Published with the financial support of
the Ministry of Economy,
PKO Bank Polski,
the PKO Bank Polski Foundation
and the OFEK Foundation

Warsaw 2012
ISBN 978-83-920897-2-8
Copyright © by
the Faculty of Mathematics, Informatics and Mechanics,
University of Warsaw

*Editors* Krzysztof Diks, Tomasz Idziaszek, Jakub Łącki, Jakub Radoszewski
*Copy editor* Richard G. Hallas
*Additional translations* Justyna Diks, Jerzy Jaromczyk
*Design and typography* Emilka Bojańczyk / Podpunkt
*Illustrations* Emilka Bojańczyk, Diana Gawronkiewicz / Podpunkt
*Print* Perfekt SA

*Fonts* The book is set in Foral Pro and FdSymbol

# Contents

Introduction ..... 9

**SZYMON ACEDAŃSKI**
Ants ..... 18

**MARCIN ANDRYCHOWICZ**
Hyperclock ..... 25
Fishes ..... 31

**PIOTR CHRZĄSTOWSKI**
Pilots ..... 41
Skiers ..... 53

**MAREK CYGAN**
Barricades ..... 63
Travel Agency ..... 68

**TOMASZ CZAJKA**
Mushrooms ..... 77
Sweets ..... 83

**KRZYSZTOF DIKS**
Coding of Permutations ..... 91
Rooks ..... 102

**ANDRZEJ GĄSIENICA-SAMEK**
Monkeys ..... 108

**TOMASZ IDZIASZEK**
Plotter ..... 116
Termites ..... 123

**GRZEGORZ JAKACKI**
Window ..... 132
Altars ..... 138

**TOMASZ KOCIUMAKA**
Axes of Symmetry ..... 151
Leonardo Numbers ..... 157

**ERYK KOPCZYŃSKI**
Ritual ..... 170
Questions ..... 176

**MARCIN KUBICA**
Chocolate ..... 186
Fibonacci Sums ..... 193

**TOMASZ KULCZYŃSKI**
The Search ..... 202
Hashing ..... 209

**JAKUB ŁĄCKI**
Afternoon Tea ..... 216
Kangaroos ..... 220

**KRZYSZTOF ONAK**
Sums ..... 235
Superknight ..... 240

**JAKUB PACHOCKI**
Ice Skates ..... 251
Termites 2 ..... 256

**PAWEŁ PARYS**
Cave ..... 264
Shuffle ..... 272

**JAKUB PAWLEWICZ**
Game of Tokens ..... 277
Straight Lines ..... 281

**MARCIN PILIPCZUK**
Guilds ..... 293
Reconstruction of Byteland ..... 302

**MICHAŁ PILIPCZUK**
Army Training ..... 307
Riddle ..... 315

**JAKUB RADOSZEWSKI**
Fuel ..... 323
Ploughing ..... 330

**WOJCIECH RYTTER**
Canoes ..... 338
Painter's Studio ..... 342

**KRZYSZTOF STENCEL**
Triangles ..... 352
Triangles 2 ..... 353
Circular Game ..... 360

**WOJCIECH ŚMIETANKA**
Byteland ..... 368
Cakes ..... 374

**TOMASZ WALEŃ**
Building Blocks ..... 379
Matching ..... 387

**JAKUB WOJTASZCZYK**
Party ..... 395
Dice ..... 403

**FILIP WOLSKI**
Two Parties ..... 409
Guesswork ..... 416

# Introduction

This book outlines the most significant moments and achievements in the almost 20-year history of Polish algorithmic and programming contests organized or co-organized by staff and students of the Faculty of Mathematics, Informatics and Mechanics at the University of Warsaw. The history of competitions in Poland is all about the people—both organizers and contestants. The fact that Poland was chosen to host the International Olympiad in Informatics in 2005, and the 2012 World Finals of the ACM International Collegiate Programming Contest, says a lot about the quality of programming contests that have been held in Poland over the years.

Participants in Polish competitions have numerous international achievements on their résumés. The most important ones include: first place for Filip Wolski and Tomasz Kulczyński in the International Olympiad in Informatics in 2006 and 2007 respectively, as well as the two-time victory of the University of Warsaw in the World Finals of the ACM International Collegiate Programming Contest thanks to Tomasz Czajka, Andrzej Gąsienica-Samek and Krzysztof Onak in 2003 and Marek Cygan, Marcin Pilipczuk and Filip Wolski in 2007. Polish contestants have had their fair share of successes—including victories—in other competitions as well, such as TopCoder Open, Google Code Jam, Facebook Hacker Cup and Microsoft Imagine Cup. These achievements would not have been possible without the level of challenge presented by programming contests held in Poland, which have enabled talents to be identified and given adequate opportunity for development.

**Competitions in informatics**

The role of competitions in selecting and educating particularly gifted high school and university students cannot be overestimated. These educational events require knowledge and skills greatly exceeding what is taught in schools and universities. A good competition is one that relates to the core of the discipline it concerns, and knowledge and skills acquired through participation therein are not volatile but constitute grounds for further field-related development. What is of even greater importance is for the competition to help participants develop skills that will be useful in their future professional lives: diligence, constant striving for success and self-development, self-discipline, hunger for knowledge, self-improvement, teamwork, honesty, ambition, eagerness to compete, dedication. Taking part in a good competition should pose an intellectual challenge to a young individual, whilst success should be a reason for pride and recognition.

Contest tasks say a lot about the quality of a competition itself. They should be original, engaging and of different levels of difficulty. Finding a solution should cause the contestant to feel great satisfaction, whereas being unable to solve a given task should encourage an individual to broaden their knowledge and develop new skills. This book contains the best tasks from algorithmic and programming competitions organized or co-organized by the University of Warsaw, together with their exemplary solutions. The selection of the tasks was undertaken by people who have played significant roles in the history of Polish algorithmic and programming competitions as their participants or organizers. All of the authors of the texts presented in this book are closely affiliated with the Faculty of Mathematics, Informatics and Mechanics at the University of Warsaw, whether as former or current students or as academic staff.

Each of the tasks presented and discussed in this book was used during one of the following events: Polish Olympiad in Informatics, Junior Polish Olympiad in Informatics, Polish Olympiad in Informatics Training Camp, Central European Olympiad in Informatics, Polish Collegiate Programming Contest and Algorithmic Engagements.

*

The **Polish Olympiad in Informatics** was established in 1993 by the Institute of Computer Science at the University of Wroclaw, then chaired by Professor Maciej Sysło. The main founders of the event included four staff members from the University of Warsaw: Piotr Chrząstowski-Wachtel, Jan Madey (director of the Institute of Informatics at that time), Wojciech Rytter and Stanisław Waligórski (who became the first chairman of the Main Committee of the Polish Olympiad in Informatics). Since the very beginning it was the University of Warsaw that facilitated the event, in terms of both scientific background and technical solutions. All authors of texts presented in this book were, or still are, affiliated with the Olympiad.

The Olympiad in Informatics is a competition for high school students. The first four winners of national programming competitions represent their countries at the International Olympiad in Informatics.

The Polish Olympiad in Informatics consists of three stages. The first stage is usually organized in October or November and attracts around a thousand contestants, who are presented with five tasks to be solved at home. Completed solutions are then sent for evaluation to the competition organizers over the Internet. About 400 contestants qualify for the second stage of the competition, which lasts for three days and is held in a few selected regional centers that cooperate closely with the universities that offer the best computer science programs in Poland. The purpose of the first day is to acquaint the participants with the rules and regulations of the contest. Over the following two days, participants are asked to solve two or three tasks on their own during a five-hour supervised session. The solutions are collected from all the regional centers in Poland and are evaluated in a common environment using the same checking methods. Around 80 authors of the best solutions will be selected for the Olympiad's finals. This third stage is organized in one place and lasts for five days: there are three competition days and two days for recreation and sightseeing. The finals are very similar to the second stage of the competition. The resulting four best contestants will represent Poland in international programming competitions, including the International Olympiad in Informatics.

To date, 19 meetings of the Polish Olympiad in Informatics have presented 15,000 students with a total of 300 tasks to solve.

*

The **International Olympiad in Informatics** takes place every summer and gathers together the best high school programmers in the world. The first International Olympiad in Informatics took place in 1989. Since then, Polish students have won 31 gold, 28 silver and 22 bronze medals.

Academic staff and students from the University of Warsaw have actively participated in organizing international programming competitions affiliated with the Olympiad in Informatics: the **International Olympiad in Informatics** itself (IOI 2005), three meetings of the **Central European Olympiad in Informatics** (CEOI 1997, 2004, 2011) and two instances of the **Baltic Olympiad in Informatics** (BOI 2001, 2008).

Each year, finalists in the Polish Olympiad in Informatics have the opportunity to participate in the **Polish Olympiad in Informatics Training Camp**, where they get to solve and thoroughly discuss algorithmic and programming tasks prepared mostly by older students from the University of Warsaw.

For the past six years, yet another event has been organized regularly, namely the **Junior Polish Olympiad in Informatics**—a competition for junior high school students. The idea for the Junior Olympiad came from the great teacher Ryszard Szubartowski, and the Institute of Informatics at the University of Warsaw was actively involved in launching and developing the contest.

*

Many finalists of the Olympiad in Informatics choose the University of Warsaw as the place to further their education in informatics. The University allows its students to develop their competitive drive by trying out for a spot in the team representing the University of Warsaw in team programming competitions, including the **ACM International Collegiate Programming Contest**. The ACM-ICPC is the oldest and most prestigious computer science competition in the world, and it is considered to be a world championship in team programming. Each team consists of three students representing a single university. Regionals are the first stage of the competition; a few dozen regional eliminations are organized on all inhabited continents. The best teams selected during the regionals (including winners and a few runner-up teams, depending on the strength of the region and the number of competing teams) advance to the finals.

Both regionals and finals have the same format: each team, composed of three members, is presented with one computer and given five hours in which to solve between eight and twelve tasks. The solutions proposed by the contestants are evaluated in real time, after which the teams receive post-evaluation feedback in the form of a short but clear communication: *accepted*; *run-time error*; *time limit exceeded*; *wrong answer*; *presentation error*. All tasks are evaluated individually and marked as accepted or rejected. The team that completes the most tasks successfully wins the competition. In the event that several teams complete the same number of tasks, they are ranked based on the total time spent solving the tasks, shorter times being better. However, each rejected submission of a completed task results in a 20-minute time penalty.

The history of the ACM-ICPC dates back to 1977 and is closely intertwined with the work of Bill Poucher, the originator and director of the competition. The contest was popularized among Polish students and academic staff by Professor Jan Madey, who formed the first team representing the University of Warsaw in the competition in 1994. The team was sent to Amsterdam to compete in the eliminations, and succeeded in getting through. As regional winners, these University of Warsaw students were invited to the finals in Nashville, where they came 11th. Ever since then, teams from the University of Warsaw have qualified for the ACM-ICPC World Finals every year, and have won the entire competition twice: in 2003 and 2007. In 2012 the University of Warsaw has the pleasure of hosting the finals of the ACM-ICPC.

It is worth mentioning that the University already has some experience in organizing large-scale events like this. Between 2001 and 2003 the University of Warsaw hosted the **Central European Regional Contest**—an event that serves as eliminations for ACM-ICPC. In 1998–2001 and 2011 the University organized the **Polish Collegiate Programming Contest**.

*

In 2001, during the course of preparations for the Baltic Olympiad in Informatics, a new programming competition, Algorithm Busters, was launched. In 2005 it changed its name to **Algorithmic Engagements**. Algorithmic Engagements is an individual competition for any person of any age and professional status who is willing to sign up for it. The only things that really matter are each contestant's knowledge and skills. The competition consists of two stages. The first is organized via the Internet, lasts for about a week, and is divided into five or six rounds. During each of the rounds, contestants are asked to solve between one and four tasks. The level of difficulty increases from one round to the next. The top 20 participants from the first stage are invited to regular finals that have a format similar to those of the ACM-ICPC (the only exception being that the contestants solve tasks on their own rather than in teams). The winner

receives the title of Algorithm Master of the Year. Algorithmic Engagements has become quite popular over the years; every year a few thousand people sign up for the event, and some of them even update their CVs with their competition achievements.

*

As one may imagine, over the course of many competitions we have organized in the past, hundreds of original programming tasks have been invented and properly described. In order to "keep them alive", we do our best to upload as many of them as possible to an educational portal called **MAIN** (to be found at http://main.edu.pl), which was created at the initiative of organizers of the Polish Olympiad in Informatics and is administered by academic staff and students of the University of Warsaw. MAIN's archive is integrated with the computer system of the Polish Olympiad in Informatics, called **SIO**, which enables solutions to tasks uploaded on the portal to be evaluated in real time, 24/7. Readers of this book can prepare their own solutions to tasks presented herein and then have them checked on the MAIN portal. Good luck!

**About the authors**

The authors of the texts presented in this book are closely affiliated with the University of Warsaw and have had a significant impact on the course of the history of algorithmic and programming competitions organized or co-organized by the University. Due to the nature of this book, we have limited ourselves solely to people who have experience in writing and solving competition tasks. The authors include lecturers and students, former contestants with international successes in programming competitions, educators and popularizers of informatics.

Each author was asked to choose the two tasks that they found the most interesting from competitions co-organized by the University of Warsaw, and to discuss their solutions. The choice was based on the level of difficulty of the tasks, the methods and techniques of solving them, their educational value or the particular author's personal memories of a specific task. Each task and description of its solution is preceded by a short biography of the author, illustrating the links between their careers and programming competitions as well as their scientific and private interests.



**Acknowledgements**

In the pages of this book we have only been able to honor a mere 30 individuals who have had a significant impact on the development of programming competitions in Poland—thereby contributing at the same time to the popularization of informatics and the education of many talented computer scientists. However, I would like to extend my thanks to everyone who has made it possible for this book to be published. I wish to thank the authors of the texts presented in this book, who managed to submit their extracts despite a very tight deadline. Special thanks should go to those who not only authored some of the texts in this book but also used their subject-matter expertise to edit the whole publication: Tomasz Idziaszek, Jakub Łącki and Jakub Radoszewski. Without their engagement and editorial hard work this book would have had no chance of being published. During the preparation of the English translation of the book we received help from my daughter Justyna Diks and my friend Jerzy Jaromczyk, assisted by Neil Moore. For the copy-editing of the English version, my special gratitude goes to Richard Hallas, who was supported in his work by Grzegorz Jakacki and Marcin Kubica. I am also extremely grateful to graphic studio Podpunkt for the layout design of this book and for patience with editors during the process of typesetting material for publication. While creating this book we received a lot of support and encouragement from Rafał Sikorski—a lawyer with a keen interest in informatics.

Having this book published as it is—both in Polish and English—and released in a limited run of 1000 copies in each language would not have been possible but for the financial support of the Ministry of Economy, PKO Bank Polski and Fundacja PKO Banku Polskiego. I would like to take this opportunity to express my appreciation to the management of these institutions for acknowledging the significant educational value of this book and supporting such educational undertakings.

