cards from the deck and skip turns) and will be the loser — provided the first one finds the winning sequence.

You will be given the position of the game in which only two players are left. Your task will be to determine whether such a winning sequence for the first player exists or not. We will consider that the first player have already taken all cards from the deck that he had to (if any), so he cannot take any cards from the deck. We will also consider that if the last laid card is a skip-turn card, it was the second player who skipped the turn.

Input

The first line contains cards of the first player separated by spaces. The second line contains the last laid face up card. The card description consists of two characters. The first of them corresponds to the card value (2-9 for digits, T for 10, J for Jack, Q for Queen, K for King and A for Ace). The next describes the suit of the card and may be one of the following: S for Spades, C for Clubs, D for Diamonds or H for Hearts. Joker is represented by a character '*'. If the last laid card is Queen, it is followed by a suit letter. If the last laid card is a joker, then the '*' is followed by an actual card description (the card specified by the player who laid the Joker).

Output

The first line should contain a single word YES or NO signalling whether the winning sequence exists. If the answer is positive the second line must contain the winning sequence of cards separated by spaces. As in the input, the Joker is to be followed by a card specification and the Queen should follow a suit letter. If there is more than one solution you may output an arbitrary one.

Sample

input	output

`6C QD 6S KS 7S *`
`*QHS`
	`YES`
	`7S KS 6S 6C *6D QDS`

Solution:

This problem was tricky in terms of understanding and coding, but the idea hiding behind it is more or less standard.

First, let's read the text of the problem closely and see if we can draw some preliminary conclusions.

- The part about picking up cards from the deck is obviously irrelevant to the solution so we can simply ignore it wholesale.
- Let's call cards that do not have a special rule attached to them *ordinary* cards. It is clear that if the first player has two or more ordinary cards in his hand he cannot finish the game, and if he has exactly one ordinary card the only way to finish is to play that card last.
- Queens and eights can for our purposes be considered ordinary cards as well (although we will have to pay attention to the case when a queen is the last laid card at the start).
- There is no point to play a joker as a card you already possess in your hand – it is always smarter to simply play that card instead and keep the joker because it is more versatile.

- There is no point to play a joker as an ordinary card.

From all this, we can conclude that the largest amount of cards we will be dealing with at any point is 16 – 4 sixes, 4 sevens, 4 eights, the King of Spades, two jokers and one ordinary card.

The simplest way to solve this is to use **backtrack** to try all possible paths until we reach one that successfully gets rid of all the cards. Let's try to estimate how much time this will take.

The highest amount of possible moves in a position (the branching factor) is 7 (if the last card was 6, 7 or 8 of spades and we have at least one joker in hand – note again that we should never use jokers as replacements for cards we currently possess, and that if we have an ordinary card we should never play it until the very end). Of course, we can be in that kind of situation at most five times (until we spend the three spades and the two jokers), and otherwise the branching factor cannot be over 6. Also, we never have a meaningful choice when we have reached the last special card (even if it is a joker). So the upper bound on the number of positions we can go through is $N_{max} = 7^5 \cdot 6^9 \cong 10^{11}$.

Note that a much better upper bound can be calculated by examining the problem more closely and that card configurations that come close to that bound will have many solutions, so only a small portion of the search space will have to be traversed before we find one. This means that it is actually possible that a very fast implementation of this approach using some good heuristics will pass all the tests. However, a much safer way is to use a **bitmask** to store the positions we have already been in and thus avoid unnecessary calculations.

To keep all data about a position during our move we will need the following information: which cards from our original hand are still present (1 bit per card for 13 special cards and 2 more bits to keep track of the jokers) and which card is laid on the table (5 more bits – remember that the only time this can be a non-special card is at the very beginning so we use 4 bits to specify a card, plus one more bit to know if it was actually a joker posing as that card). This gives a total of 2^{20} possible positions at most.

For each position all we care about is whether it leads to a solution, which takes up just one bit. If a solution is found, we can easily reconstruct the path since we are already storing the last-played card for every step of the way.

Now, since we know that we never have to go through any position more than once, we can conclude that the overall time is $O(2^n)$, where n is the number of special cards (or jokers) in our original hand. Since we have seen that n cannot be larger than 15, we finally conclude that, even after including the constant factors, we still have more than enough time to calculate everything. The memory complexity is also $O(2^n)$, but it is probably simpler to just allocate 2^{20} bits (or even 2^{20} booleans) in advance and not worry about it afterwards.

There are several small obstacles involved in the implementation, such as correctly interpreting the input (including cases when a joker or a queen is the last laid card) and dealing with jokers, but we will leave that to the reader.

Solution by:
Name: ***Andrija Jovanović***
School: School of Computing, Belgrade
E-mail: ja.andrija@gmail.com

Problem R2 02: Shots at Walls (ID: 1390)

Time Limit: 3.0 second

Memory Limit: 64 MB

A new pistol is being tested. The pistol can fire shots with variant bullet speeds. In some points of time shots are fired from the point of origin with certain horizontal speeds, and in some other points of time walls are built on a horizontal platform. The walls are non-singular segments lying on lines that do not go through the point of origin. The walls may intersect. For processing of the test results, you are to determine the time that each shot bullet had been flying for. You can assume that the speed of the bullet after shot is constant.

Input

Each line of the input begins with either "shot", "wall", or "end" (without quotes). The number of lines doesn't exceed 50000. After "shot", the two coordinates of speed of the bullet are listed; the speed cannot be zero. After "wall", the four numbers follow, being the coordinates of wall's beginning and end. "end" denotes the end of the input. All the coordinates are integers whose absolute values doesn't exceed 10000. All the events are listed in chronological order, and time intervals between the events exceed the time needed to build a wall, or the time needed for bullet to reach the next wall or end of the proving ground.

Output

For each of the shots, you must output the single number, on a line by itself: the time that the bullet had been flying for, with precision of 10^{-6}. If the bullet doesn't hit any wall, you must output "Infinite" instead of a number.

Sample

input	output
`shot 1 0`	`Infinite`
`wall 1 0 0 1`	`0.50000000000000000000`
`shot 1 1`	`Infinite`
`shot -1 3`	`0.50000000000000000000`
`wall 1 0 -1 2`	`0.33333333333333333333`
`shot -1 3`	`2.00000000000000000000`
`wall 1 1 -1 1`	`0.05000000000000000000`
`shot -1 3`	`0.00020000000000000000`
`wall 2 3 2 -3`	`2.00000000000000000000`
`wall 3 -2 -3 -2`	`0.00100000000000000000`
`shot 1 -1`	`Infinite`
`shot 40 -39`	`0.00099950024987506247`
`shot 9999 -10000`	`1.00000000000000000000`
`shot -1 -1`	`0.50000000000000000000`
`shot -3000 -2000`	`1.00000000000000000000`
`shot -3001 -2000`	`0.90909090909090909091`
`shot -3000 -2001`	`0.43478260869565217391`
`shot 1 0`	`0.83333333333333333333`
`shot 1 1`	`2.00000000000000000000`
`wall -1 2 10 -10`	`3333.33333333333333333`

shot -1 1 shot 0 1 shot 1 1 shot 1 0 shot 1 -1 wall 0 -10000 -10000 0 shot -2 -1 end	

Solution:

We can consider each shot having two attributes – direction and speed. The direction is the polar angle at which the shot is fired. We will use it to determine the wall which the shot hits. The speed is the change of each coordinate for 1 time unit and it is given in the input. Let's denote them with vx, the change in x-coordinate, and vy, the change in y-coordinate. Once we know which wall is shot, if any, we can easily find the time the bullet had been flying for.

We find the equation for the line of the wall in the form $A \cdot x + B \cdot y = C$. We know that the shot crosses the wall - so the bullet crosses the line. The bullet is at point $(t \cdot vx, t \cdot vy)$ at time t. From this we have the equation $A \cdot t \cdot vx + B \cdot t \cdot vy = C$. We can find t from this equation.

Now the tricky part: how to find which wall is hit by the bullet? For each shot we know the direction – an angle from segment $[0, 2 \cdot \pi)$. We must use some data structure so we can do two things with it:

- answer the question: which wall is hit if the direction of the shot is α ?
- insert a new wall and update the data structure.

At each time we keep a sequence of intervals and a wall corresponds to each interval. It looks like: $[\alpha_1, \beta_1]$ – wall w_k for $k \in [1, p]$, where $0 \leq \alpha_1 < \beta_1 < \alpha_2 < \beta_2 < \cdots < \alpha_p < beta_p < 2\pi$. There might be gaps in this sequence of intervals. When there is a gap the bullet hits no wall and we must output "*Infinite*".

Let's imagine we add a wall. We know the coordinates of the wall's both ends, so we can find their polar angles – say they are α_x and β_x (we have to be careful when the wall crosses the Ox axis, i.e. when the angle 0 should be in the interval). If no other interval we had in our structure before intersects with $[\alpha_x, \beta_x]$, we can just add it to the data structure and say it corresponds to the current wall we add. This means that no other wall had covered the interval $[\alpha_x, \beta_x]$

Otherwise we have to process each interval which intersects it. The order in which we process them doesn't matter. Imagine we have to process an interval $[\alpha_y, \beta_y]$ which crosses $[\alpha_x, \beta_x]$. First we remove $[\alpha_y, \beta_y]$ from the data structure. We see the walls corresponding to both intervals and decide which is closer to the origin at each angle – we get some new intervals. We are left with a part of $[\alpha_y, \beta_y]$ – which is the interval which intersected the interval we add. This part is added to the data structure – it won't cross anything else and we are done with it. We have a part of $[\alpha_x, \beta_x]$ – the one we try to add which doesn't cross the part of $[\alpha_y, \beta_y]$ we added in the data structure. But this interval can cross another interval from our structure. While we are left with a part of the interval we are trying to add on this step which crosses an interval from the data structure we have to process those intervals and split them into smaller intervals.

A little more detail about the data structure: it has to support fast searches, erases and insertions. In C++ we can use the set class to store the intervals. We just need to predefine the comparison operator so the intervals are sorted as we want. We have a number of disjoint intervals. We want the interval $[L1, R1]$ to be before $[L2, R2]$ when it is entirely to the “left” of it i.e. $R1 < L2$. If we overload the comparison operator like this we can easily find the intervals which our interval intersects. We just need to search for our interval in the set and the one returned will be one of the intervals intersected by ours. How to find which wall is hit if the direction of the shot is α? We just need to search for the interval $[\alpha, \alpha]$.

Solution by:

Name: ***Yordan Chaparov***

School: 'Atanas Radev' Mathematics High School, Yambol, Bulgaria

E-mail: ancho_mg@abv.bg

Problem R2 03: Wires (ID: 1460)

Time Limit: 1.0 second

Memory Limit: 32 MB

Connoisseur of sound Vova decided to update his equipment. One of the ways to improve the sound is to use point-to-point wiring with heavy wires, and the wires must be as short as possible to diminish the resistance. It is clear how to connect two terminals, it is also easy to find an optimal wiring for three terminals. But what about four terminals?

There are four terminals on a circuit board. You should connect them (there must be a contact between each pair of terminals). It is permitted to add at most three auxiliary terminals and to connect terminals with wire pieces. The goal is to minimize the total length of the wires.

Input

N is the number of tests

$x1\ y1$ the first test

$x2\ y2$

x3 y3

x4 y4

x1 y1 the second test

x2 y2

x3 y3

x4 y4

…

$1 \leq N \leq 100$, x_i, y_i are integers, $-200 \leq x_i, y_i \leq 200$, no two points coincide in each test.

Output

For each test, you should output a line containing the minimal possible length of the wires. The number must be given with at least four fractional digits.

Sample

input	output

2
0 0
2 0
2 1
3 0
 	
0 0
0 1
1 0
1 1
	3.9093
	2.7321

Solution:

This problem in combinatorial optimization is known as the **Steiner tree problem**. The original problem (also known as Euclidean Steiner tree problem) is: Given N points in the plane, the goal is to connect them by lines of minimum total length in such a way that any 2 points are connected directly by a line segment, or via other points and line segments. It may be shown that for the Euclidean Steiner problem points added to the graph (called Steiner points) must have a degree of three, and any two of these three line segments must form a 120 degree angle. It follows that the maximum number of Steiner points that we need to add is $N - 2$, where N is the initial number of given points. In this task we have 4 points, so the number of Steiner points here equals 2. We are not going to prove this, instead we will just show how to find Steiner points and give one of the ways to implement it.

We can split the task into two smaller tasks, based on whether the starting quadrilateral is convex or not.

If the quadrilateral is not convex, we take the 3 points that form the triangle which contains the last point and find **Fermat's point** for this triangle. If Fermat's point is inside a triangle then we reach the minimum total length by connecting all 4 points with it, else we need to connect the point in the triangle with the other 3 points.

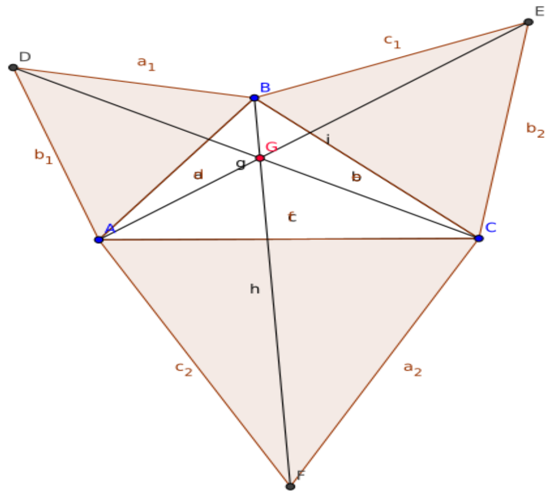


Figure 1 - G is Fermat's point of triangle ABC

If the quadrilateral is convex, we need to find two Steiner points using the following algorithm:

- We construct 2 equilateral triangles using 2 opposite edges of the quadrilateral and mark their third points with E and F (E and F are outside of the quadrilateral)
- We construct circles $K1$ and $K2$ around these 2 triangles
- Intersection of line EF with circles $K1$ and $K2$ consists of two points (possibly identical) G and H.
- Points G and H are Steiner points of that quadrilateral.

We get the minimal network if we connect the points like in the Figure 2.

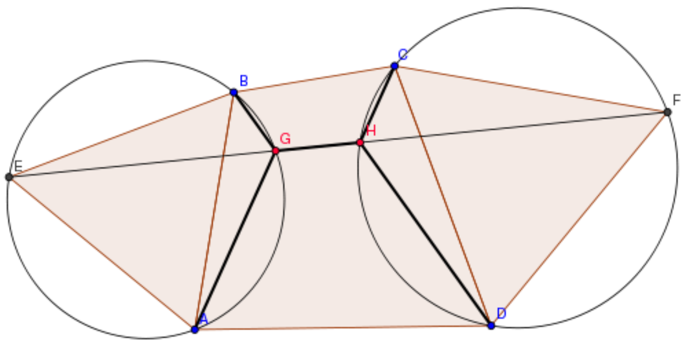


Figure 2. Example for the convex case

If these points are outside of the quadrilateral, then it is best not to include any of them, and connect the starting points by edges of the quadrilateral like in the Figure 3.

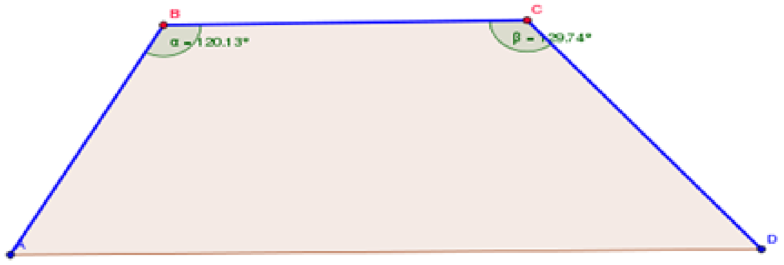


Figure 3. Another example for the convex case

Since it is always better to avoid many special cases in the implementation of a geometry problem, here the easiest way to do that is the following: we find Fermat's point for each 3 points of the quadrilateral (that way we eliminate the case in which the quadrilateral is not convex), and find Steiner points for both pairs of opposite edges. This way we get 8 points which we can include in getting the minimum network. Now we try out all cases for adding 2 points, for adding one point, and without adding any points (which is $8 \cdot 8 + 8 + 1 = 73$ cases). When we have chosen the points we want to add, we make the **minimum spanning tree** where vertices are the points we have, and edges exist between every two of the vertices and their weights are distances between the points. From all the cases, we take the smallest tree we have got as the solution. With such implementation we don't need to manually check whether the quadrilateral is convex nor whether Steiner and Fermat points are inside or outside of the quadrilateral.