To all readers of this book: enjoy the read and the satisfaction of discovering the secrets of algorithms!

*Krzysztof Diks*

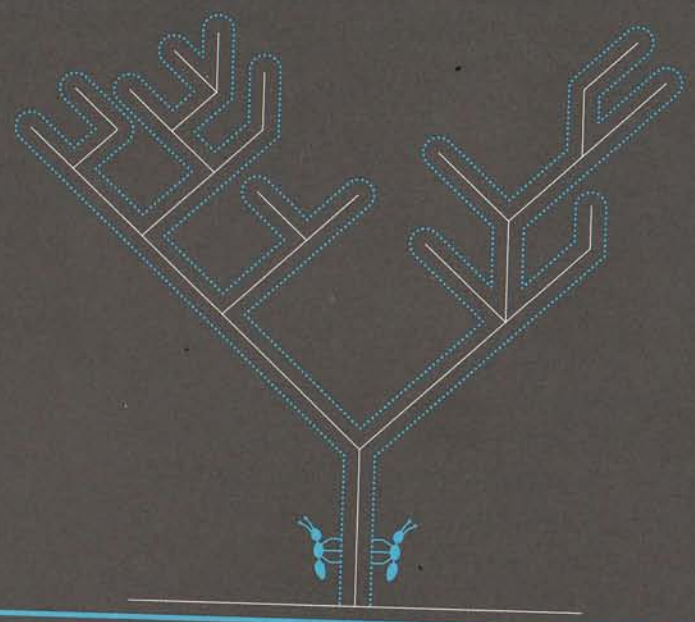
# The tasks >

**SZYMON ACEDAŃSKI**

Szymon Acedański is a Ph.D. student at the Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, and is interested in Natural Language Processing, especially morphosyntactic tagging and deep parsing. He cooperates with the Linguistic Engineering Group at the Polish Academy of Sciences, where he runs projects from the field. Szymon Acedański has won two gold medals at the Baltic Olympiad in Informatics. In 2004, his team won the ACM-ICPC Central European Regional Contest and represented the University of Warsaw at the ACM-ICPC World Finals in Shanghai, China. Szymon Acedański is the Technical Director of the Polish Olympiad in Informatics and a volunteer at the Polish Children's Fund, teaching at workshops for gifted high-school students every year. His hobbies include photography and, during warmer seasons of the year, sailing and traveling.

# / Ants

**Contest:** Polish Collegiate Programming Contest 2011
**Task author:** Szymon Acedański
**Memory:** 6 MB
http://main.edu.pl/en/archive/amppz/2011/drz



Computer geeks like trees. Ants also like trees. Therefore we are given a tree with two ants walking on it—the Left Ant and the Right Ant—as shown in the figure above. The ants walk along the path depicted by a dotted line. They start their journey at the lower end of the trunk, on opposite sides. The Left Ant needs 2 seconds to walk along a single edge of the tree if walking from the root (upwards), and 1 second if walking towards the root (downwards). The Right Ant is two times faster. When the two ants meet, they both turn around and start walking in the opposite direction. If either of the ants steps from the tree to the ground, it immediately starts to climb up the opposite side of the trunk. Apart from that, the ants are so tiny that they would not be visible even under a microscope (they are intentionally depicted much larger in the figure). Your task is to write a program that computes the moment when the ants turn around for the second time.



## Input

The first line of the input contains a single integer $t$ ($1 \le t \le 1000$) representing the number of test cases described in the input.

The description of each test case consists of two lines. The first line contains an even integer $n$ ($2 \le n \le 100{,}000{,}000$) specifying the number of edges in the tree. The second line holds a description of the tree. It is a string consisting of $\frac{n}{2}$ characters representing a $2n$-bit binary number written in hexadecimal format (using digits and small letters from $a$ to $f$). This number shows the Left Ant's path around the whole tree, assuming that the Right Ant stands still. The consecutive bits of this number (starting from the left) denote whether the Left Ant walks away from the root of the tree along the corresponding edge (bit = 1) or towards the root along this edge (bit = 0). The root has a trunk; that is, there is exactly one edge leading from the root of the tree.

The size of the input file does not exceed 50 MB, which is much more than the amount of memory available for your program.

## Output

Your program should output $t$ lines containing answers to the consecutive test cases. Each answer should represent the moment (in seconds) when the ants turn around for the second time, given as an irreducible fraction $p/q$ (without any white space around /), where $p$ and $q$ are positive integers. If the answer is integer then, obviously, $q = 1$.

## Example

For the input data:

1
28
fb1da30d1b7230

the correct result is:

282/5

**Explanation of the example:** The sample data corresponds to the figure above, and transforms to the following sequence of bits:

1111 1011 0001 1101 1010 0011 0000 1101 0001 1011 0111 0010 0011 0000

# / Solution

*Ants* is the only programming problem that I have successfully written with the aim of throwing something together quickly at the outset. All my earlier attempts at putting together a problem on request, with the expectation that it would be finished in a short time, had been futile. However, on that particular sunny afternoon, somewhere far away—thanks to the inspiring presence of my colleague Jakub Radoszewski, who announced that new problems were needed for the Polish Collegiate Programming Contest—I had this conversation with myself:

— Hmm, an exercise... Good. So a character is needed.
— Byteasar? Well... no, it's not for the Olympiad in Informatics... So perhaps some animal. Everyone likes animals.
— Let's choose from some typical candidates: termites; ants; fleas?
— Oh, ants! I recall a nice problem that I heard from Paweł Parys:

***Problem.*** Ants are walking on a branch that extends horizontally from the trunk of a tree. All of the ants move at the same speed. Whenever two of them meet, they turn around and continue walking in opposite directions. Ants that reach the tree trunk also turn around and begin ascending the branch again. An ant that gets to the tip of the branch falls down. Given the initial positions of the ants and their speed, compute the time when the last of the ants falls down*.

— So, what if the ants do not fall down from the tree, and the task is to compute their positions on the branch?
— Well, nothing exciting. They would be going back and forth. So perhaps some of them could walk faster than others?
— No, no, too hard! I myself can only come up with a brute force solution. So what if, in addition, we reduce the number of ants to, say, two?
— Well, that's too easy. What if a branch is replaced with something more complex?
— Oh, a tree would be ideal. Moreover, a natural one, with the root pointing down.
— Well, a tree just slightly obscures the fact that the ants are moving in a circle.
— So, at least let me make the "up" segments harder to traverse.

* The reader is strongly encouraged to try to tackle this problem. It has a beautiful solution.



— Sure, but what would be the task? Perhaps finding the meeting point for the moving ants.
— Oh, and I know how to solve it without reading in the representation of the entire tree. It's sufficient to simulate the movement of one of the ants. Knowing the number of "up" and "down" segments still remaining in the tree for the ant, I can also compute at each point the time needed for the other ant to traverse this remaining portion of the tree... yes, that's it.

I was tempted to stop at this point and claim victory. But wait:

— What if, instead, the problem were to ask for the *second* meeting point of the ants to be computed?

I must admit that this question intrigued me. I did not expect that the described problem would have surprising properties, though. Specifically, I was surprised that the following observation holds true:

***Observation.*** After the first meeting the Left Ant will always return to the root faster than the Right Ant.

It was the Right Ant that was supposed to move faster...

**Solution**

Dear reader, it is unlikely that the above story has brought you any closer to the solution! In fact, even the above observation seems somewhat suspicious. So I will try to explain the solution step by step. The input specifications are written so as to make it impossible to store the description of the entire tree in memory. The author's solution algorithm reads the description once without storing it. Here is a sketch of the solution:

1. We read in the size of the tree; it gives us the number of "up" and "down" segments.
2. While reading the description, we simulate the walk for the Left Ant until the first meeting with the Right Ant. For each edge, we can check if this is the place of the meeting since, knowing the distance traveled by the Left Ant, we know the distance the Right Ant would need to cover to the meeting point.
3. We continue reading in the input, this time simulating the walk of the Right Ant, until the Left Ant reaches the root.
4. From this point, we continue to simulate the walk of the Right Ant, until the second meeting point.

To see that the above sketch can be developed into a complete algorithm, let us take a look at a more formal description of the author's solution, written by Tomasz Idziaszek when he was preparing the problem for the contest.

We will use standard graph terminology for trees, with the exception that the root of the tree is at the bottom. A point on the tree can be described by a pair of real numbers $(a, k)$, where $a$ indicates the total number of edges traversed by the Left Ant from the root to this point (possibly traversing some edges in both directions, with those edges counted twice), and $k$ indicates the height of the point (the number of edges from the root to the point). If the ant is not at a vertex of the tree, $a$ and $k$ have non-integer parts indicating the portion of the ant's current edge that has been traversed, or that is below the ant, respectively. If it takes the Left Ant $t_g$ seconds to climb an "up" edge and $t_d$ seconds to descend along a "down" edge, then this ant reaches $(a, k)$ in

$$[a, k, t_g, t_d] := \frac{a+k}{2} t_g + \frac{a-k}{2} t_d$$

seconds. Therefore, the first meeting of the Left Ant with the Right Ant, $(a_1, k_1)$, happens when

$$[a_1, k_1, 2, 1] = [2n - a_1, k_1, 1, \tfrac{1}{2}]. \tag{1}$$

Again, we can find this point by simulating the walk of the Left Ant while reading the tree description edge by edge. When the Left Ant is at point $(a, k)$, which is the beginning of some edge, we can check if the meeting could happen on this edge itself. Let $b = 1$ if this is an "up" edge, and $b = -1$ otherwise. If the meeting is to happen after walking an $\varepsilon$ piece of the edge then (1) is satisfied for

$$a_1 = a + \varepsilon, \quad k_1 = k + b\varepsilon,$$

therefore

$$\varepsilon = \frac{6n - 9a - k}{9 + b}.$$

Thus, if $0 \le \varepsilon < 1$, then we have located the first meeting point. The point will be reached in

$$t_1 = [a_1, k_1, 2, 1] = \frac{3a_1 + k_1}{2}$$

seconds.



If we let the ants continue and return to the root, then the Left Ant would reach the root in $[a_1, -k_1, 2, 1] = t_1 - k_1$ seconds, and the Right Ant in

$$[2n - a_1, -k_1, 1, \tfrac{1}{2}] = t_1 - \frac{k_1}{2}$$

seconds. So we can see now that the Left Ant would be at the root before the Right Ant.

Now we can start looking for the second meeting point. We will do so by simulating the Right Ant's walk. After the Right Ant walks the distance $\varepsilon'$ on the edge starting at $(a', k')$, it reaches the second meeting point $(a_2, k_2)$, provided that

$$a_2 = a' + \varepsilon', \quad k_2 = k' + b'\varepsilon',$$

$$[a_1, -k_1, 2, 1] + [2n - a_2, k_2, 2, 1] = [a_2 - a_1, k_2 - k_1, 1, \tfrac{1}{2}],$$

thus

$$\varepsilon' = \frac{12n - 9(a' - a_1) + (k' - k_1)}{9 - b'}.$$

If $0 \le \varepsilon' < 1$, then we have found the second meeting point. The time to reach this point is

$$t_2 = [a_2 - a_1, k_2 - k_1, 1, \tfrac{1}{2}] = \frac{3(a_2 - a_1) + (k_2 - k_1)}{4}$$

seconds. Therefore the total time from the first to the second meeting of the ants equals

$$t_1 + t_2 = \frac{3(a_1 + a_2) + (k_1 + k_2)}{4}.$$

Note that $\varepsilon$ is a multiple of $\frac{1}{9+b}$, and simple algebraic manipulations reveal that $\varepsilon'$ is a multiple of $\frac{9-b}{(9+b)(9-b')}$. This implies that the numbers $a_1$, $a_2$, $k_1$, $k_2$ are multiples of $\frac{1}{800}$. Therefore, all calculations can be carried out using 64-bit integers representing values multiplied by 800.

All of this results in an algorithm whose every step consists of reading one consecutive edge from the input and executing a constant number of operations. The running time of the algorithm is thus $O(n)$ with constant memory.

**MARCIN ANDRYCHOWICZ**

Marcin Andrychowicz is a student at the Faculty of Mathematics, Informatics and Mechanics, University of Warsaw. In high school he won three gold medals at the International Olympiad in Informatics, ranked third at the TopCoder High School Tournament, and won Algorithmic Engagements in 2008. During his university years, he captained a team that won the ACM-ICPC Central European Regional Contest and earned a bronze medal at the ACM-ICPC World Finals in 2009.

Marcin Andrychowicz is interested in cryptographic protocols that are provably secure against physical attacks, and algorithmic trading. In his spare time, he enjoys mountain hiking, snowboarding and traveling.

# / Hyperclock

**Contest:** Polish Olympiad in Informatics Training Camp 2007
**Task author:** Jakub Pawlewicz
**Memory:** 32 MB
http://main.edu.pl/en/archive/ontak/2007/hip

Brainy Smurf concluded that days were too short, and, because of that, he would not manage to write all the volumes of *Quotations of Brainy Smurf*. He decided to visit Father Time and ask him to extend each day to 48 hours. Father Time didn't want to accept his request, but, after Brainy Smurf's importunate demands, eventually agreed. However, he imposed a condition: Brainy Smurf must first solve the Hyperclock puzzle.

The Hyperclock consists of $N$ clocks. Each clock has one hand and some numbers on its face. On the face of the $i$-th clock ($1 \le i \le N$) there are numbers from 1 to $k_i$. At the very beginning, all hands point to the number 1. The puzzle was explained by Father Time as follows:

*You have to perform a sequence of moves. In one move you can turn the hand of an arbitrary clock one position clockwise or counterclockwise. After the last move, the Hyperclock's hands must return to their initial positions. Moreover, each possible configuration of the Hyperclock must appear exactly once.*

Brainy Smurf has been tinkering with the Hyperclock for over an hour and is becoming increasingly convinced that, despite his wisdom, he will not manage to solve the puzzle. Help him!

## Input

The first row of the input contains one positive integer $N$. In each of the following $N$ rows there is one integer greater than 1: in the $(i+1)$-th row there is the number $k_i$. The number of all possible configurations (namely $k_1 k_2 \cdots k_N$) is not greater than $10^6$.

## Output

If there is no solution to the puzzle, output the single word NIE (Polish for *no*). Otherwise, $k_1 k_2 \cdots k_N$ rows should be output, describing the subsequent moves. Each row must contain two integers. The first number $i$ ($1 \le i \le N$) is the number of the clock whose hand is turned. The second number indicates a direction: 1 (for

clockwise) or −1 (for counterclockwise). If there are many possible solutions, you can output any of them.