Solution by:
Name: ***Dušan Zdravković, Dimitrije Dimić, Stefan Stojanović***
School: Gymnasium "Svetozar Marković", Niš
E-mail: zdravkovicdusan@hotmail.com, dimke92@gmail.com, dolarlord@gmail.com

Problem R2 04: Spy Satellites (ID: 1478)

Time Limit: 1.0 second

Memory Limit: 64 MB

Martian spy satellites have taken a photo of an area on the dark side of the Moon. In this photo, only a lot of light points are seen in the dark. The Martian general suggests that the points are secret objects at lunar military bases. He wants to know how many bases there are on the Moon. The Martians suppose that the bases are seen at the photo as clusters of light points and satisfy the following property: the distance between any two objects at the same base is strictly less than the distance from any object at this base to any object at any other base. The area on the photo can be assumed flat, and the distance between objects having in the photo coordinates (*A*,*B*) and (*C*, *D*) is assumed to be $\sqrt{(A-C)^2+(B-D)^2}$.

Input

The input contains several tests separated by an empty line. The first line of each test contains the number of objects on the photo N. The next N lines contain coordinates of the objects, two integers separated by a space per line. Absolute values of all coordinates do not exceed 10^4. After the last test there is an empty line and the number 0. The sum of all N in the input does not exceed 5 000, the sum of all N^2 does not exceed 400 000, and the sum of all N^3 does not exceed 250 000 000.

Output

For each test, you should output all possible numbers of bases on the photo in the form of a line of length *N* consisting of zeros and ones. For example, the line 110 means that there may be one or two bases on the photo, and the line 011 means that there may be two or three bases.

Sample

input	output

```
4
-1 -1
1 1
1 -1
-1 1

4
1 0
2 4
1 1
0 1

0
```
	```
	1001
	1101
	```

Solution:

First let's see what the problem actually was.

Given a set S of points in the plane, we say that a subset C of S is a valid *'cluster'* for S if and only if for each point p in $S \setminus C$ we know that the (Euclidean) distance between any point q from C and p is strictly greater than the distance between any pair of points in C.

We'll call a valid '*clustering*' of a set S some family F of valid disjoint clusters of S, such that the union of all elements of F yields S, or in other words, each point is a member of exactly one cluster.

In this problem we're interested in the possible sizes F can have. We have to verify for each size of F from 1 to $|S|$ whether that size can be achieved by some valid clustering of S.

For starters, let's gain some intuition about the structure of these clusters, and get some insight into how the sizes of F might be calculated. For example, we might look at a cluster as if it were a graph: We have an edge between each pair of points in the cluster with weight equal to their distance. This graph is obviously complete.

Now, if we're given two clusters, namely C_0 and C_1, and we're given the solutions s_0 and s_1 to C_0 and C_1 respectively as recursive sub-problems of the initial problem on cluster C (where $s_0[i]$ tells us whether or not C_0 can be decomposed into i disjoint clusters, so s_0 and s_1 are Boolean vectors), we can construct the solution s for the cluster C, which is equal to the union of C_0 and C_1. This is done by noting that if $s_0[i]$ and $s_1[j]$, then $s[i+j]$. We'll call this binary operation of combining two solutions '*multiplication*', and our operator will be $*$. Thus $s = s_0 * s_1$.

It's easy to see that this idea generalizes any number of initial clusters so given solutions $s_0, s_1, s_2, \dots, s_{k-1}$ we have $s = s_0 * s_1 * s_2 * \dots * s_{k-1}$.

The idea of the solution is to start with $|S|$ clusters, each equal to a single point in S, and then to grow the clusters by increasing the allowed distance.

The algorithm will be quite similar to **Kruskal's algorithm** for finding the minimum spanning tree of a weighted graph.

We'll be adding the edges to the graph of all points in increasing order of their weight. If we find a set of edges of equal weight, we'll add them all at once. After we add some weight (perhaps more edges of the same weight), we're interested if there's a connected component in this graph that is also its complete sub-graph - as in that case we know we've identified a new cluster. When we find such a cluster C, we can look at the clusters C_0, C_1, $C_2, \dots, C_{k-1}$ that it was made from (the clusters that were connected by adding edges to form C). We may assume that solutions $s_0, s_1, s_2, \dots, s_{k-1}$ are already known, so we get s by multiplying all of these k solutions.

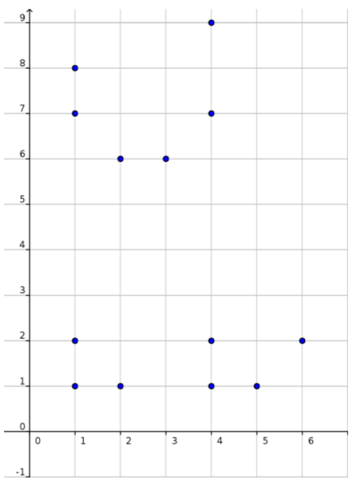


Figure 1

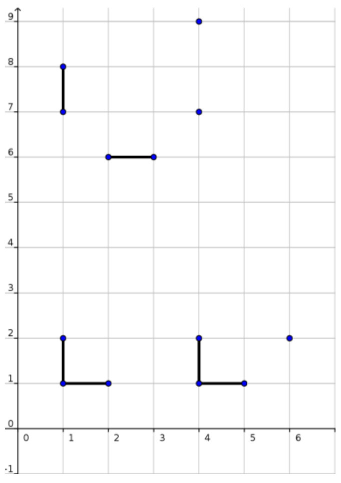


Figure 2

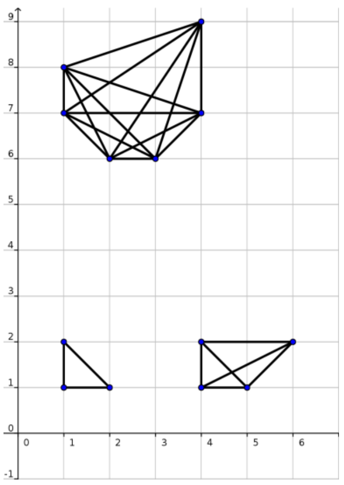


Figure 3

Here we see figures showing some steps of the algorithm on an example. In the beginning each point is a cluster of its own (Fig 1). The first step adds all the edges with weight equal to 1 (Fig 2). After adding some more edges we'll have three valid clusters (complete graphs). In the end they will all merge into a single cluster and in order to calculate its solution we will use the solutions for those three clusters.

Next we can analyze the complexity of this idea. The heaviest operation in this algorithm is obviously multiplication, which seems to take $O(|S|^2)$ time.

In the beginning we have $|S|$ clusters, and merging of any two clusters decreases their number by one, thus we will make a multiplication $O(|S|)$ times. The overall complexity of the algorithm looks like $O(|S|^3)$.

Note that the multiplication algorithm can be optimized by doing breaks:

```
for( int i = 0; i <= |S|; ++i ) {     *
  if( s0[i] == 0 ) break;
  for( int j = 0; i+j <= |S|; ++j )   **
    if( s1[j] ) s[i+j] = 1;
}
```

Now the next idea I will present here justifies the name of our merging operation.

Let's look at these two polynomials:

$$P_0(x) = s_0[0] + s_0[1] \cdot x + s_0[2] \cdot x^2 + \dots + s_0[|S|] \cdot x^{|S|}$$

$$P_1(x) = s_1[0] + s_1[1] \cdot x + s_{10}[2] \cdot x^2 + \dots + s_{10}[|S|] \cdot x^{|S|}$$

It's easy to verify that $s[i] = 1$ if and only if the i'th coefficient of $P_0 \cdot P_1$ is nonzero.

As the multiplication of two polynomials can be done in $O(|S| \log |S|)$ using **FFT (Fast Fourier Transform)**, we've just found a $O(|S|^2 \log |S|)$ algorithm for the problem.

You may note that the $O(|S|^3)$ algorithm with optimizations probably runs faster in practice.