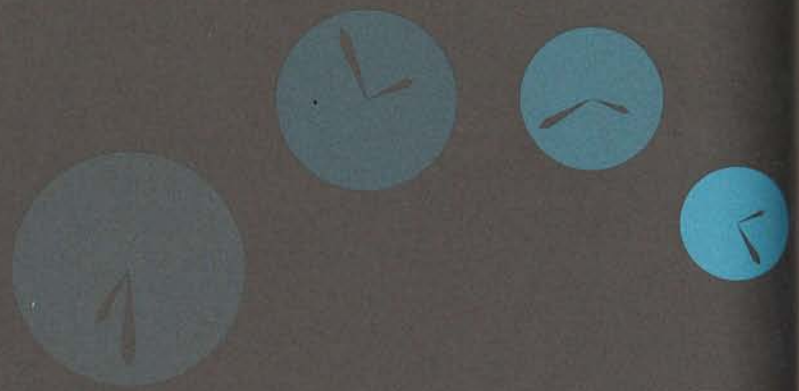
**Example**

For the input data:

3
2
2
3

the correct result is:

1 1
2 1
1 −1
3 1
1 1
2 −1
3 1
2 1
1 −1
2 −1
3 −1
3 −1

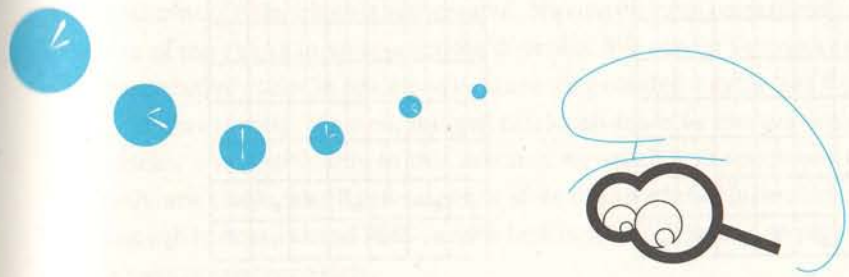




# / Solution

Let's start by expressing the task in mathematical terms. The Hyperclock's configuration can be described with a sequence of integers $x_1, x_2, \ldots, x_N$, indicating numbers being pointed to by individual clocks, where $x_i \in \{1, 2, \ldots, k_i\}$. In one step, we can turn one clock's hand one position clockwise or counterclockwise, so we can go from a configuration $x_1, x_2, \ldots, x_N$ to $y_1, y_2, \ldots, y_N$, if only there is an index $j$, such that $y_i = x_i$ for $i \neq j$ and $y_j = (x_j \pm 1) \bmod k_i$.

It is natural to interpret the Hyperclock as a graph, in which vertices correspond to possible configurations and edges to possible transitions in one move. How can we interpret the puzzle in terms of that graph? We are looking for a cycle going through each vertex exactly once, i.e. a *Hamiltonian cycle*. The problem of finding a Hamiltonian cycle in an arbitrary graph is an NP-complete problem. It means that it probably can't be solved in polynomial time. Our graph from the task has a specific form*, which allows it to be solved in linear time.



* Formally, this graph is a Cartesian product of simple cycles.

### Two clocks

Let's start by solving the task for the case where the Hyperclock consists of exactly two clocks. It is natural to visualize the situation as a $k_1 \times k_2$ grid. We can go from one field to another if they share a side; moreover, we can go from the last field of a row or column to the first one and *vice versa*.

Figure 1 presents the situation for $k_1 = 6$ and $k_2 = 9$ with the solution. The above idea can be easily generalized for situations where at least one of the sides has an even length. The following pseudocode corresponds to the picture, assuming that we start from the upper left corner.

```
Algorithm EvenSide(k1,k2)
  {we assume that 2 | k1}
  for i := 1 to k1/2 do
    for j := 1 to k2 − 1 do
      print 2 1 {down}
    print 1 1 {right}
    for j := 1 to k2 − 1 do
      print 2 −1 {up}
    print 1 1 {right}
```

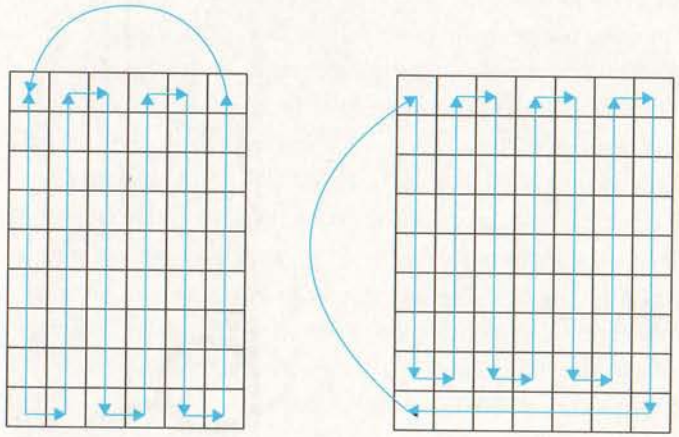


*Figure 1: Solution for $k_1 = 6$, $k_2 = 9$ (left) and for $k_1 = 7$, $k_2 = 9$ (right).*



Situations where both sides have odd lengths require only a minor modification of the above idea. In figure 1 there is a sample solution for $k_1 = 7$ and $k_2 = 9$. Again, the following pseudocode corresponds to the picture, assuming that we start from the upper left corner.

```
Algorithm OddSide(k1,k2)
  {we assume that 2 ∤ k1}
  for i := 1 to (k1 − 1)/2 do
    for j := 1 to k2 − 2 do
      print 2 1 {down}
    print 1 1 {right}
    for j := 1 to k2 − 2 do
      print 2 −1 {up}
    print 1 1 {right}
  for i := 1 to k2 − 1 do
    print 2 1 {down}
  for i := 1 to k1 − 1 do
    print 1 −1 {left}
  print 2 1 {down}
```

### More clocks

We already know how to cope with the problem for $N = 2$. It turns out that it is sufficient to solve the general version of the task. After finding a Hamiltonian cycle for the first two clocks, we can treat them as one clock with $k_1k_2$ states—it is enough to put the resulting Hamiltonian cycle on the face of the new clock!

The hand of the new clock points to a pair of numbers indicating the state of both clocks from which it was created. Moving its hand corresponds to moving one of the clocks in an appropriate direction. We can go between each pair of consecutive states in one move because they created a cycle (see figure 2).

Subsequently, the new, merged clock can again be merged with the third clock... and, continuing in this manner, we reach a situation where there is only one clock, which indicates the state of the whole Hyperclock. Then it is enough to turn its hand 360° and the task is solved. It means, in particular, that a solution always exists.

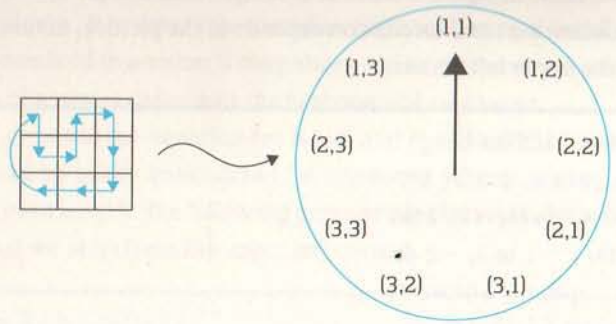



*Figure 2: Merging two clocks of length 3 into one.*

**The clocks' representation**

To implement the above idea, we need first to decide how we would store the created clocks. We don't really need to know the values on their faces—it is enough to realize what moves correspond to turning their hands clockwise. For instance, in the above example we need to know that moving from the first position to the second (from the state (1,1) to the state (1,2)) corresponds to turning the second clock clockwise (i.e. making a move 2 1) and so on. Therefore, a clock with $k$ states will be represented as an array of length $k$, where under the $i$-th index there is a pair $(a, b)$, such that $a \in \{1;2, \ldots, N\}$ and $b \in \{-1, 1\}$, indicating a move that has to be made to go from the $i$-th state to the $(i+1)$-th (or from the last one to the first).

This information is sufficient for turning the clock in both directions, because if a pair $(a, b)$ corresponds to the transition from the $i$-th state to the $(i+1)$-th, then a move from the $(i+1)$-th state to the $i$-th corresponds to the pair $(a, -b)$. Using the solution for two clocks, we can merge two clocks in time linear with the number of states in the clock being created. Let's estimate the total time of all $N-1$ merges. Let $M = k_1 k_2 \cdots k_N$. The whole solution works in time proportional to

$$k_1k_2 + k_1k_2k_3 + \cdots + M = M\left(\frac{1}{k_3k_4 \cdots k_N} + \frac{1}{k_4k_5 \cdots k_N} + \cdots + 1\right) \leq$$
$$\leq M\left(\frac{1}{2^{N-2}} + \frac{1}{2^{N-1}} + \cdots + 1\right) < 2M,$$

so it is linear with the input size.



# / Fishes

**Contest:** Algorithmic Engagements 2009
**Task author:** Eryk Kopczyński
**Memory:** 256 MB
http://main.edu.pl/en/archive/pa/2009/ryb

In an archipelago far away there lives a rare species of predator fish. These fish have a very regular rhythm of life. Each fish wakes up every morning at the same time of day and goes hunting. In the evening, it returns to the place from which it set off. It goes to sleep there at the same time every day, but it may wake up in a different place, as it can be moved a little by ocean currents.

Throughout the whole day, a fish sticks to the following rule: at every moment it has to be able to see where it was at the same time on the previous day; that is, exactly 24 hours earlier. Of course, a fish cannot see a point at the opposite side of any island.

Ichthyologists have been observing the fish in the archipelago for quite a long time, and every couple of days they have kept a record of one route taken by some fish. Unfortunately, after collecting a large amount of data, there has been an accident. Some of the data is now missing, and the rest is completely messed up. The scientists do not even know which fish traveled by each route they recorded. They have asked you for help. They are going to give you the messed up descriptions of the fishes' routes and ask you to tell them the minimum number of different fish that they have observed during their research.

**Input**

The input consists of two parts: a description of the archipelago and a description of the fishes' routes. The first line of the archipelago description contains two integers, $w$ and $h$ ($3 \leq w \leq 1000$, $3 \leq h \leq 1000$), separated by a single space. In each of the following $h$ lines there is a $w$-character-long string that describes a part of the archipelago. A character . represents ocean, whereas # represents land. There is water (. character) in all cells on the boundary of the map.

One point in the archipelago is visible from another if no segment connecting them has common points with the interior or boundary of any piece of land. The direction in which a fish is swimming does not matter here.

In the next line, there is an integer $n$ ($2 \leq n \leq 1000$) specifying the number of recorded fishes' routes. The following $2n$ lines contain descriptions of those

routes. In the first line of a route description, there are three integers, $x$, $y$, and $d$ ($1 \le x \le w$, $1 \le y \le h$, $2 \le d \le 10{,}000$), separated by single spaces. The numbers $x$ and $y$ are the coordinates of the place where a fish woke up (column and row) and $d$ is the length of the route. The second line is a string of $d$ characters, N, W, S or E, representing the direction of the fish. They stand for up, left, down and right respectively. It is guaranteed that the routes go only through cells containing water, that fish do not leave the fragment of archipelago described in the input, and that each route ends in the same cell from which it starts.

A fish can only swim horizontally or vertically; it moves along a broken line that connects the centers of the cells along its route. However, we do not know its speed. The fish can speed up or slow down so that it can always see the point it occupied exactly 24 hours earlier.

Ocean currents can move a sleeping fish by at most one cell up, down, left or right from the place where the fish went to sleep. You can assume that, for every two water cells in the archipelago, there exists some (hypothetical) fish route that goes through both of them.

**Output**

In the first line of the output your program should write an integer $k$ representing the smallest possible number of different fishes consistent with the data gathered by the scientists. Each of the following $k$ lines should contain a list of routes of a single fish. The fish *did not* necessarily need to swim along those routes *on consecutive days*, but on any two days of its life.

A fish can travel two routes *in two consecutive days*, if both routes start in the same cell or in neighboring cells (cells sharing an edge) and if the fish could swim along the second route, being able to see the point it occupied exactly 24 hours earlier at all times.

The routes are numbered from 1 to $n$, according to their order in the input. The numbers of routes in each line of the output should be written in increasing order, and the lines should be ordered in such a way that the first numbers in all lines form an increasing sequence.



**Example**

For the input data:

```
10  8
..........
........#..
.........#.
..........
...#......
......###.
......###.
..........
4
2  7  12
NNNEEESSSWWW
2  8  24
WNNNNNNNEEEEESSSSSSSWWWW
1  8  46
NNNNNNNEEEEESSSSSSSEEEENNNNNNNSSSSSSSWWWWWWWWW
1  8  32
NNNNNNNEEEEEEEESSSSSSSWWWWWWWWW
```

the correct result is:

```
2
1  2  3
4
```

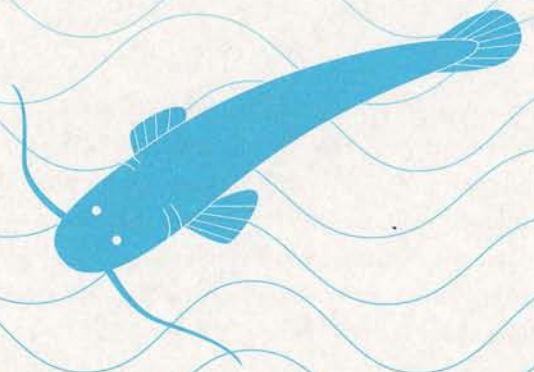
**Explanation of the example:** The first two routes could have been traveled by the same fish (even in two consecutive days). After a few days, the fish could have followed the third route. However, the last route must have been traveled by another fish. Unlike the first three routes, it goes around the big island.

# / Solution

The task *Fishes* is certainly not a standard one. It is the only task known to me that concerns topology. The fishes' behavior was described in the task statement in such a way that, after reading it, it is not intuitive when two routes could be traveled by one fish (not necessarily on consecutive days). It turns out that this condition can be expressed much more simply, and that it corresponds to a concept from topology. We will call such routes *equivalent*. The task is to compute the abstraction classes of routes' equivalence relation.

First notice that two routes with identical shapes and directions of swimming, differing only in the place where a fish woke up, are equivalent. This is because the fish could swim the same route every day with a constant velocity, starting each day from the next cell on the route (onto which it was moved during the night by the ocean's currents). Therefore, we can disregard the place where a fish wakes up and consider only the shapes of routes, which are directed loops.

Examining the example in figure 1, it can be guessed that two routes are equivalent when one can be continuously deformed into the other. In topology, such loops are called *homotopic*. The outline of the proof of this observation can be found at the end of the solution; knowledge of it is not necessary to understand the rest of the solution.





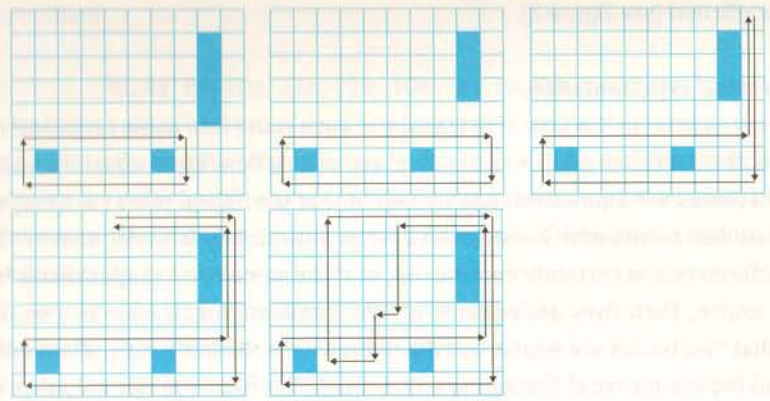


*Figure 1: An exemplary route that could be swum by one fish over consecutive days. The place where the fish wakes up is not marked, because it can be disregarded.*

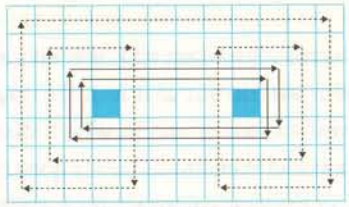


*Figure 2: Both routes circumnavigate both islands twice. Despite that, they are not equivalent.*

### The first idea

Let us consider when two routes are equivalent. It can be seen from figure 1 that such routes can have completely different shapes. In an archipelago where there are no islands, all routes are equivalent. However, if there is one island, a route circumnavigating it is not equivalent to another that doesn't go around it. Moreover, a route circumnavigating it twice is not equivalent to a route that does so only once. The direction of circumnavigation (clockwise or counterclockwise) is relevant too. The above remarks may lead to the idea that counting, in some way, how many times each route circumnavigates each island permits

nonequivalent routes to be distinguished. However, it turns out that such testing is insufficient (see figure 2).

### The routes' representation

The key to solving the task is to transform each route into some canonical representation. We would like two routes' representations to be equal if and only if two routes are equivalent. Let's imagine that the fishes' routes are changed into rubber bands which shrink and wrap around any islands' shores. That transformation is certainly continuous, so if the same band shapes result from two routes, then they are equivalent. The converse implication is also true, i.e. that two routes are equivalent if and only if the rubber bands' shapes after shrinking are identical (including a direction). The reader may verify that fact by drawing arbitrary routes and finding the band's shapes. We assume that routes which don't circumnavigate any islands shrink to a microscopic size and disappear, so they are all equivalent.

How can we simulate the process of shrinking? Let's begin by imagining that on each horizontal continuous fragment of the ocean there is a laser beam, which registers the fact of its crossing by a fish and the fish's direction of movement (see figure 3).

We assume that a fish crosses a beam whenever it swims into the field with the beam from the south or swims out of it into the south. For each route we can compute which beams it crosses and in which directions. Let's observe that if a fish crosses a beam and then immediately goes back, we can disregard both intersections, getting the equivalent route. It may cause another pair of intersections to become adjacent, and we are able to delete them too.

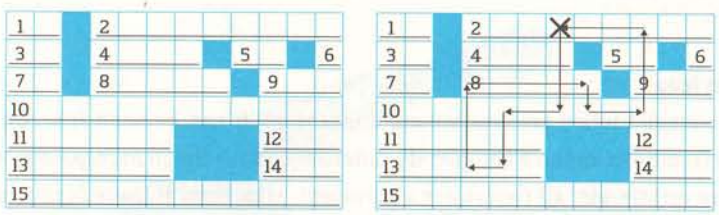



*Figure 3: Left: an archipelago with marked laser beams, which are additionally numbered with natural numbers. Right: an exemplary fish's route for which we find a sequence of intersections. A cell in which the fish wakes up is marked by a cross.*



For the example in figure 3, we would get the following sequence of intersections:

2S, 4S, 8S, 10S, 11S, 11N, 10N, 8N, 8S, 9N, 5N, 2N,

where N or S indicates that the fish swam, respectively, north or south. Erasing iteratively the adjacent intersections concerning the same beam, we get the following sequence of intersections (the pair that is deleted in each step is shown in bold):

1. 2S, 4S, 8S, 10S, **11S**, **11N**, 10N, 8N, 8S, 9N, 5N, 2N,
2. 2S, 4S, 8S, **10S**, **10N**, 8N, 8S, 9N, 5N, 2N,
3. 2S, 4S, **8S**, **8N**, 8S, 9N, 5N, 2N,
4. **2S**, 4S, 8S, 9N, 5N, **2N** (the route is cyclic in reality, so we assume that the first and last intersection are adjacent),
5. 4S, 8S, 9N, 5N.

Intersections to erase can be calculated in linear time during computation of the beams that are crossed by the fish. We can keep the encountered intersections on a stack. While facing a new intersection, we check whether it concerns the same beam as the one on top of the stack: if so, we delete both of them. At the end we erase the first and the last intersection as long as they concern the same beam. During removal, we can ignore the directions of intersections, because a fish cannot cross one beam twice in a row from the same side. We will call the sequence thus created a *representation* of the route. The directions are meaningful only while comparing the representations of two different routes, as they allow us to distinguish the routes circumnavigating the island clockwise and counterclockwise.

### Cyclic equivalence

It turns out that two routes are equivalent if and only if their representations are cyclically equivalent*. We leave that fact without a proof, but the reader may be convinced by analyzing a few examples. Representations may be shifted cyclically, because the cyclic shift of the route depends on the place where the fish wakes up, which, as we know, has no influence on the equivalence of the routes.

The task is to find the abstraction classes of the equivalence relation, which correspond to the abstraction classes of the cyclic equivalence relation of their representations. The simplest algorithm verifying whether two words are

* Two words are *cyclically equivalent* if one is a cyclic shift of the other. A *cyclic shift* of a word $a_1a_2...a_n$ is any word of form $a_ia_{i+1}...a_na_1a_2...a_{i-1}$.

cyclically equivalent works in time $O(d^2)$, where $d$ is the length of both words. We can compute abstraction classes, checking whether each pair of routes' representations are cyclically equivalent. It leads to a solution that works in time $O(wh + n^2d^2)$.

**Improving the complexity**

Faster solutions may be obtained using more efficient algorithms for checking cyclic equivalence. It is not difficult to prove that two words $v$ and $w$ of the same length are cyclically equivalent if and only if $v$ is a subword of $ww$. Therefore, using any linear pattern-matching algorithm (e.g. the Knuth-Morris-Pratt algorithm) we can check whether two words are cyclically equivalent in linear time. It leads to a solution with time complexity $O(wh + n^2d)$.

However, we can try a different approach. Instead of checking cyclic equivalence for each pair of representations, we can transform each representation into some particular form, which we will call a *canonical representation*. It will be lexicographically maximum cyclic equivalent (LMCE), i.e. a cyclic equivalent of the representation that is the greatest in lexicographical order. For the aforementioned example, a canonical representation for 4S, 8S, 9N, 5N is 9N, 5N, 4S, 8S. It is worth mentioning that computing an LMCE can be reduced to finding a lexicographically maximum suffix in a word. It is easy to prove that a lexicographically maximum suffix in a word $vv\#$, where # is a new symbol less than all occurring in $v$, starts on a position indicating the beginning of the LMCE of $v$. The lexicographically maximum suffix in a word can be found using Duval's algorithm, which works in linear time.

Then two routes are equivalent if and only if their canonical representations are equal. We can compute a canonical representation for each route and then sort all the representations. A simple scan through the created array allows us to compute the abstraction classes. That kind of solution works in time $O(wh + nd \log n)$ if a linear-logarithmic sorting algorithm is applied. It can be improved to $O(wh + nd)$—that is, linear with the input size—by using bucket sorting or sorting hashes instead of the whole representations.

**Outline of a proof of the routes' homotopy theorem**

This section is dedicated to the following theorem's outline of a proof:

***Theorem.*** Two routes are equivalent if and only if they are homotopic.

We will begin with some definitions from topology. Let $W \subseteq \mathbb{R}^2$ be an area covered by water.



***Definition 1.*** A *loop* (directed) in $W$ is any continuous function $p : [0,1] \to W$ such that $p(0) = p(1)$. Intuitively, $p(t)$ is a place where there was a fish at time $t$.

***Definition 2.*** Let $p$ and $q$ be loops in $W$. *Homotopy* between $p$ and $q$ is any continuous transformation $F : [0,1]^2 \to W$, such that the following holds for any $0 \le r, t \le 1$:

$$F(0,t) = p(t), \quad F(1,t) = q(t), \quad F(r,0) = F(r,1).$$

Two loops are called *homotopic* if there exists a homotopy between them.

***Lemma.*** The routes traveled by one fish in two consecutive days are homotopic.

*Proof.* Let $p$ and $q$ be such routes and $H : [0,1]^2 \to W$ is defined by a formula $H(r,t) = (1-r)p(t) + rq(t)$. The image of $H$ is contained in $W$, because the fish always sees the place where it was exactly 24 hours earlier, which means that the segment between $p(t)$ and $q(t)$ is contained in $W$ for any $t$. It is easy to verify that $H$ is then a homotopy between $p$ and $q$. □

A homotopy relation is an equivalence relation, so if two routes swum by one fish in two consecutive days are equivalent, then any two routes traveled by one fish are equivalent. This ends the proof of implication "rightwards".

Let's observe that the condition put on fishes' routes in two consecutive days means precisely that there exists a homotopy between them, which is linear with $r$. Therefore, two routes are equivalent if and only if there exists a piecewise linear homotopy between them (moreover, we demand that $H(r)$ is a correct route if $r$ is a point joining the linear segments). It turns out that the existence of an arbitrary homotopy implies the existence of a homotopy in the desired form, but we won't prove that fact here.

**PIOTR CHRZĄSTOWSKI**

Piotr Chrząstowski is an assistant professor at the Institute of Informatics, University of Warsaw. For years he has taught *Introduction to Programming* to first-year students of computer science. He likes to spot algorithmic problems where others might not even expect them, and then he uses them as the basis for tests and exams. The most challenging ones are presented to participants of the Polish Olympiad in Informatics. Piotr Chrząstowski has been a member of the Main Committee of the Polish Olympiad in Informatics, with a short break, ever since the event was launched.
His scientific research focuses on the theory of distributed computation, including, first and foremost, Petri nets, which he uses as a model to describe business processes. He is an author of many programming projects and programs, with extensive practical experience in combining computer science and business. Lately he has also been involved in the design of user interfaces.
Piotr Chrząstowski is a chairman of the jury of the EU Contest for Young Scientists. He enjoys playing chess, bridge and Go. One of his biggest passions is classical music.

# / Pilots

**Contest:** 17th Polish Olympiad in Informatics
**Task author:** Piotr Chrząstowski
**Memory:** 64 MB
http://main.edu.pl/en/archive/oi/17/pil

At the Bytean Training Center, the pilots prepare for missions that require extraordinary precision and control. One measure of a pilot's capability is the length of time he is able to fly along a desired route without deviating too much—simply put, whether he can fly steadily. This is not an easy task, as the simulator is so sensitive that it registers even the slightest movement of the yoke (a device used for piloting an aircraft). At each moment the simulator stores a single parameter describing the yoke's position. Before each training session a certain tolerance level $t$ is set. The pilot's task is to fly for as long as possible in such a way that all of the yoke's positional measurements during the flight differ by at most $t$. In other words, a fragment of the flight, starting at time $i$ and ending at time $j$, is within tolerance level $t$ if the positional measurements, starting with the $i$-th and ending with the $j$-th, form a sequence $a_i, \ldots, a_j$ such that, for all elements $a_k, a_l$ of this sequence, the inequality $|a_k - a_l| \leq t$ holds.

Your task is to write a program that, given a number $t$ and a sequence of yoke position measurements, determines the length of the longest portion of the flight that is within the tolerance level $t$.

### Input

In the first line of the input two integers are given, $t$ and $n$ ($0 \leq t \leq 2{,}000{,}000{,}000$, $1 \leq n \leq 3{,}000{,}000$), separated by a single space, specifying the tolerance level and the number of yoke position measurements taken. The second line gives those measurements, separated by single spaces. Each measurement is an integer within the interval from 1 to 2,000,000,000.

### Output

Your program should print a single integer to the output: the maximum length of a portion of the flight that is within the given tolerance level.

**Example**

For the input data:

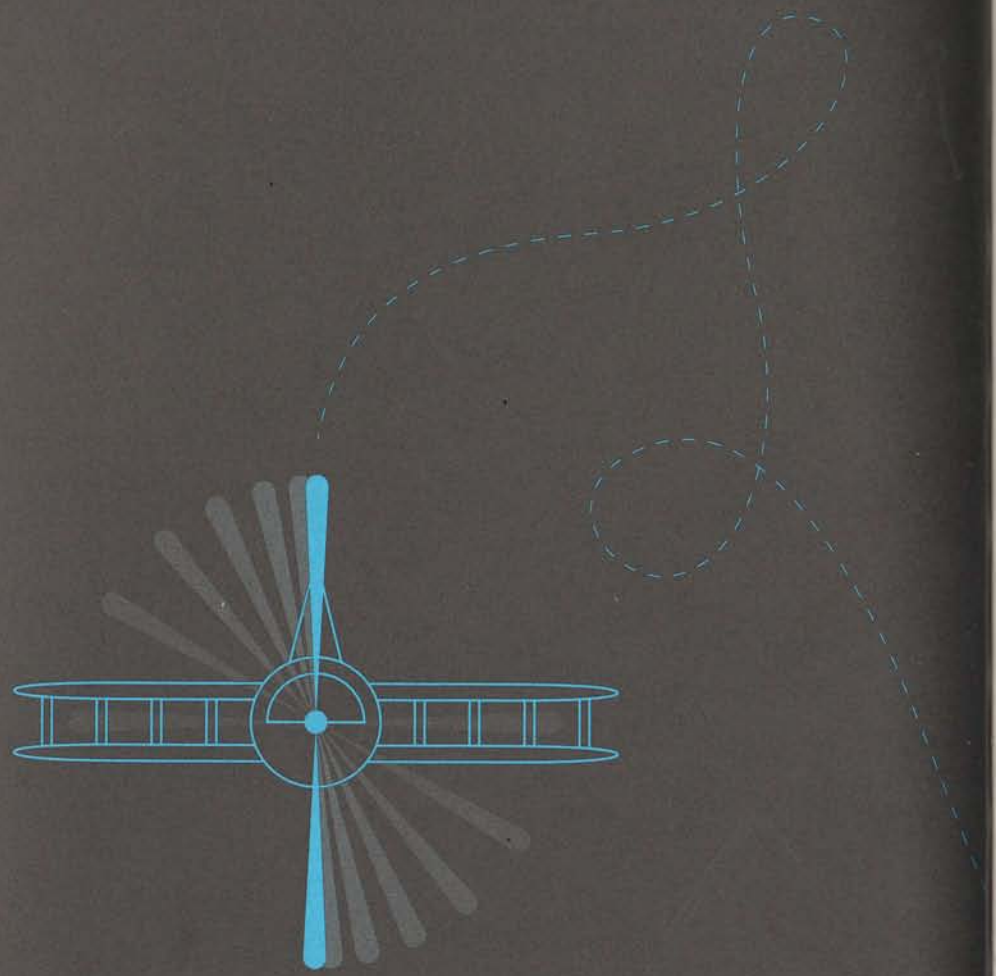
3 9

5 1 3 5 8 6 6 9 10

the correct result is:

4

**Explanation of the example:** There are two longest fragments, both of length 4: a sequence 5, 8, 6, 6 and a sequence 8, 6, 6, 9.