Implementation issues

For the actual implementation of this algorithm one may use the disjoint-set structure. For identifying complete graphs we only need to know the number of edges and nodes in some component. To find the solution, we also need to know the solutions of all active clusters, and id's of clusters that merged into some other cluster.

Exercise 1: Solve the problem with a small modification: for each size i find the number of decompositions that yield i clusters.

Exercise 2: Show that a really simple observation converts the first algorithm to an algorithm with overall complexity of $O(|S|^2)$, which beats the FFT one.

(Hint: a trivial change to the lines marked * and ** makes it an overall $O(|S|^2)$ algorithm, you should note that both loops can go to the actual size of the given cluster, not all the way to $|S|$, which is quite obvious, but it's not that clear how it gives a quadratic complexity.)

Solution by:

*Name: **Gustav Matula***

School: XV. Gimnazija Zagreb

E-mail: syntax.error.ffs@gmail.com

Problem R2 05: Square Country 3 (ID: 1667)

Time Limit: 0.5 second
Memory Limit: 64 MB

The governor of Yekaterinozavodsk region had to deal with ambassadors from one of the nearby states — Square country. All square inhabitants of this country loved the squares of integers. So, they declared to the governor that they would build a square metropolitan from Yekaterinozavodsk to one of the suburbs only if he would be able to fill a rectangular table $N \times M$ with squares of different positive integers in such a way, that the sum of numbers in each row and in each column would also be a square. The governor wasn't a square man, and also he wasn't good in maths, so he asked for your help.

Input

The first line contains an integer T— the number of test cases ($1 \leq T \leq 20$). The following T lines contain the pairs of integers N and M($1 \leq N, M \leq 20$).

Output

For each test case output the required table: N lines with M numbers in each line. All numbers in the table shouldn't exceed 10^{17}. If there is no such table, output -1. Answers for different test cases should be delimited with an empty line.

Sample

input	output

```
3
1 2
3 1
2 2
```
	```
	9 16

	1024
	25
	274576

	4761 8464
	627264 1115136
	```

Solution:

The first thing we should notice is that, as in many other problems at Timus, the part “If there is no such table, output -1” is deceiving – actually an answer always exists. Most of the competitors probably saw this on the discussion board for the problem, so it will not be analyzed further (it will be shown later that there is always a solution by generating the solutions themselves).

Now, before going into the solution itself we should make some crucial observations:

1. The first one is rather obvious – if we multiply a square by a square the resulting number is also a square (for example $4 \cdot 9 = 36$, here 4 and 9 are squares, and so is 36).
2. The second one follows directly from the first one – if we multiply a vector of squares (i.e. sequence

of integers that are squares) by a square, the resulting vector also contains only squares. This is the first observation applied to each of the elements of the vector.

3. The third one uses the first and second one: if we have a vector of squares, whose sum of elements is also a square (i.e. a valid row or column of the required rectangle of numbers), then if we multiply it by a square, the resulting vector is also of the desired type. Let's see why this is correct. If we have a square number M and a vector $\{A_1, A_2, \ldots, A_n\}$, and each of the numbers A_i is a square, as is their sum $S = A_1 + A_2 + \ldots + A_n$, then using observation 2 we know that the vector $\{M \cdot A_1, M \cdot A_2, \ldots, M \cdot A_n\}$ is also a vector of squares, and using observation 1 we know that the sum of the numbers, which is $M \cdot S$ is also a square.
4. The last one is the third one applied several times: If we have two vectors a and b of squares of sizes N and M respectively, whose sum of elements is also a square, then the matrix $C_{i,j} = a_i \cdot b_j$ is of the required type (each of its elements is a square and the sum of each row and column is also a square).

Now we (almost) have a way to generate matrices of the desired type. Just generate two vectors of sizes N and M, respectively, and then generate the matrix. Generating such vectors is rather easy – just generate vectors of random numbers squared and check if their sum is also a square until you find one. These vectors are relatively common, so it doesn't take much time for a computer to find one.

For example if $N = 2$ and $M = 3$, we can chose the vectors $a = \{9,16\}$ and $b = \{4,9,16\}$ to get:

	4	9	36
9	36	81	324
16	64	144	576

We see that the resulting matrix is of the type we want – all numbers are squares and the sums of the rows and columns are also squares.

Why isn't this solution correct? Well, looking at the statement *"…only if he would be able to fill a rectangular table N × M with squares of **different** positive integers in such a way, that the sum of numbers in each row and in each column would also be a square."* we see that we did nothing to guarantee that the numbers will be different. In fact, here is an example that shows that our current solution is wrong (although the matrix has squares inside and the sum of each row or column is also a square, there is a duplicate number – 36):

	1	9	16	25	49
4	4	**36**	64	100	196
9	9	81	144	225	441
36	**36**	324	576	900	1764

From here on there are a few ways we can go to fix this issue. One of them is applying some number theory and setting restrictions on the vectors that would guarantee that there are no duplicates in the final matrix. For example if the vectors we use are squares of prime numbers, and all used prime numbers are unique, this automatically means that the resulting matrix has no duplicates. However, this approach is not as easy as it seems, because generating the vectors becomes more difficult.

We will show another solution that is much easier to think of. It turns out that duplicate numbers are rare (this can also be seen from the examples above and if you generate some longer ones and try it by yourselves). In fact, if we try few random (valid) vectors we quickly find such ones that do not have duplicates in the resulting matrix. So hence the solution – just generate random vectors until you end up with a matrix that has no duplicates. The generation of the vectors should be done relatively smart though, otherwise this solution might time out. The only optimization my solution used in order to fit in the time limit was to store the vectors with different sizes once generated and try them in other combinations. The most time is used to generate valid vectors, so this reduces the time required for this process dramatically.

There is also a safer solution, however. Since the input is quite limited – only numbers up to 20 - we can easily generate the vectors for the 200 different combinations of $(\min(N, M), \max(N, M))$ offline, put them in the code and just generate the matrix when given a query. This is probably the easiest and safest way (there is no test that can break your solution even if eventually additional tests are added). Since the previous solution is actually really close to the time limit even if not optimized, generating all the vectors doesn't take much time. All that is left to be done is put them in the code and write a simple loop that prints the matrix. The complexity of such solution is $O(Q \cdot N \cdot M)$, where Q is the number of queries, N is the number of rows and M is the number of columns of the current query, which is clearly well below the time limit.

Solution by:

*Name**: Alexander Georgiev***

School: Sofia University, Sofia, Bulgaria

E-mail: espr1t.net@gmail.com

Problem R2 06: Monkey at the Keyboard (ID: 1677)

Time Limit: 5.0 second

Memory Limit: 64 MB

The “Entertaining math” showmen decided to ensure their audience in the well-known fact that a monkey randomly pressing keyboard buttons will sooner or later type the required word according to the probability theory.

The monkey taken to the shooting from the city zoo already can type — every second she types one of N first letters of English alphabet with equal probability. Fortunately, the word prepared by the showmen also contains some of these N letters only.

However, prior to giving a keyboard to the monkey, the showmen want to calculate the time it would take her to finish the job. More precisely, after how many seconds the given word will appear in the typed string for the first time?

Input

The first line contains an integer N — the number of letters the monkey can type ($1 \leq N \leq 26$). The second line contains a word proposed by the showmen. The word can contain only the first *N* lowercase Latin letters. Its length is positive and doesn't exceed 30000.

Output

Output the expected time the monkey will need to type a word, rounded down to the nearest integer.

Sample

input	output
2 aa	6
2 ba	4

Solution:

The problem is to find the expected time of the first occurrence of a given string in a random sequence (whose elements are from a given finite alphabet). In short, we have to find the position in an infinite random sequence at which the given string occurs for the first time, **on average**. Although the solution to this problem is quite beautiful (considering the final form) and could be found using good intuition, formal proof and complete understanding requires some strong background in math probability (including theory of stochastic processes and Markov’s chains). We present a solution with non-formal proofs which should be understandable to high school-level students. For a formal solution, one can see [1].