# / Solution

We begin by defining a few notions. Let the sequence of consecutive measurements be denoted by $a_1, a_2, \ldots, a_n$. We say that the segment $[i,j]$ is *t-stable* for a given $t$ if it satisfies the requirement of keeping the measurements within the range $t$, so for each pair of indices $i'$, $j'$, such that $i \le i' \le j' \le j$, we have $|a_{i'} - a_{j'}| \le t$. Our task is to determine the longest $t$-stable segment for a given constant $t$.

**Naïve $O(n^3)$ solution**

The first idea is to consider all pairs of indices $i$, $j$ and determine the one that corresponds to the longest $t$-stable segment.

```
record := 1 {the length of the longest t-stable segment so far}
for i := 1 to n do
    for j := i + 1 to n do
        if t-stable(i, j) then
            if j − i + 1 > record then
                record := j − i + 1
return record
```

How can we verify if a given segment $[i,j]$ is stable? Of course, we could consider all pairs $i'$, $j'$ such that $i \le i' \le j' \le j$, and verify if all values $a_{i'} - a_{j'}$ fit within the tolerance $t$. Such a solution would be too expensive: since there are quadratically many such pairs $(i',j')$ for each of the $O(n^2)$ segments $[i,j]$, its complexity would be $O(n^4)$. But a simple observation enables us to test the $t$-stability of each of the $[i,j]$ segments in linear time. To assure $t$-stability, one has to know only two values: the minimum and the maximum within the segment. If the difference between them does not exceed $t$, then such a segment is $t$-stable; otherwise it is not. We can do it easily using $2n - 2$ comparisons, and with some optimization even $\lceil 3n/2 \rceil - 2$ comparisons are enough. So, having found this idea, we have obtained a solution with a cost of $O(n^3)$, but we are still far away from the optimal solution.

**Improved $O(n^2)$ solution**

The first important remark is that extending a non-$t$-stable segment from any end cannot make it $t$-stable. In other words, if the segment $[i,j]$ is not $t$-stable, then no segment $[i',j']$ is $t$-stable for any $i' \le i$ or $j \le j'$. This means that if, for the fixed beginning of the segment $i$, we are moving the end of the segment $j$ forward, and recognize that for some $j$ the segment $[i,j]$ is not $t$-stable, then we can stop regarding $i$ as an interesting beginning: no further segments starting with $i$ will be $t$-stable, and hence the record will not be broken. Note that when we go forward with $j$, we can update the maximal and minimal values if only $a_j$ happens to be the biggest or the smallest of the considered values. This is the way to obtain a still-simple $O(n^2)$ algorithm.

```
i := 1
record := 1
while i ≤ n − record do
    {there is no need to consider greater i, because such i}
    {cannot be the beginning of the longest t-stable segment}
    min := a[i] {the smallest... }
    max := a[i] {and the biggest value in the range}
    j := i + 1
    stable := true
    while (j ≤ n) and stable do
        if a[j] > max then
            max := a[j]
        else if a[j] < min then
            min := a[j]
        if max − min > t then
            stable := false
        else
            j := j + 1
    if j − i > record then
        record := j − i
    {we need not check this condition at each loop iteration}
    {because we know the best j for given i}
return record
```



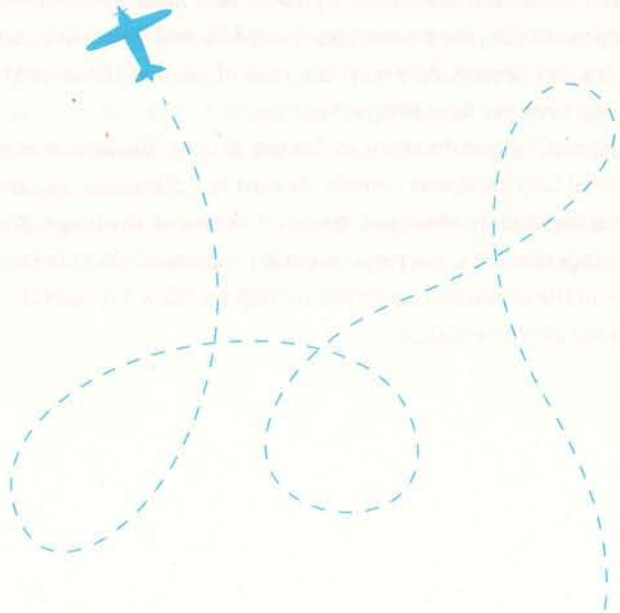
Note that we exit the loop with $j$ as the smallest index for which the segment starting at $i$ is not stable. The difference $j - i$ gives us the number of elements of the longest stable segment starting at $i$.

We have here one linear loop nested inside another, so the total cost of this solution is quadratic. Cutting the cost even further requires some more advanced techniques.

**Still better $O(n \log n)$ solution**

The main idea now is to increment two pointers representing the ends of a stable segment. If $a_j$ destroys the stability then we move $i$ forward; otherwise we increment $j$. In order to test the stability quickly, we should know the minimal and maximal values of the considered segment. To do it rapidly we will need a special technique to represent a multiset containing elements $a_i, \ldots, a_j$ in such a way that determining the minimal and maximal values is cheap. We need a multiset representation because the values can be repeated within one segment.

If we use balanced trees, like AVL, then with each value we will keep its multiplicity. The height of an AVL tree is logarithmic in the number of represented values. So, since the minimal and maximal values lie on the leftmost and the rightmost paths respectively, they can be determined in $O(\log n)$ time.



The pseudocode of this algorithm is quite simple.

```
i := 1
InitAVL(a[1]) {creates 1-element AVL tree containing a[1]}
record := 1
min := a[1]
max := a[1]
j := 2
while j ≤ n do
    InsertAVL(a[j])
    if a[j] > max then
        max := a[j]
    else if a[j] < min then
        min := a[j]
    while max − min > t do
        {if a[j] did not change min or max or if it did change, but}
        {without losing t-stability, then this loop will not be executed}
        DeleteAVL(a[i])
        Update(min, max)
        i := i + 1
    j := j + 1
    if j − i > record then
        record := j − i
return record
```

We can update the values *min* and *max* after deletion of $a[i]$ in a cleverer way by modifying the procedures *InsertAVL* and *DeleteAVL*, but let's not pursue this idea too deeply. Anyway, the cost of each of these operations is at most $O(\log n)$, even without this optimization.

The whole algorithm runs in $O(n \log n)$ time. The cost is so small because the number of loop iterations is linear: at least one of the indices $i$ and $j$ is incremented in each iteration (it does not matter that one of the loops is nested inside the other: together they can run at most $2n - 2$ times). In each loop iteration, every access to the multiset data structure requires $O(\log n)$ operations, thanks to the AVL trees implementation.



**Linear solutions**

The key observation is that we do not need to remember all the measurements from the segment $[i,j]$, but only the minimum and maximum ones. If we are able to do it quickly, the rest of the values will not be needed. We are going to present two solutions, each with a linear time cost.

**Linear implementation specific to this problem.** Let $p[j]$ for each $j$ denote the smallest index $i$ for which the $[i,j]$ segment is $t$-stable. We are looking for the longest $t$-stable segment, so if we know $p[j]$ for each $j$, then comparing the values $j - p[j] + 1$ will give us the result. Let $minval(i,j)$ and $maxval(i,j)$ be the smallest and the biggest values in the segment $[i,j]$.

We will find the solution by using inductive reasoning: a powerful method that is present in many efficient algorithms. In our case, we will consider consecutive elements $a_j$ for $j = 2,\ldots,n$ and design a method to advance from $j-1$ to $j$. Let's assume inductively that we know the value $p[j-1]$ and want to use this knowledge in determining $p[j]$. The first question is why this value should be different from $p[j-1]$. This can happen only when $a_j$ destroys stability, so either $a[j] - minval(p[j-1], j-1) > t$, or $maxval(p[j-1], j-1) - a[j] > t$. When none of these inequalities holds, then simply $p[j] = p[j-1]$.

The case in which the stability is broken, so $p[j] \neq p[j-1]$, requires finding the smallest index $i$ that makes the $[i,j]$ segment stable. Consider three cases:

- If $a[j-1] > a[j]$ then the danger comes only from the top. The new element $a[j]$ can be too small for some of the elements from the segment $[p[j-1], j-1]$. So we should find among these indices some $k$ such that none of the elements $a[k],\ldots,a[j-1]$ exceeds $a[j]$ by more than $t$. It is hence convenient to know the indices of the decreasing left-maximums in this segment, beginning with the greatest value in the whole segment. (A left-maximum in a segment is an element such that none of the elements with a bigger index is greater.)
- If $a[j-1] < a[j]$ then the danger comes only from the bottom. Analogously, we should have easy access to the increasing sequence of left-minimums. (A left-minimum in a segment is an element such that none of the elements with a bigger index is smaller.)
- If $a[j] = a[j-1]$ then we have nothing to do, except for verifying that the current segment is the longest (we have $p[j] = p[j-1]$, and the segment is now longer).

We see here that the key to success is the information about two specific sequences: the decreasing sequence of left-maximums and the increasing sequence of left-minimums for the segment $[p[j-1], j-1]$. In each of these cases we will memorize the indices of the stored values. We should also decide which of the possibly equal values will be stored. It is quite clear: if some left-maximum value $m$ is too big for a given $a[j]$, then all the occurrences of $m$ in the segment $[p[j-1], j-1]$ are equally bad, so we will be interested in the last occurrence of each such value. The first candidate for $p[j]$ will be the element whose index is greater by 1 than the last occurrence of this left-maximum. It can happen, however, that consecutive left-maximums (even all of them) are too big. In such a case, we should stop at any such leftmost left-maximum that does not violate $t$-stability (in the extreme case, the considered element $a[j]$ can become such a left-maximum, so we will always find one). An analogous situation concerns left-minimums: we need the indices of the latest occurrences of increasing values being left-minimums. For instance, for $t = 4$ and a 4-stable sequence

$$5, 4, 6, 8, 7, 7, 4, 6, 4, 5$$

the interesting decreasing values of left-maximums are 8, 7, 6, 5 with the corresponding indices being 4, 6, 8, 10, while the increasing values of left-minimums are 4, 5 with indices 9, 10. Clearly here we have $p[10] = 1$. If we extend the considered sequence with an 11th element, we must modify these two sequences. For instance, augmenting our sequence by $a[11] = 2$ we have $p[11] = 7$—the longest 4-stable segment begins with the value 4 at the 7th position. The updated sequence of left-maximums for this new 4-stable segment ending on the added 2 will be equal to 6, 5, 2, with indices 8, 10, 11, while the sequence of left-minimums will have just one element: 2 with the index 11.



Note that the actual value of $p[j]$ is not important except for the case where we increment $j$ without changing $p[j]$. Only then can the record be beaten.

Let us make it more formal. For each $j$ we define the decreasing sequence of left-maximums $UP_j$ in the following way. The first element of the sequence is the biggest value in the segment $[p[j], j]$. If there is more than one such value, we choose the last one. Consecutive elements of this sequence are the last occurrences of such elements $x$ that are smaller than the previous chosen elements, but which satisfy the property that up to the end of the segment (so up to $j$) there are no bigger values than $x$. The increasing sequence of left-minimums $DOWN_j$ is defined analogously, starting from the latest occurrence of the smallest element in the segment $[p[j], j]$.

Having defined these two sequences, we can easily determine the value $p[j]$. When we look at the first sequence $UP_j$, we see that all the values that are bigger than $a[j]$ by more than $t$ should be excluded, together with all the preceding elements. So a candidate for $p[j]$ will be the index bigger by 1 than the last element of $UP_j$, which destroys the $t$-stability. Of course, if even the first element of the sequence is not too big, then we stop this part of the investigation. In this case, $p[j]$ is possibly equal to $p[j-1]$. If, instead, the first element of the sequence exceeds $a[j]$ by more than $t$, then we delete from $UP_j$ all elements that are too big and stop the deletion on such an element whose value is bigger than $a[j]$ by at most $t$. We memorize the index of the last deleted element. The next index is a candidate for $p[j]$. Note that it may happen that all the elements of $UP_j$ will be deleted.

We perform the analogous procedure for the other sequence $DOWN_j$, determining another candidate for $p[j]$. Neither of them will object to a new segment's end $a[j]$, and both of them have indices greater by 1 than the last deleted elements. The value $p[j]$ (new value for $i$) is equal to the bigger index of these two candidates.

Now the time has come to update the values of *UP* and *DOWN* from the other end. The new value $a[j]$ must become the last value of these two sequences. In the first case we should delete all the values that are smaller than $a[j]$ and, if this value was not present in $UP_{j-1}$, augment the sequence by $a[j]$. If $a[j]$ was present in $UP_{j-1}$, then we need not augment the sequence with anything, but we must update the index of this value to $j$—this is its last occurrence now. Of course, we do the same with the other sequence. We delete all the greater values and add it at the end, or update the index if this value has been found there. Performing these actions will preserve the invariant of the loop: the sequences *UP* and *DOWN* are satisfying the definition, and the last occurrences of the values are memorized.

Now it is time to check whether the new element $a[j]$ has increased the longest $t$-stable segment so far. It is enough to check whether $j - i + 1$ is greater than the record.

Our algorithm now has three phases:

1. The updating of *UP* and *DOWN* from the beginning.
2. The updating of *UP* and *DOWN* from the end.
3. The updating of the record.

Both of our sequences can be maintained with the help of bi-queues, i.e. structures that allow the insertion and deletion of elements from both sides. The following operations will be needed (assume that the bi-queue elements will have two fields, *val* and *ind*, representing the value and the index of a segment element):

- Init($q$)—to initialize an empty bi-queue $q$,
- Empty($q$)—to verify the emptiness of $q$,
- First($q$)—to return the value of the first element of $q$,
- DeleteFirst($q$)—to delete the first element of $q$ and return its value,
- Last($q$)—to return the value of the last element of $q$,
- DeleteLast($q$)—to delete the last element of $q$ and return its value,
- InsertLast($q, x$)—to insert $x$ at the end of $q$.

Such a data structure can easily be implemented, for instance, with the help of a doubly-linked list or in an array representing a circular buffer. Each of these operations can be performed in constant time.



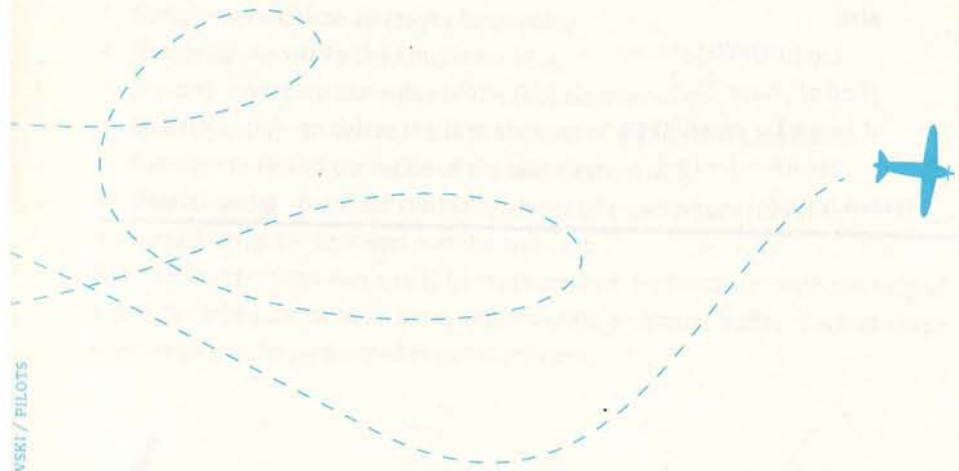
```
Init(UP)
Init(DOWN)
last_up := 0; last_down := 0
record := 1
for j := 1 to n do
    x.ind := j
    x.val := a[j]
    while (not Empty(UP)) and (First(UP).val − t > a[j]) do
        last_up := DeleteFirst(UP).ind
    while (not Empty(DOWN)) and (First(DOWN).val + t < a[j]) do
        last_down := DeleteFirst(DOWN).ind
    i := max(last_up + 1, last_down + 1)
    {End of phase 1}
    while (not Empty(UP)) and (Last(UP).val < a[j]) do
        DeleteLast(UP)
    if Empty(UP) or (Last(UP).val > a[j]) then
        InsertLast(UP, x)
    else
        Last(UP).ind := j
    while (not Empty(DOWN)) and (Last(DOWN).val > a[j]) do
        DeleteLast(DOWN)
    if Empty(DOWN) or (Last(DOWN).val < a[j]) then
        InsertLast(DOWN, x)
    else
        Last(DOWN).ind := j
    {End of phase 2}
    if j − i + 1 > record then
        record := j − i + 1
return record
```

Let's analyze the complexity of this solution. The first impression is that the complexity is not linear. Inside the outermost loop "for each $j$" we have two loops, each of them processing potentially linearly long sequences *UP* and *DOWN*. But the overall processing cannot take more than linear time, because the inner loops delete the elements, which had to be inserted previously. And at each loop iteration we can insert only one element for each $j$. So the total number of deletions is linear too.

**Range Minimum Query.** Another idea that results in a linear solution is the general RMQ algorithm. This allows us to find the minimal and maximal values in a given segment of an array quickly. After preprocessing, which is linear with respect to the sequence length, the answers are given in constant time for each desired segment. However, this algorithm is a bit tricky to implement.

### To sum up

This problem exposes a very typical case: we can improve the first idea we have by several orders of magnitude, but doing so requires quite a lot of effort and some knowledge. On this occasion we have achieved a tremendous decrease in complexity: from $O(n^3)$ (or even $O(n^4)$) to linear. Just as in real life, "cheap" does not usually mean "easy"; but it does pay off.





# / Skiers

**Contest:** 9th Polish Olympiad in Informatics
**Task author:** Piotr Chrząstowski
**Memory:** 32 MB
http://main.edu.pl/en/archive/oi/9/nar

On the south slope of Bytemount there are several ski tracks and one ski lift. All the tracks run from the top station of the ski lift to the bottom station. Every morning a group of ski lift workers examines the condition of the tracks. Together, they take the lift up to the top station, and then each of them skis down along a chosen track to the bottom station. Each worker skis down only once. The tracks of the workers may be partially the same. Each track examined by any of the workers always leads downwards.

The map of ski tracks consists of a network of clearings connected by cuttings in the forest. Each clearing lies at a different height. Any two clearings may be connected directly by at most one cutting. Skiing down from the top to the bottom station, one can choose a track to visit any one clearing (although probably not all of them in a single run). Ski tracks may cross only at clearings, and do not run through tunnels or over bridges.

### Task

Write a program that:

- reads from the input the map of ski tracks,
- computes the minimum number of workers needed to examine all the cuttings,
- writes the result to the output.

### Input

In the first line of the input there is one integer $n$ equal to the number of clearings ($2 \le n \le 5000$). The clearings are numbered from 1 to $n$.

Each of the successive $n - 1$ lines contains a sequence of integers separated by single spaces. The integers in the $(i + 1)$-th line specify which clearings the cuttings from clearing number $i$ lead down to. The first integer $k$ specifies the number of such clearings. The successive $k$ integers are the numbers of the clearings, ordered from west to east according to the arrangement of the cuttings

leading to them. The top station of the ski lift lies at clearing number 1 and the bottom station at clearing number $n$.

**Output**

In the first and only line of the output there should be exactly one integer: the minimum number of workers required to examine all the cuttings in the forest.

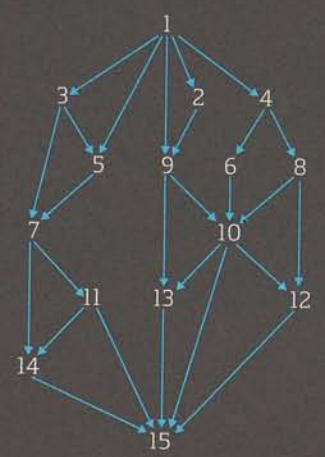
**Example**

For the input data:

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

the correct result is:

8






# / Solution

Note that we are dealing with a very special kind of graph. Nodes in the graph are the clearings, while edges are forest cuttings—each cutting connects a clearing with a different clearing. This is a planar graph, since it describes ski routes, which can intersect only at clearings: there are no tunnels or overpasses. Additionally, this graph has no cycles. Finally, the graph has two distinguished nodes: the upper and lower ski-lift stations. Each node can be reached from the top station, and from each node the bottom station can be reached. Let us call each such graph a *ski graph*.

For each ski graph we can define a dual graph as follows. The nodes of the dual graph are the edges of the original graph. There is an edge in the dual graph that connects $e_1$ with $e_2$ if the cutting $e_1$ exits at a clearing from which one can enter the cutting $e_2$. So the neighboring nodes in the dual graph are the edges of the original ski graph that lie one after another on the way from the top to the bottom of the hill. Let us denote the original ski graph by $N$ and the dual graph by $N_d$.

Consider now the relation $\leq$ in the dual graph $N_d$, defined between its nodes (cuttings) as follows: $e_1 \leq e_2$ if and only if there is a path in $N_d$ leading from $e_1$ to $e_2$. It is therefore a relation expressing the fact that one of the nodes is above the other one in a path from the top to the bottom. This relation has the following properties:

- it is reflexive, i.e. $\forall e \in E : e \leq e$;
- it is antisymmetric, i.e. $\forall e_1, e_2 \in E : e_1 \leq e_2 \wedge e_2 \leq e_1 \Rightarrow e_1 = e_2$;
- it is transitive, i.e. $\forall e_1, e_2, e_3 \in E : e_1 \leq e_2 \wedge e_2 \leq e_3 \Rightarrow e_1 \leq e_3$.

Each relation fulfilling these properties is called a *partial order* relation. Partial order relations play an important role in computing. They allow the classification of data and have been the subject of intensive research for many years.

We say that a set $A$ is partially ordered if a relation of partial order is defined in it. Such an ordered set is denoted as a pair $\langle A, \leq \rangle$. It is always important to know what order we are talking about, because the sets can be ordered in a variety of ways. Our set of forest cuttings has just been ordered by the relation $\leq$, as defined above.

If, between two elements of a partially ordered set, the relation $a \le b$ holds, we say that $a$ is not greater than $b$ and $b$ is not less than $a$. We will introduce two concepts that allow us to formulate the key theorem for the solution of our task.

In a given set $A$, partially ordered by the relation $\le$, the sequence $\{a_1, a_2, a_3, \ldots, a_n\}$ is called a *chain** if $a_1 \le a_2$, $a_2 \le a_3$, ..., $a_{n-1} \le a_n$. In our case, the chains are, for example, all possible segments of ski routes. We say that a chain $C$ is a *maximal chain* if you cannot extend it without losing the chain property. In our case, maximal chains will match routes that start at the top station and end at the bottom one. For a given chain $C = a_1, \ldots, a_n$, the $a_1$ element is called the *smallest* and $a_n$ the *biggest* element in $C$.

We define also the dual concept of an antichain. A set $B = \{b_1, \ldots, b_n\}$ is called an *antichain* if no two elements of this set are ordered by the relation $\le$. Similarly, antichain $B$ is called a *maximal antichain* if it cannot be extended without losing the property of being an antichain. In other words, $B$ is a maximal antichain if it is an antichain and, for each $x \in A \setminus B$, the set $B \cup \{x\}$ is not an antichain.

Let $A_1, \ldots, A_m$ be a family of subsets of $A$. We say that it *covers* the set $A$ if $\bigcup_{k=1}^{m} A_k = A$. Note that a given ski ride from top to bottom corresponds to selecting a maximal chain in the graph $N_d$. Our task is reduced therefore to finding such a family of maximal chains, whose elements cover the set of nodes of the dual graph of $N_d$ and which has the smallest possible number of elements.

We can make use of the following Dilworth theorem, discovered in 1950 and described in an excellent book by W. Marek and W. Lipski, *Combinatorial Analysis*.

**Theorem.** For any finite partially ordered set $\langle P, \le \rangle$ the maximum number of elements of an antichain is equal to the minimum number of chains covering the set $P$.

*Proof.* We follow the proof from the cited book. Assume that the maximum number of elements of an antichain in the set $P$ is $m$. Note first that you need at least $m$ chains to cover the set $P$. Every chain may contain at most one element of the antichain. We will show that $m$ chains are sufficient to cover the set $P$.

We will use induction on the number of elements of $P$. For $|P| = 1$ the theorem holds trivially. The only maximal antichain has one element, and this is the minimum number of chains that cover $P$.

* We consider only finite chains in this task. In set theory infinite chains are also considered, naturally extending the definition, but for our needs they are not necessary.



Suppose now that the thesis holds for all sets $P'$ with $k$ elements, where $k < n = |P|$. Consider any maximal chain $C$ in $P$ and any antichain $A$ of the largest possible cardinality $m$. Let $R(C)$ denote the set of elements of $C$. If we remove all elements of $C$ from the set $P$, and in the resulting set $P \setminus R(C)$ we restrict the relation $\le$ to these elements, then we have two cases.

In the first case the length of the longest antichain in the restricted graph, denoted by $A'$, is $m - 1$ and then, by the induction hypothesis, the set $P \setminus R(C)$ can be covered using $m - 1$ chains. Together with the chain $C$ we have found a cover that consists of $m$ chains.

In the second case the length of $A'$ is still equal to $m$ (it cannot be less than $m - 1$, because each chain can have at most one common element with each antichain, so while subtracting $R(C)$ we removed at most one element from $A$). Note that $A'$ is the maximal antichain not only in $P \setminus R(C)$ but also in $P$, and, in addition, $A' \cap C = \varnothing$. Define two sets:

$$B = \{x \in P : \exists\, a \in A' : a \le x\} \quad \text{and} \quad T = \{x \in P : \exists\, a \in A' : x \le a\}.$$

Intuitively, the set $B$ is the set of elements that do not lie above the antichain $A'$ and $T$ is the set of elements that do not lie below $A'$.

Consider the first element $c$ of $C$ (the highest). It does not belong to $B$, because otherwise the chain $C$ could be extended by some element $a \in A'$, $a < c$, contradicting the maximality of $C$. (In this case, $a < c$ implies that $a \le c$ and $a \ne c$.) Similarly, $T$ does not contain the maximal element of $C$. Thus, $|B| < |P|$ and $|T| < |P|$, and using the induction hypothesis we can see that there exist covers of these sets by chains:

$$B = B_1 \cup \cdots \cup B_m, \quad T = T_1 \cup \cdots \cup T_m,$$

because the antichain $A'$ in $P$ is also an antichain in each of these sets ($A' \subseteq B$ and $A' \subseteq T$).

Let $A' = \{a_1, \ldots, a_m\}$. Assume, without loss of generality, that $a_i \in B_i \cap T_i$ for each $1 \le i \le m$. Note that $P = B \cup T$, because otherwise having an element $x$ not comparable with any $a_i$ we could extend the antichain $A'$ by $x$, contradicting the assumption of its maximality. Therefore

$$P = (B_1 \cup T_1) \cup \cdots \cup (B_m \cup T_m)$$

is a decomposition of the set $P$ into $m$ chains. This completes the proof. □

We have therefore proven that the minimum number of ways you can go downhill to cover the whole graph is equal to the size of a longest antichain in the dual graph. Note that the Dilworth theorem does not require the chains to be maximal: we can always extend a non-maximal chain to a maximal one, allowing some overlapping of chains, and getting a correspondence with the minimum number of ski routes from the top to the bottom.

**Remark.** A clever general algorithm exists that, in a finite graph representing a partial order, finds a minimal set of chains covering the set of nodes. In this algorithm, we assume that the edges connect any two nodes that lie on the same path from the top to the bottom. We want to cover all the nodes with the minimum number of node-disjoint paths (the minimum path cover). Such cover will have as many elements as the minimum chain cover—we just have to make sure that the covering chains are disjoint. Finally, the minimum path cover of any acyclic directed graph can be found by a reduction to the problem of finding a maximum matching in a specific bipartite graph. In our case, however, such a solution would be quite inefficient. To obtain a better method, we will make use of the fact that the ski graphs are planar.

First, note that the cuttings and clearings divide the surface of the mountain into forest areas, visible to the naked eye when you look at the slope. Every two points of such an area can be connected by a polyline without crossing any of the ski routes. In figure 1, for instance, two such areas are surrounded by the cuttings connecting the nodes (sometimes uphill) 1, 2, 9, 10, 6, 4, 1 or 9, 10, 13, 9. We have 12 forest areas in the whole graph here, denoted by the letters A,B,...,L. Such areas in a planar graph are called *faces*. Now note that every maximal antichain of forest cuttings in $N_d$ consists of such cuttings, which represent a sequence of edges separating adjacent faces. Indeed, if some face were missing somewhere, then it would mean that we voluntarily gave up the maximality requirement of an antichain. Let us agree that we will therefore cross the edges from west to east. Each cutting goes at least a bit downhill, so we always know which end of a cutting is the western and which is the eastern. Such a sequence of crossed edges can easily be found in our sample graph: for instance, we can cut the graph "horizontally", crossing the edges 1-3, 1-5, 9-13, 9-10, 6-10, 8-10, 8-12. These edges form a maximal antichain, though in this graph there exist antichains that have more elements. The longest antichain will correspond to the longest horizontal cut in this graph, and its length will be greater by 1 than the number of crossed faces along this cut. How should we find this automatically?