Denote by *X* size of the alphabet, by *S* the given string, by *N* its length and by *A* an infinite random sequence in which each element of given alphabet occurs with equal probability. We say that *S* occurs in *A* at position *pos* for the first time if $A_{pos-N+i} = S_i$, for $1 \leq i \leq N$ (*S* is 1-indexed) and there is no other position *pos'* < *pos* for which that holds. Let P_i = probability that S occurs at position *i* for the first time. Denote by *SOL* – solution to our problem. Then, by definition of expected value of random variable, we have

$$SOL = 1 \cdot P_1 + 2 \cdot P_2 + 3 \cdot P_3 + \dots = \sum_{i=1}^{\infty} i \cdot P_i$$

However, computing solution using the previous formula is not trivial since those probabilities are not easy to calculate and that's why we will use a different approach. Imagine we observe sequence *A* and all occurrences of *S* in *A*. Let's divide those occurrences in 2 categories: **good** and **bad** ones. Definition of good-bad is recursive: first occurrence of *S* in *A* is good; every other occurrence of S is bad if it overlaps with some earlier **good** occurrence, otherwise it is good. For example, if $S = AAA$ and alphabet is $\{A, B\}$, all positions of good occurrences of S in following sequence are maked bold:

BAA**A**ABBABAA**A**BBAA**A**AA**A**ABB ...

Also, denote by *F(S)* the frequency of string S, i.e. the number of its occurrences per length (of random sequence). Since all characters are equally probable, frequency only depends on string length and since the sequence is random, we can consider every successive *N* characters as an independent random string of length *N* (despite the fact that they overlap) – therefore $F(S) = \frac{1}{X^N}$.

Now, let's mark the position of every good occurrence of *S* in the random sequence – that way we get an infinite sequence of positions $p_1, p_2, \dots$ Observe that $p_i - p_{i-1}$ represents the position of the first occurrence of *S* in *A* if we imagine that *A* starts at $p_i + 1$. Therefore we have a (infinite) collection of values *SOL* and we need the average value of those. It is easy to see that the average value for first *n* marks is

$$\frac{\sum_{i=1}^{n}(p_i - p_{i-1})}{n} = \frac{p_n}{n}$$

And therefore

$$SOL = \lim_{n \to \infty} \frac{p_n}{n} = \frac{1}{F(good\ occurrences\ of\ S)}$$

Notice that is **very important** that we only count good occurrences of *S* – otherwise some of our *SOL*'s from collection will not be valid because they had a greater probability of occurring, as some prefix was already there. Also notice that a necessary condition for existence of bad occurrences of *S* is that *S* has a prefix which is at the same time a suffix of *S* – we will call such substrings **infixes**. For example, if $S = ABABA$, than *S* has two infixes: *A* and *ABA* of lengths 1 and 3, respectively. Let $I(S) = \{i \mid S\ has\ an\ infix\ of\ length\ i\}$ be a set of all *S*'s infix lengths. ($I(\mathrm{ABABA}) = \{1, 3\}$). As we will see, those sets fully determine value SOL for a given string *S*.

Now we have to calculate the appropriate frequency. Let *F(S) = s*, *F(good occurrences of S) = g*, *F(bad occurrences of S) = b*. Since every occurrence of *S* in sequence *A* is either good or bad (but not both), we have *s = g + b*. Also, the probability that some good occurrence of *S* will produce a bad one starting at position $N - i + 1$ (within *S*) is equal to $\frac{1}{X^{N-i}}$ if *i* ∈ $I(S)$ (since we need the next *N - i* elements to be exactly $S_{1,N-i}$), and 0 otherwise. This directly implies that the expected number of bad occurrences starting from position $N - i + 1$ of some good one is also $\frac{1}{X^{N-i}}$, because there can be only 0 or 1 such bad occurrences. It follows that the expected number of bad occurrences produced by a (fixed) good occurrence of S is equal to $e_{bad} = \sum_{i \in I(S)} \frac{1}{X^{N-i}}$. Therefore we have $b = g \cdot e_{bad}$, which implies $g = \frac{s}{1+e_{bad}}$. Combining all results so far we finally get

$$SOL = \frac{1}{g} = \frac{1+e_{bad}}{s} = X^N \left(1 + \sum_{i \in I(S)} \frac{1}{X^{N-i}}\right) = X^N + \sum_{i \in I(S)} X^i$$

It only remains to calculate all infixes of the given string *S*. But this problem is solved by the well-known **Knuth-Morris-Pratt (KMP) algorithm** (more precisely, its subroutine). During pre-processing of string *S* the algorithm calculates its $\pi\ function$, defined as $\pi[i] =$ length of the longest prefix of *S* which is also a suffix of $S_{1,i}$, in linear time (of length of *S*). It is not hard to see that $I(S) = \{\pi[N], \pi[\pi[N]], \dots\}$. For a detailed description of the algorithm, see [2]. After that, it is a matter of simple calculation. If we represent our polynomial as

$$a_N X^N + a_{N-1} X^{N-1} + \dots + a_0 = a_0 + X(a_1 + X(a_2 + X(a_3 + \dots + X a_N))\dots)$$

than this problem can be solved in $O(N)$. Note that Big Integer arithmetic is needed.

This is very a interesting problem and we can get some interesting facts after result analysis, for example: expected sequence length is always an integer, it heavily depends on string's self-repetition properties, all strings of same lengths have equal frequencies but not expected sequence lengths, expected sequence length for ABCDE is almost twice as short as for AAAAA (why?) etc…

References:

[1] Terry R. McConnell, *The Expected Time to Find a String in a Random Binary Sequence*

[2] T. Cormen, C. Leiserson at. al., *Introduction to algorithms*, MIT Press

Solution by:

Name: ***Nikola Milosavljević***

School: Faculty of Mathematics, University of Niš

E-mail: nikola5000@gmail.com

Problem R2 07: Mnemonics and Palindromes 2 (ID: 1714)

Time Limit: 3.0 second
Memory Limit: 64 MB

At last, Vasechkin had graduated from the university and it was the time to choose his future. Vasechkin recalled all the inadequate outcomes, unsolvable problems, and incomprehensible problem statements that he encountered at programming contests, so he decided to join a program committee. Soon he was asked to prepare a problem for the forthcoming student contest, which would be dedicated to binary alphabets. The problem had to fall under that topic. However, Vasechkin wanted the participants to remember his problem for a long time, so he decided to give the problem an unusual and complicated name.
Vasechkin decided that the name had to consist of the letters "a" and "b" only and contain exactly n letters. In addition, the name had to be as *complex* as possible. The *complexity* of a name is defined as the minimal number of palindromes into which it can be decomposed. Help Vasechkin to invent the most complex name for his problem.

Input
The only line contains an integer n $(1 \leq n \leq 1000)$.

Output
Output the required name of length n consisting of the letters "a" and "b" only. If there are several such names, output any of them.

Sample

input	output
6	aababb

Solution:

On the Timus online judge, this problem is tagged as the "hardest problem". Before the Bubble Cup qualifications, only 26 contestants solved it. The small acceptance rate is not a consequence of its difficulty. The reason is its method for the problem analysis – "I will sit down and try anything that is reasonably good until I find some general pattern". The source of this problem is NEERC 2009, Eastern subregional contest. It is interesting that during the contest only one team attempted to solve this problem.

The idea behind this problem is the following problem from the International Mathematical Tournament of Towns:

Prove that every binary word of length 60 can be divided into 24 symmetric subwords and that the number 24 cannot be replaced by 14.

As we mentioned, road to the solution is very bumpy, so here we are only going to present the algorithm. Complete analyses of this algorithm and its proof can be found in [1].

Let us denote with $m(w)$, where w is a binary word, the minimal number of palindromes whose product is equal to w. We need to find a value $K(n)$, where

$$K(n) = \max\{m(w) | \, w \; binary \; word \; of \; length \, n\}$$

For small numbers of n, $K(n)$ can be computed with simple backtracking and some dynamic programming:

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$K(n)$	1	2	2	2	2	3	3	4	4	4	5	5	5	6	6

With this data we can suggest and prove a precise formula for $K(n)$

Theorem For every number $n \in N$, where $n \neq 11$, we have that

$$K(n) = \left\lceil \frac{n}{6} \right\rceil + \left\lceil \frac{n+4}{6} \right\rceil + 1$$

Theorem shows that the "worst" word of length n is very asymmetric: it cannot be divided into less then $\frac{n}{3}$ palindromes. It is interesting that a random binary word is far from being symmetric, it cannot be divided into less then $\frac{n}{11}$ palindromes.

After playing a little bit with binary strings, many contestants found different patterns for requested string. One way (that is presented in paper [1]) is:

$$w = \begin{cases} aabab(bbaaba)^m [\![abbaba]\!]^{n-5 \; mod \; 6}, & if \; n \; mod \; 6 \neq 2 \\ aabab(bbaaba)^{m-1} bbaababb, & if \; n \; mod \; 6 = 2 \end{cases}$$

where $m = (n-5)/6$.

Main fact needed for proving this is following

Lemma For every $n \in N$ the word $(bbaaba)^n$ does not contain a palindrome with length greater or equal to 5.

Implementation of this method is pretty straightforward. Time and memory complexities are linear.

References:

[1] Olexandr Ravsky, *One the palindromic decomposition of binary words*, Journal of Automata, Languages and Combinatorics, 2010

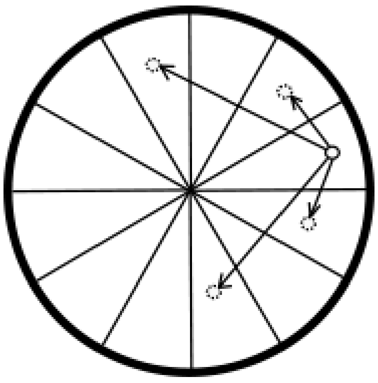
Solution by:
Name: ***Andreja Ilić***
School: The Faculty of Mathematics and Sciences, Niš
E-mail: andrejko.ilic@gmail.com

Problem R2 08: Expert Flea (ID: 1763)

Time Limit: 0.5 second
Memory Limit: 64 MB

A flea has jumped onto a round table used in the popular quiz "What? Where? When?" In this quiz, the questions are put inside envelopes lying on the sectors of the round table. A panel of experts has to answer questions chosen by a roulette pointer from those lying on the table. The flea wants to read all the questions in advance and thus have more time to find the answers.

The round table is divided into n sectors numbered clockwise from 1 to n. The flea has jumped onto the first sector. From this sector it can either run to an adjacent sector or jump across two sectors (for example, if the table is divided into 12 sectors, then in one move the flea can get to sectors 2, 4, 10, and 12). The flea wants to visit each sector exactly once and return to the first sector, from which it will jump down to the floor and run away to think about the questions. Find the number of ways in which the flea can complete its journey.



Input

The only input line contains the number n of the sectors of the round table ($6 \le n \le 10^9$.

Output

Output the number of ways to visit each of the sectors exactly once and return to the first sector modulo $10^9 + 9$.

Sample

input	output
6	12

Solution:

This was one of the hardest problems in qualifications and only five teams managed to solve it. It is interesting that before the competition only fifteen people solved it on Timus online judge. The main reason for this is inaccessibility of the problem with standard methods – it is necessary to precalculate many things, after which the solution can be obtained just by multiplying some hardcoded matrices. There were a couple of different approaches, but here we are going to present a more general technique. This editorial is a short review of [1].

Let us define this problem in graph theory language. We can look at sectors as vertices in graph G and label them with $\{0,1,\dots,n-1\}$. Vertex v, $0 \le v < n$, is connected to four other vertices $(v+1)\ mod\ n$, $(v-1)\ mod\ n$, $(v+3)\ mod\ n$ and $(v-3)\ mod\ n$. Now we have that the number of ways in which the flea can complete its journey is equal to the number of Hamiltonian cycles in this graph. As we mentioned, we will address a more general class of graphs, given by the following definition.

Definition The n-node undirected circulant graph with jumps $1 \le s_1 < s_2 < \dots < s_k \le n-1$ is denoted by $C_n^{s_1,s_2,\dots,s_k}$*. This is the* $2k$ *regular graph with* n *vertices labeled* $\{0,1,\dots,n-1\}$ *such that each vertex* v*,* $0 \le v \le n-1$*, is adjacent to* $2k$ *vertices* $v \pm s_1, v \pm s_2, \dots, v \pm s_k$ *mod* n*. Formally*

$$C_n^{s_1,s_2,\dots,s_k} = \left(V(n), E_C(n)\right), \textit{ where } V(n) = \{0,1,\dots,n-1\} \textit{ and } E_C(n) = \{(i,j) \mid (i-j) \bmod n \in \{s_1,s_2,\dots,s_k\}\}$$

Graph G from our problem is a special case of circulant graph: $G \equiv C_n^{1,3}$. In what follows we assume that the step sizes $s_1, s_2, \dots, s_k$ are fixed and known. To simplify our notation we will often drop the jump list and just write C_n. Let us denote the number of Hamiltonian cycles in graph G by $H(G)$.

The main idea is to, instead of trying to decompose a large circulant graph into smaller ones, work on *step-graphs*, the *unhooked* version of circulant graphs. We want to construct a system of linear recurrence relations on the number of different types of forests in the step graph, and then express the number of Hamiltonian cycles of the circulant graph in terms of the number of different types of forests of the step-graph. This technique is very general and, unlike the algebraic methods previously employed, can also be used to enumerate other parameters of circulant graphs.

Definition The n-node undirected step graph with jumps $1 \le s_1 < s_2 < \dots < s_k \le n-1$ is denoted by $L_n^{s_1,s_2,\dots,s_k} = \left(V(n), E_L(n)\right)$ *where*

$$V(n) = \{0,1,\dots,n-1\} \qquad E_C(n) = \{(i,j) \mid (i-j) \in \{s_1,s_2,\dots,s_k\}\}$$

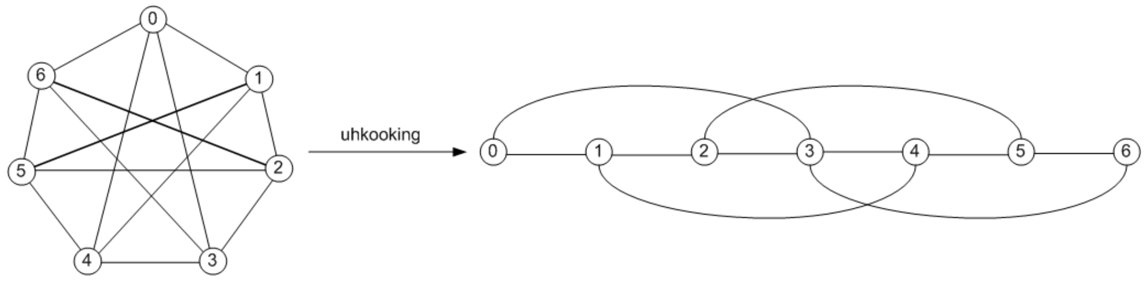


Figure 1. Example of $C_7^{1,3}$ and its step graph.

As we can see from the above definition, step graph L_n is obtained from circulant graph C_n by removing all edges that cross the interval $(n-1,0)$. Let us denote this set of edges by

$$Hook(n) = E_C(n) - E_L(n) = \bigcup_{i=1}^{k} \{(n-j, s_i - j) \mid 1 \le j \le s_i\}$$

It will be useful to have the following notation $W(n) = \{0,1,\dots,s_k - 1\} \cup \{n - s_k, n - s_k + 1, \dots, n-1\}$. From here we have that both ends from hook edges are in this set.