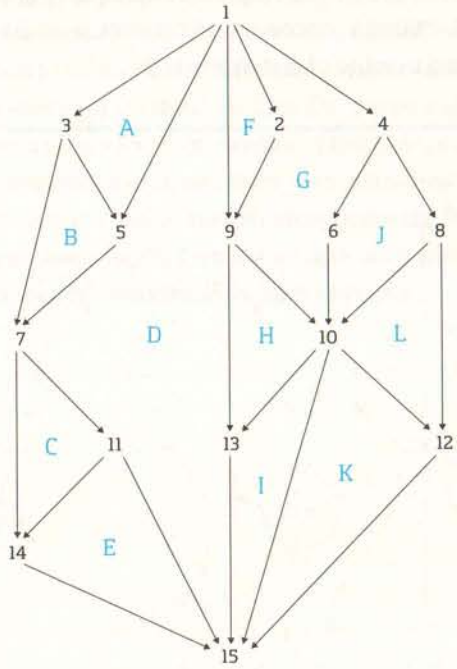



***Figure 1:*** *Example graph with associated labels of its faces.*

Let us start with the construction of the graph of faces $S$ for the original ski graph $N$. Let the nodes of $S$ be the faces of $N$, and let the edges connect these faces, which are separated by a cutting from the west to the east. Figure 2 shows such a graph of faces for the leading example. Graph nodes are labeled with symbols of the faces, and the edges of the graph are labeled by pairs of numbers representing the ends of the corresponding cuttings in the original graph $N$.

Note that in this graph of faces we have some freedom: the faces F and G could equally well be connected by the edge 2-9. In order not to complicate the description we have chosen any of the edges—certainly from two edges separating two faces, only one can be a member of any antichain, so the choice of any of them is arbitrary.

Now the problem reduces to determining the length of the longest path in this graph. The final result will be greater by 2, taking into account the external faces. This can be done, for example, using the following method. First we sort the faces topologically, storing them in an array one by one in this order. Next,

in a single pass, we can update the longest paths from some initial face. For our graph the maximum value will correspond to six equally good paths: BADFGJL, BADHGJL, BADHIKL, CEDFGJL, CEDHGJL and CEDHIKL, each having six edges. So, finally, the minimum number of test-skiers is 8.

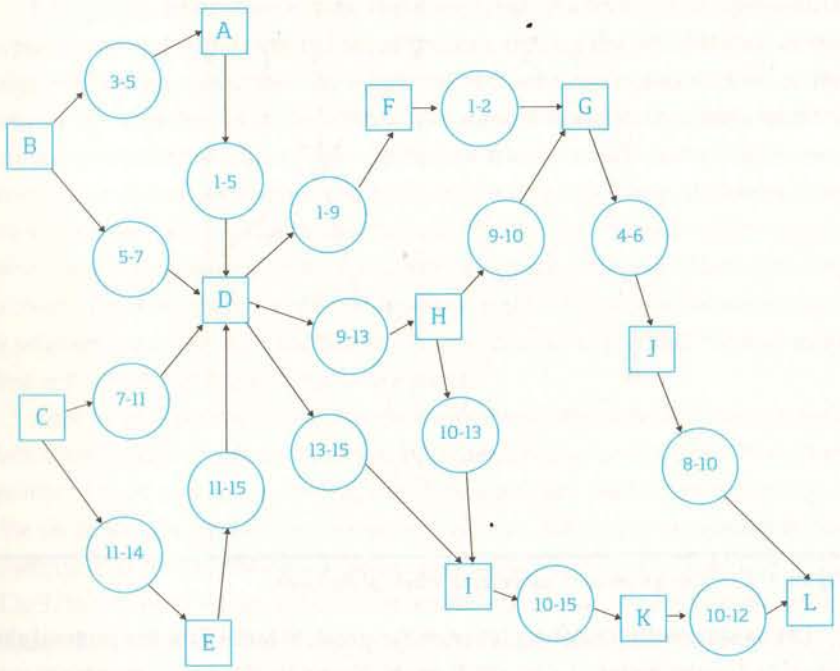



***Figure 2:*** *Graph of faces for the example ski graph.*

Consider the complexity of our algorithm. The graph of faces can be created in linear time with respect to the number of edges in $N$. Topological sorting also requires linear time. In planar graphs the number of edges and faces is linear with respect to the number of nodes (this follows from Euler's theorem), so the complexity of the overall algorithm is linear in the number of nodes of the original graph $N$.

You may, of course, complain that the solution of this task should not involve the proof of the non-trivial Dilworth theorem. It is enough to have "sufficiently good intuition" and "feel" it, and then program the described algorithm. This is true; but it is nevertheless the case that such solutions tend usually to be known, or invented, by the kinds of people who can carry out such proofs.

The idea for this task came into my mind when I stood in front of a map of an attractive ski resort and wondered how many times I would have to go up the mountain to try every part of the ski routes. Of course, a single person must go up as many times as the minimum number of test skiers ascend the ski lift in the morning. As you might imagine, there was more than one ski lift in the resort, but I thought better than to further complicate the task with multiple lifts. As it turned out, even for this simplified task with just one ski lift, only three contestants scored the maximum number of points.

**MAREK CYGAN**

Marek Cygan earned his Ph.D. at the Faculty of Mathematics, Informatics and Mechanics, University of Warsaw. His main research areas include different aspects of algorithmics, with special emphasis on approximation algorithms and parameterized complexity.

It was only after Marek Cygan had started his M.S. studies at the University of Warsaw that he scored his first successes in computer science competitions. Together with his teammates Marcin Pilipczuk and Filip Wolski (and earlier also with Piotr Stańczyk), Marek Cygan won the Polish Collegiate Programming Contest, ACM-ICPC Central European Regional Contest and ACM-ICPC World Finals in 2007. Individually, Marek Cygan was a champion of Google Code Jam 2005 and finished third in TopCoder Open 2009. As a multiple finalist of Google Code Jam, TopCoder Open and TopCoder Collegiate Challenge, he is currently ranked ninth in the TopCoder rankings.

Besides working and spending time with his family, Marek Cygan enjoys running, swimming and watching the financial markets.

# / Barricades

**Contest:** Algorithmic Engagements 2007
**Task author:** Marek Turski
**Memory:** 32 MB
http://main.edu.pl/en/archive/pa/2007/bar

Byteland is an island with a number of cities connected by two-way roads. The road network is designed in such a way that there is exactly one way of driving between any pair of cities (without turning back).

Unfortunately, hard times have come—Byteland is preparing for a war. Byteasar, the lead strategist of Byteland, is preparing a plan of defence of Byteland, which includes the creation of a *special security zone*. The zone will be created by blocking some of the existing roads in Byteland in such a way that it will not be possible to drive along these roads. In order to make the zone completely secure, the following conditions must be fulfilled:

- from each city inside the zone, it must be possible to drive to any other city inside that zone,
- it is not possible to get from a city outside the zone to a city inside the zone,
- the zone has to consist of exactly $k$ cities.

Many different solutions to the problem are being considered—for different values of $k$, it is necessary to determine the minimum number of roads to be blocked in order to obtain a special security zone of size $k$ (consisting of exactly $k$ cities). Help Byteasar—write a program that, for a specified value of $k$, calculates the required number of barricades.

## Task

Write a program that:

- reads from the input a description of the roads in Byteland and the set of queries (different values of $k$),
- for each query, determines the minimum possible number of barricades sufficient to construct a special security zone of the required size,
- writes the results to the output.

## Input

The first line of the input contains one integer $n$ ($1 \le n \le 3000$), representing the number of cities in Byteland. Cities are numbered $1, 2, \ldots, n$.

Each of the following $n-1$ lines of the input contains a pair of integers $a$, $b$ ($1 \le a, b \le n$). Each pair $a$, $b$ represents a direct road connecting cities $a$ and $b$. Each pair of cities is connected with at most one direct road.

In the following line of the input there is an integer $m$ ($1 \le m \le n$), representing the number of queries to process. Each of the following $m$ lines contains one integer $k_i$ ($1 \le k_i \le n$), representing query number $i$—the number of cities that must be inside the $i$-th special security zone.

**Output**

Your program should write to the output exactly $m$ numbers, each on a separate line. The number on the $i$-th line should be:

- $-1$, if creation of a special security zone of size $k_i$ is not possible,
- the minimum number of roads that must be blocked in order to construct a special security zone of size $k_i$ otherwise.

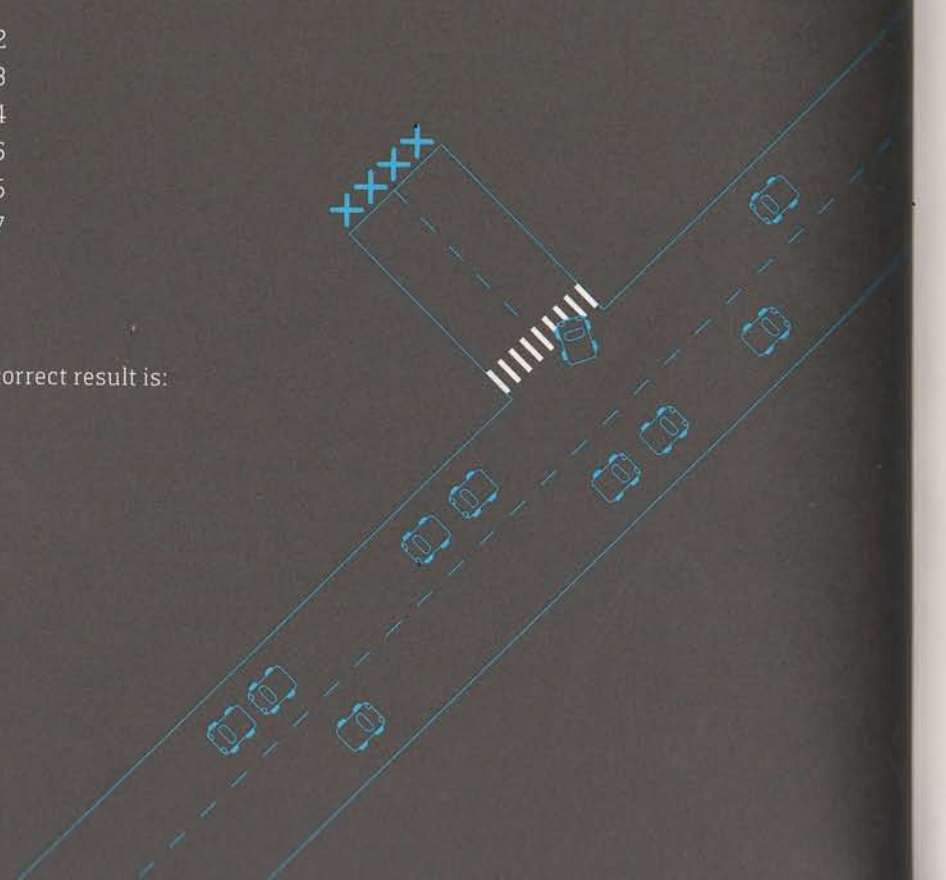
**Example**

For the input data:

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

the correct result is:

```
2
1
```





# / Solution

I encountered this problem as a finalist of Algorithmic Engagements 2007. Unfortunately, during the tournament I failed to solve the problem, which caused me to lose the competition.

Each seasoned contestant, after just a few moments, figures out a polynomial time algorithm for the problem. The time complexity of a standard dynamic programming routine can be easily upper-bounded by $O(n^3)$. However, one simple—and at the same time brilliant—observation proves the algorithm to run in $O(n^2)$ time. I find this trick very nice, and for this reason I would like to devote this chapter to *Barricades*.

**Algorithm**

Let us start by describing a standard solution using dynamic programming. For the sake of simplicity, let us root the input tree in an arbitrary vertex, and from now on assume that we are dealing with a rooted tree $T$. Consequently we use standard notation, such as parent and child relationships in the tree $T$. As usual, in a dynamic programming solution, the important part is to define exactly what our algorithm is going to compute.

For a vertex $v$ by $T_v$ denote the subtree of the tree $T$, rooted at $v$, whereas $|T_v|$ is the number of vertices in $T_v$. Moreover, for a vertex $v$ and an arbitrary integer $1 \le i \le |T_v|$, $t[v][i]$ is the minimum number of edges one has to remove from $T_v$ to make the connected component containing the vertex $v$ consist of exactly $i$ vertices. Having all the values $t[\cdot][\cdot]$ one can easily solve the problem, computing an answer for each query in $O(n)$ time. We leave the details as an exercise for the reader.

If $v$ is a leaf in the tree $T$, then the subtree rooted in $v$ has only one vertex, so $t[v][1] = 0$. Therefore, let us assume that the vertex $v$ has exactly $k$ children $u_1, \ldots, u_k$ in the tree $T$. Moreover, we assume that all the values $t[u_j][i]$, for $1 \le j \le k$ and $1 \le i \le |T_{u_j}|$, have already been computed. What remains is to merge the values that have already been obtained.

We compute an array analogous to $t$ for bigger and bigger subtrees rooted in $v$. First, we compute it just for the vertex $v$; next for $v$ and the subtree rooted in $u_1$; then for $v$ and subtrees rooted in $u_1$ and $u_2$, and so on. By $T_v^j$ let us denote the tree $T_v$ with subtrees rooted at $u_z$ removed, for each $j < z \le k$ (an example is depicted in figure 1). Let $t_j[v][i]$ be the value $t[v][i]$ obtained for the tree $T_v^j$. Observe that our goal is to compute $t[v][i] = t_k[v][i]$.

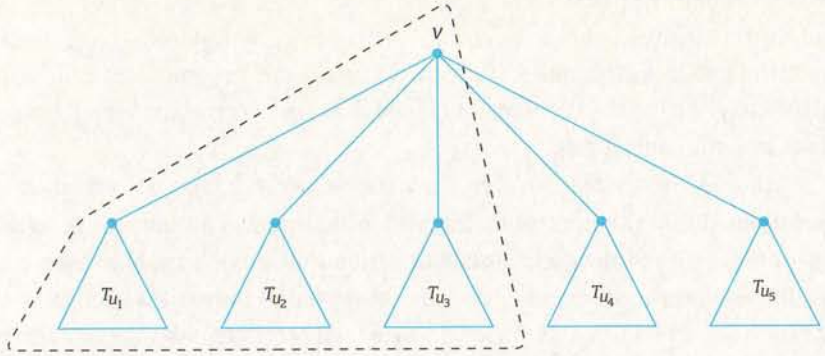



***Figure 1:*** *The area surrounded by the dashed frame contains the vertices of the tree $T_v^3$.*

We start with $j = 0$. Then $v$ is a leaf in $T_v^j$, so $t_j[v][1] = 0$. Now, we are going to compute values $t_j[v][i]$ for subsequent values of $j$ according to the following formula:

$$t_j[v][i] = \min\left(t_{j-1}[v][i] + 1, \min_{\substack{a+b=i \\ a,b\geq 1}} (t_{j-1}[v][a] + t[u_j][b])\right). \qquad (1)$$

The above formula considers two possibilities. We can erase edge $vu_j$, so that the connected component containing $v$ consists only of vertices from $T_v^{j-1}$. We can also leave the edge in the tree, in which case the connected component containing $v$ is made of $a$ vertices from $T_v^{j-1}$ and $b$ vertices from $T_{u_j}$.

**Complexity analysis**

Observe that the most time-consuming part of our algorithm relates to the formula (1). One can easily upper bound the number of operations needed to obtain all the values $t[\cdot][\cdot]$ by $O(n^3)$, since our algorithm fills $n-1$ arrays $t_j[v][\cdot]$ (one array per edge of $T$), each of which takes $O(n^2)$ running time. However, surprisingly, one can obtain a better upper bound without changing the algorithm. It suffices to investigate thoroughly the number of operations performed.