Let H be an Hamiltonian cycle of C_n. When we remove hook edges from H we can get two things:

a) a Hamiltonian cycle of L_n
b) a collection of disjoint paths (an isolated vertex is considered as a path) which represents a partition of the vertices (ends are elements of $W(n)$)

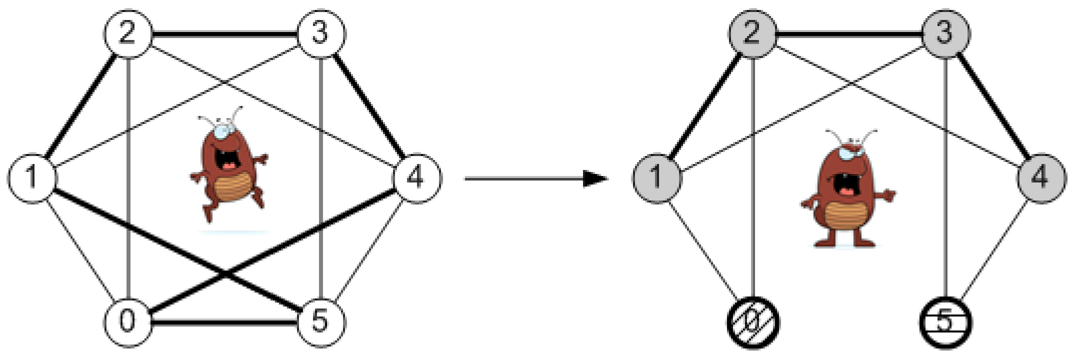


Figure 2. Decomposition of the Hamiltonian cycle in $C_6^{1,2}$

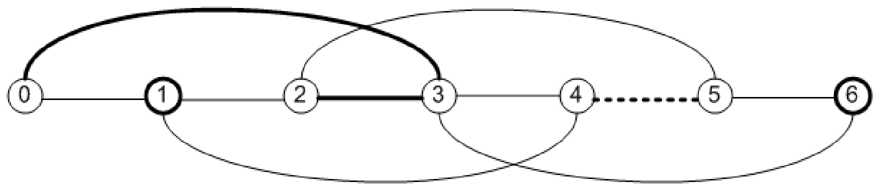
It is quite difficult to count the number of Hamiltonian cycles in C_n. So let's try a different approach. With this unhooking technique, we want to decompose a Hamiltonian cycle in the above path collection and see how we can connect these paths with edges from the hook set. For this idea we will need to use the following notation:

- Legal path decomposition (LPD) in L_n is either
  - a Hamiltonian cycle of L_n
  - a collection of disjoint paths in L_n such that end vertices are from set $W(n)$ and every vertex belongs to exactly one of the paths

- If D is LPD of L_n, then with $C(D)$ we denote the set
  $$C(D) = \{(u_1, v_1), \dots, (u_k, v_k), (w_1), \dots, (w_t)\}$$
  where (u_i, v_i) represents the end vertices of the i-th path in D, and (w_i) are isolated vertices in D.

- With P we will denote the collection of all set partitions of subsets of $W(n)$ where each element of the set partition is of size 1 or 2 (this partition will represent end points for LPD).

- For $X \in P$ let
  $$H_x(n) = |\{D : D \text{ is LPD of } L_n \text{ with } C(D) = X\}|$$
  In other words, $H_x(n)$ is the number of LPDs with ends X.



$W(7) = \{0,1,2,4,5,6\}$

$D = \{\{0,3,2\},\{1\},\{4,5\},\{6\}\}$

$C(D) = \{(0,2),(4,5),(1),(6)\}$

Figure 3. One example of LPD in $L_7^{1,3}$.

Now we can define vectors β and $\overrightarrow{H(L_n)}$ as

- $\beta = (\beta_X)_{X\in P}$ - where β_X represents the number of edge subsets of $Hook(n)$ for which adding it to some LPD D with $C(D) = X$ yields a Hamiltonian cycle in C_n.
- $\overrightarrow{H(L_n)} = (H(L_n)_X)_{X\in P}$ - where $H(L_n)_X$ represents the number of different LPDs for which $C(D) = X$

With the above definition we can count the number of Hamiltonian cycles in graph C_n through L_n:

$$H(C_n) = \beta \cdot \overrightarrow{H(L_n)}$$

This is a direct corollary of the above definitions. The only constraint for this is that $n > 2 \cdot s_k$.

How we have a new problem of initializing the vectors β and $\overrightarrow{H(L_n)}$. Vector β does not depend on n, so it can be computed by some backtracking algorithm. The main problem is $\overrightarrow{H(L_n)}$. For this we will need one more set

$$T = E_L(n+1) - E_L(n)$$

With this set we can define a recurrent relation between these vectors. Let A be square matrix where

$$a_{X',X} = \text{number of subsets } U \subset T \text{ for which if adding } U \text{ to LPD } D \text{ with } C(D) = X'$$

$$\text{yields to LPD } D' \text{ with } C(D') = X'$$

We can establish the recurrent relation

$$\overrightarrow{H(L_{n+1})} = A \cdot \overrightarrow{H(L_n)}$$

Finally, with the above recurrent relation and the relation with C_n and L_n we get

$$H(C_n) = \beta \cdot A^{n-2\cdot s_k} \cdot \overrightarrow{H(L_{2\cdot s_k+1})}$$

In [1] the authors have given an example of the above vertices and matrices for $C_n^{1,2}$. We will strongly recommend for reader to go through this example first (because of the many definitions that we have given here).

In our case, for the graph $C_n^{1,3}$ cardinality of the set P is 243. Matrix A is binary and the sum of its elements is only 295. We can save matrix A in our code as sparse matrix. With vectors β and $H(L_7)$ we will need to hardcode only ~ 800 integers.

The complexity of this algorithm (after computing the needed values) is $O(243^3 \cdot \log n)$. We can speed up this algorithm relying on the matrix properties but usual multiplication and logarithmic powering will pass the Timus tests.

References:

[1] Mordecai J. Golin, Yiu C. Leung, *Unhooking Circulant Graphs: A Combinatorial Method for Counting Spanning Trees, Hamiltonian Cycles and other Parameters*, Proceedings of the 30'th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'2004) (2004).
[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to algorithms, MIT Press (2009)

Solution by:
Name: ***Andreja Ilić***
School: Faculty of Mathematics and Sciences, Niš
E-mail: andrejko.ilic@gmail.com

Problem R2 09: Fair Fishermen (ID: 1818)

Time Limit: 1.0 second
Memory Limit: 64 MB

Fishermen caught a lot of fishes and, of course, they drank. In the morning it was the time to divide fishes. The first who woke up counted the fishes and it happened that in order to divide fishes equally he should throw away a_1 fishes. So he did that: threw away a_1 fishes and took his part. The second who woke up didn't know that the first fisherman took his part. He behaved the same as the first one: threw away a_2 excess fishes and took his part. The same story happened with the rest of the fishermen.

Given the amount of fishes thrown away by each fisherman, find the minimal possible number of fishes they caught. It is known that they caught at least one fish.

Input

The first line contains an integer $n (2 \leq n \leq 2000)$. The second line contains integers $a_1, a_2, \dots, a_n$ $(0 \leq a_i \leq n-1)$, where a_i is the number of fishes thrown away by the i-th fisherman.

Output

Output the minimal number of fishes the fishermen had to catch.

Sample

input	output
2 1 1	3
3 1 0 2	19

Solution:

This problem requires some knowledge about number theory. The easiest way to solve it is to analyze it "backwards": suppose that after the last fisherman took his part, there were *x* fish left (for the given conditions, it follows that *x* is a non-negative *integer*). Let b_n be the number of fish before the *n*-th fisherman woke up. Since he discarded a_n fishes and took an equal share, we have $(b_n - a_n) - \frac{1}{n}(b_n - a_n) = x$ and it follows $b_n = x\frac{n}{n-1} + a_n$. Analogously, the number of fish before $(n-1)-th$ fisherman woke up was $b_{n-1} = \left(x\frac{n}{n-1} + a_n\right)\frac{n}{n-1} + a_{n-1}$ etc. Going to the very first one, we get that the number of fish they caught is

$$X = \left(\left(\dots\left(\left(x\frac{n}{n-1} + a_n\right)\frac{n}{n-1} + a_{n-1}\right)\frac{n}{n-1} + a_{n-3}\right)\dots\right)\frac{n}{n-1} + a_1$$

One can easily see that the above expression is actually a polynomial function of degree *n* of variable $\frac{n}{n-1}$ and coefficients $x, a_n, a_{n-1,} \dots a_1$. Therefore, after extracting the common denominator, we get

$$X = \frac{xn^n + a_n(n-1)n^{n-1} + a_{n-1}(n-1)^2n^{n-2} + \dots + a_1(n-1)^n}{(n-1)^n} = \frac{xn^n + \sum_{i=1}^{n} a_i(n-1)^{n-i+1}n^{i-1}}{(n-1)^n}$$

Lets simplify things with $SUM = \sum_{i=1}^{n} a_i(n-1)^{n-i+1}n^{i-1}$ and $M = (n-1)^n$. Now we can write

$$X = \frac{xn^n + SUM}{M}$$

Since *X* is a positive integer, it is obvious that *x* cannot be just any integer. The necessary condition for *x* to be number of leftover fish is that $M \mid (xn^n + SUM)$, but **is it sufficient**? Remember that *all* b_i must be integers too. Luckily, it turns out that **it is**. Indeed, since n and $n-1$ are **relatively prime**, it means $M \mid (xn^n + SUM) \Rightarrow b_1 = (X - a_1)\frac{n-1}{n} \in N_0$ (directly from first *X*-equation). The rest can be proven analogously by induction.

It follows that, in order to minimize *X*, we must find minimal *x* which satisfies

$$xn^n \equiv -SUM \ (mod\ M)$$

It is enough to look for it in $\{0, 1, \dots M-1\}$ i.e. *modulo M* since if *x* is a solution, than so is $x + i \cdot M$ for all $i \in N$. This is a well-known **modular linear equation** of the form

$$ax \equiv b \ (mod\ c)$$

It is known that this equation has a solution if and only if $\gcd(a, c) \mid b$. Also if $\gcd(a, c) = 1$, the solution is unique (modulo *c*). In our case, $a = n^n$ and $c = (n-1)^n$ and since $\gcd(n^n, (n-1)^n) = 1$, our equation *has unique solution modulo M*. To find it, one can use the **extended Euclid's algorithm**. As its name suggests, this is an extension of the well-known algorithm for computing $\gcd(a, c)$ in worst-case logarithmic time, which returns some extra parameters (if $\gcd(a, c) = 1$, it computes the **multiplicative inverse of *a* modulo *c***). For more information on modular linear equations and the Extended-Euclid algorithm it is recommended to read Chapter 31 in [1].

With this, the problem is solved. However, there are some things we should consider while coding:

- *M, SUM* and n^n can have a few thousand digits. Therefore, Big Integer arithmetic is required. It is also required to implement Big Integer division with remainder for Euclid's algorithm. It is much easier if Java's BigInteger type is used.
- The most time-consuming part of algorithm is calculation of the *SUM*. All numbers n^i and $(n-1)^i$ should be pre-calculated and SUM should be calculated **iteratively**, like a polynomial. This leads to an $O(n^2)$ solution. If every term of SUM is calculated separately, it will lead to $O(n^3)$ solution and TLE errors.
- There is a special case where $a_1 = a_2 = \dots = a_n = 0$. In that case $SUM = 0$ but *x* can't be 0 since "It is known that they caught at least one fish". Therefore *x* must be M and $X = \frac{xn^n + SUM}{M} = n^n$.

The overall complexity is $\boldsymbol{O(n^2)}$, for calculating $\boldsymbol{SUM}$.

Solution by:
Name: ***Nikola Milosavljević***
School: Faculty of Mathematics, University of Niš
E-mail: nikola5000@gmail.com

Problem R2 10: Professional Approach (ID: 1819)

Time Limit: 2.0 second

Memory Limit: 64 MB

Denis, Eugene and Misha take professional approach to ACM ICPC. They don't have any common interests and communicate with each other only during the competitions. Recently they arrived in Saint Petersburg to participate in the regional contest and haven't seen each other yet. At morning before the contest they will leave their hotel at different times and will go to the contest site, Anichkov palace. Help them find out if there are such three paths from the hotel to Anichkov palace, that no two of them share a common road segment.

Input

The first line contains integers n and k ($2 \leq n \leq 50.000$, $1 \leq k \leq 50.000$), which are the number of crossroads and the number of road segments in Saint Petersburg, respectively. Crossroads are numbered with integers from 1 to n. Each of the following k lines contains two different integers, which are the numbers of crossroads, connected by a road segment. All road segments are bidirectional. There is at most one road segment between any two crossroads. The next line contains the number of test cases q ($1 \leq q \leq 50.000$). Each of the following q lines contains two different integers, which are the numbers of crossroads, where the hotel and Anichkov palace are situated, respectively.

Output

For each test case output "Yes" if there are three such paths that no two of them share a common road segment. Otherwise, output "No".

Sample

input	output

```
6 9
1 2
1 5
1 4
1 6
2 3
3 4
3 5
4 5
4 6
9
1 2
1 3
1 5
2 4
5 6
3 6
3 4
2 6
2 3
```
	```
	No
	Yes
	Yes
	No
	No
	No
	Yes
	No
	No
	```

Solution:

Professional Approach is a very challenging graph problem. By Menger's theorem (Karl Menger, 1927), the maximum number of edge-independent paths from node *a* to node *b* is equal to the size of minimum edge cut for *a* and *b*. That means that the minimum number of edges whose removal disconnects the graph is equal to the maximum number of edge-independent paths. A graph is *k-edge-connected* if it remains connected whenever fewer than *k* edges are removed.

In this problem, we have to find all **3-edge-connected components** in order to answer the queries in constant time. If the vertices a and b belong to the same 3-edge-connected component, there exist 3 edge-independent paths between vertices a and b.

Hopcroft and Tarjan presented a linear time algorithm for finding three-connected components in a graph, but that one deals with vertex connectivity. The first linear time algorithm for 3-edge-connected components was presented by Galil and Italiano in 1991, reducing the edge connectivity to vertex connectivity and then finding triconnected components. Since then, several other algorithms for this problem have been presented, but all of them have multiple depth-first search passes and lack plain simplicity. Just a few years ago, a paper called *A Simple 3-Edge-Connected Component Algorithm* by Yung H. Tsin was published, where he describes an algorithm that uses only one depth-first search over the graph to find 3-edge-connected components in linear time. The algorithm assumes that the graph is bridgeless, however it can easily be modified to include the detection of bridges, or the well-known linear time algorithm for finding all the bridges of a graph by Tarjan can be run beforehand.

Tsin's algorithm is based on the so-called *absorb-eject* operation. By using this operation, the original graph is transformed into a new edgeless graph, where each vertex represents a set of vertices from the original graph – 3-edge-connected component.

When the absorb-eject operation is applied to an edge $e = (v, u)$, vertex u is made an isolated vertex (3-edge-connected component) if $deg(u) = 2$, otherwise it is absorbed by the vertex v. When the vertex u is absorbed by the vertex v, all the edges incident upon the vertex u become incident upon the vertex v instead, while the self-loops are removed.

The depth-first-search starts at an arbitrary vertex r, and assigns a depth-first search number at each vertex v, denoted by $pre(v)$. Also, at each vertex v, $lowpt(v)$ is calculated as

$$lowpt(v) = min(\{lowpt(u) | u\ is\ a\ child\ of\ v\} \cup \{pre(v')|(v, v')is\ a\ back-edge\} \cup \{pre(v)\})$$

During the depth-first search, the so-called w-path is determined for the vertex w. When the search backtracks from vertex w to its parent v, the subtree rooted at w has been transformed into a set of isolated vertices and a w-path.

When the search first enters vertex w, w-path is initialized to null path. When the search backtracks from a child u of the vertex w (encounters a back-edge (w, u), respectively), if $lowpt(u)$ ($pre(u)$, respectively) is smaller than the current $lowpt(w)$, then the u-path is extended to include the tree-edge (w, u) (null path, respectively). The current w-path is then absorbed by the vertex w, and the new w-path becomes u-path. Otherwise, the current w-path remains unchanged, while the u-path and the tree-edge (w, u) are absorbed by the vertex w. In any of the cases, if $deg(u) = 2$ then vertex u is made an isolated vertex by the absorb-eject operation. When the search encounters a back-edge $e = (w, u)$ such that $pre(w) < pre(u)$, vertex u has to lie on a w-path, since the subtree rooted at a child of w containing vertex u has been traversed and u is not an isolated vertex because of the existence of edge e. The absorb-eject operation is applied at w to absorb the section of the current w-path from w to u. The final w-path is determined when the depth-first search backtracks to the parent of w.

For more information and proof of the algorithm, check the paper mentioned above: *A Simple 3-Edge-Connected Component Algorithm* by Yung H. Tsin.

Solution by:
Name: ***Vanja Petrović Tanković***
School: Faculty of Computing, Belgrade
E-mail: vpetrovictankovic@gmail.com

The scientific committee would like to thank everyone
who did important behind-the-scenes work.
We couldn't have done it without you.

If you think this adventure was exciting,
then you should get ready for speed up next year!
Stay curious...

Bubble Cup Crew

bubble cup 4

bubblecup.org