***Lemma.*** For a fixed vertex $v$, the number of operations needed to compute all the values $t[u][\cdot]$, where $u$ belongs to $T_v$, is $O(|T_v|^2)$.

*Proof.* We prove the lemma by induction over the depth of the tree $T_v$. If $v$ is a leaf, then the lemma clearly holds. Thus let us assume that the vertex $v$ has exactly $k$ children $u_1,\ldots,u_k$ and denote $a_j = |T_{u_j}|$. When performing the merge operation of the $j$-th child—that is, we compute $t_j[v][\cdot]$—we go through the values $t_{j-1}[v][a]$ and $t[u_j][b]$ for all possible $a$ and $b$. The value of $a$ can be chosen in $|T_v^{j-1}|$ ways, and $b$ in $|T_{u_j}|$ ways. This gives

$$O(|T_v^{j-1}|\cdot|T_{u_j}|) = O\big((1 + a_1 + a_2 + \cdots + a_{j-1})\cdot a_j\big)$$

operations. From the inductive assumption, computing all the values in the table $t$ for vertices below $v$ takes $O(\sum_{j=1}^{k} a_j^2)$ time. Summing up, the number of operations needed to compute the values $t[u][\cdot]$ for all the vertices $u$ belonging to $T_v$ is

$$O\Big(\sum_{i=1}^{k}\sum_{j=1}^{k} a_i a_j + \sum_{j=1}^{k} a_j + \sum_{j=1}^{k} a_j^2\Big) = O\Big((1 + \sum_{j=1}^{k} a_j)^2\Big) = O(|T_v|^2). \qquad \square$$

There is also a combinatorial interpretation of the proof of the lemma. Since attaching a single subtree involves $O(|T_v^{j-1}|\cdot|T_{u_j}|)$ operations, we can interpret this number as an amount of work proportional to the number of pairs of vertices of the tree $T$ that have $v$ as their lowest common ancestor. Since each pair of vertices is counted exactly once, we obtain $O(n^2)$ time complexity.

Observe that, using the lemma for the root of the tree $T$, we obtain an $O(n^2)$ upper bound on the total number of operations needed to compute all the values $t[\cdot][\cdot]$.

It is worth mentioning that, in order to obtain $O(n^2)$ time complexity, in the formula (1) we have to perform at most $O(|T_v^{j-1}|\cdot|T_{u_j}|)$ operations. In particular, we cannot allow ourselves to iterate over too wide a range of integer values $a$ and $b$, since even if each merge operation involves $\Omega(n\cdot|T_{u_j}|)$ complexity (or even $\Omega(|T_v|\cdot|T_{u_j}|)$), our algorithm will need $\Omega(n^3)$ worst case running time.


MAREK CYGAN / BARRICADES

# / Travel *Agency*

**Contest:** Algorithmic Engagements 2006
**Task author:** Marek Cygan
**Memory:** 32 MB
http://main.edu.pl/en/archive/pa/2006/biu

The inhabitants of Byteland have recently begun to travel more often than ever before. Enterprising Byteasar has had the idea of opening a travel agency. On the first day, $n$ customers came to the agency (we label them with integers ranging from 1 to $n$), each of them wanting to go on a trip. Unfortunately, every customer has individual requirements.

Your task is to help Byteasar in choosing the customers who should go on the trip so as to maximize Byteasar's profit.

Each customer tells Byteasar what value the trip has for him/her. Let $x_i$ be the value of the trip for the $i$-th customer ($-1{,}000{,}000 \le x_i \le 1{,}000{,}000$). If $x_i \ge 0$, then this customer pays $x_i$ bytean dollars for the trip, and if $x_i < 0$, then Byteasar has to pay the $i$-th customer $-x_i$ bytean dollars if this customer goes on the trip.

Apart from their financial requirements, customers also have social requirements. The $i$-th customer has $k_i$ ($0 \le k_i \le n-1$) such requirements; the $j$-th requirement of the $i$-th customer is represented by a pair of integers $(a_{ij}, b_{ij})$ ($1 \le a_{ij} \le n$, $a_{ij} \ne i$, $1 \le b_{ij} \le 1{,}000{,}000$). This requirement means that if the $i$-th customer goes on the trip, then the $a_{ij}$-th customer also has to go on the trip, or else the cost of the trip for the $i$-th customer has to be lowered by $b_{ij}$ bytean dollars. This might mean that, for some customers, the cost of the trip will decrease from positive to negative and Byteasar will have to pay some money to such customers, so they will go on the trip without some of their social requirements being satisfied.

Help Byteasar in choosing the customers who should go on the trip so as to maximize Byteasar's profit (the number of customers who can be taken on the trip is unlimited).

## Task

Tests in this task are open—all of them are available to contestants. The solution to the task should be a program that reads a single integer $k$ ($0 \le k \le 10$) from the input and writes the solution to the $k$-th test to the output.



In the first line of any output file there should be exactly one integer $k$ ($0 \le k \le n$): the number of customers who should be taken on the trip by Byteasar. If the number $k$ is positive, then the second line should contain $k$ integers, separated by single spaces and specifying the individual numbers of customers who should be taken on the trip. If there is more than one optimal solution, the output file should contain one of them. The numbers of the customers can be given in any order.

## Description of a single input file

In the first line there is exactly one integer $n$ ($1 \le n \le 1000$), specifying the number of customers of the travel agency. The next $n$ lines describe the customers. In the ($i$+1)-th line ($1 \le i \le n$) there are integers $x_i$ ($-1{,}000{,}000 \le x_i \le 1{,}000{,}000$) and $k_i$ ($0 \le k_i \le n-1$), and after them $k_i$ pairs of integers $a_{ij}, b_{ij}$ ($1 \le a_{ij} \le n$, $a_{ij} \ne i$, $1 \le b_{ij} \le 1{,}000{,}000$)—all numbers in the line are separated by single spaces. You may assume that every customer has at most one social requirement concerning any other customer.

## Example

For the input data:

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

the correct result is:

```
3
1  2  4
```

**Explanation of the example:** If Byteasar chooses customers 1, 2, 4 he will have a profit of 11 bytean dollars, and this is an optimal solution.

# / Solution

I set the *Travel Agency* task for the Algorithmic Engagements 2006 competition. It is an example of an unusual application of the maximum flow and minimum cut theorem.

Even though it is possible to solve the problem in polynomial time, contestants were given access to input test data in order to encourage those unable to solve the problem to gain at least some points by investigating the special structure of graphs in the first test cases.

**Solution**

In a typical application of a reduction to the minimum cut problem, each solution of the original task can be transformed into a cut of exactly the same cost, and similarly each cut can be transformed into a solution of the original problem of exactly the same cost. In this sense, *Travel Agency* is a bit peculiar, since not all cuts correspond to solutions of the original problem. Nevertheless, this property holds for all *minimum* cuts, which are what we are about to analyze.

Let us assume, for simplicity, that there are no customers with $x_i = 0$. At the end of this section we will justify why we can make this assumption. For a given instance of the *Travel Agency* task, we create the following auxiliary directed graph $G = (V, A, c)$ with capacity defined for each arc by the function $c: A \to \mathbb{R}_+$. As the vertex set we take $V = \{s, t\} \cup K$, where $s$, $t$ are two special vertices, whereas the set $K = \{v_1, \ldots, v_n\}$ contains exactly $n$ vertices, one for each customer, where a vertex $v_i$ corresponds to the $i$-th customer. Since the trip value is either positive or negative for each customer, we partition the set $K$ into $K_+$ and $K_-$ depending on the sign of the value $x_i$ corresponding to a customer. We create the set of arcs as follows:

- For each customer $v_i \in K_+$ add to the set $A$ an arc $(s, v_i)$ of capacity $x_i$.
- For each customer $v_i \in K_-$ add to the set $A$ an arc $(v_i, t)$ of capacity $-x_i$ (recall that for $v_i \in K_-$ we have $x_i < 0$).
- For each social requirement $(a_{ij}, b_{ij})$ of an $i$-th customer, add to the set $A$ an arc $(v_i, v_{a_{ij}})$ of capacity $b_{ij}$.

A sample graph $G$ is depicted in figure 1.



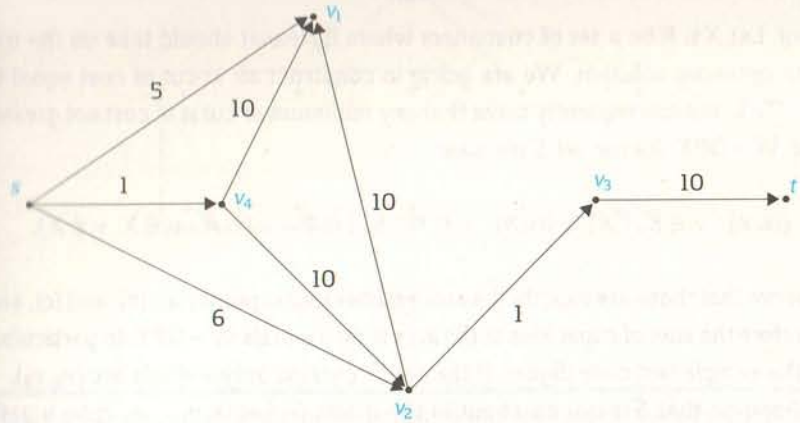



*Figure 1: Construction of the directed graph G for the sample test from the problem statement.*

A set of arcs $S \subseteq A$ is an $st$-cut if, after removing the arcs of $S$ from the graph, there is no path from the vertex $s$ to the vertex $t$. Our goal is to show that solving the *Travel Agency* problem is equivalent to finding a minimum $st$-cut in the graph $G$; that is, an $st$-cut with a minimal sum of capacities of the removed arcs.

Observe that Byteasar's problem is a maximization problem, whereas finding a minimum cut is obviously a minimization problem. Therefore we change his point of view. Let $W$ be the sum of values $x_i$ for customers from the set $K_+$. Clearly, $W$ is an upper bound on Byteasar's profit. By OPT we denote the maximum profit that Byteasar can have. Next, we consider how Byteasar's decisions affect his profit.

(a) If Byteasar does not choose a customer $v_i \in K_+$, he loses $x_i$ from the potential gain.

(b) If Byteasar chooses a customer $v_i \in K_-$, he loses $-x_i$, since he has to pay exactly $-x_i$ to the $i$-th customer.

(c) If Byteasar does not satisfy a social requirement $(a_{ij}, b_{ij})$, he has to lower the price for the $i$-th customer by $b_{ij}$ bytean dollars.

In the following lemmas the above decisions will correspond to removing arcs of each of the three types, enumerated when the set $A$ was defined.

**Lemma 1.** The cost of a minimum $st$-cut in the graph $G$ is not greater than $W - \mathrm{OPT}$.

*Proof.* Let $X \subseteq K$ be a set of customers whom Byteasar should take on the trip in an optimum solution. We are going to construct an $st$-cut of cost equal to $W - \mathrm{OPT}$, and consequently show that any minimum $st$-cut is of cost not greater than $W - \mathrm{OPT}$. As the set $S$ we take:

$$\{(s, v_i) : v_i \in K_+ \setminus X\} \cup \{(v_i, t) : v_i \in X \cap K_-\} \cup \{(v_i, v_j) \in A : v_i \in X,\ v_j \notin X\}.$$

Observe that those are exactly the arcs enumerated in points (a), (b), and (c), and therefore the sum of capacities of the arcs from $S$ equals $W - \mathrm{OPT}$. In particular, for the sample test case (figure 1) the set $S$ contains only a single arc $(v_2, v_3)$.

Assume that $S$ is not an $st$-cut in the graph $G$. Let $(s, u_1, \ldots, u_k, t)$ be a path from $s$ to $t$, which does not use any arc from the set $S$. By the definition of $G$, the arcs extending from $s$ have their endpoints in the set $K_+$, whereas all the arcs entering the vertex $t$ start in $K_-$. Therefore $u_1 \in K_+$ and $u_k \in K_-$. Since the arc $(s, u_1)$ does not belong to the set $S$, we infer that Byteasar takes the customer $u_1$ on the trip. Similarly, since $(u_k, t) \notin S$, we know that the customer $u_k$ was not taken on the trip. Consequently, there exists an integer $1 \le i < k$, such that the customer $u_i$ was on the trip and the customer $u_{i+1}$ stayed at home, and for this reason the social requirement corresponding to the arc $(u_i, u_{i+1})$ is not satisfied. Thus this arc belongs to the set $S$, which is a contradiction of the assumption that the considered path from $s$ to $t$ does not contain any arc from $S$. □

**Lemma 2.** The cost of a minimum $st$-cut in the graph $G$ is not smaller than $W - \mathrm{OPT}$.

*Proof.* Let $S \subseteq A$ be any minimum $st$-cut. We define a set of customers $X \subseteq K$ (depicted in figure 2), whom Byteasar should take on the trip, as follows. The set $X$ consists of

- each customer $v \in K_+$, for which the arc $(s, v)$ does not belong to the set $S$ and
- each customer $v \in K_-$, such that the arc $(v, t)$ belongs to the set $S$.

Let $c(S)$ be the sum of capacities of arcs from the set $S$. We want to show that the profit Byteasar achieves when taking customers from the set $X$ to the trip is at least $W - c(S)$, which consequently proves that $c(S) \ge W - \mathrm{OPT}$. In order to do that, we justify each unsatisfied social requirement by showing that its corresponding arc belongs to the set $S$.



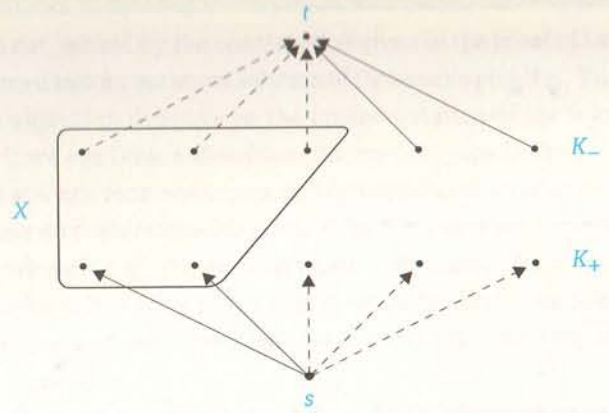



**Figure 2:** *A construction of the set of customers who should be taken on the trip based on an st-cut, the arcs of which are dashed. In the figure, arcs between vertices of the set $K_+ \cup K_-$ are not shown.*

Assume the contrary: that there exists a social requirement $(a_{ij}, b_{ij})$ of the $i$-th customer which is not satisfied, which means that the $i$-th customer belongs to the set $X$, while the customer $a_{ij}$ does not belong to the set $S$ and at the same time $(v_i, v_{a_{ij}}) \notin S$. Let $\alpha = v_i$ and $\beta = v_{a_{ij}}$ and consider the following cases depicted in figure 3. In each case, we show that in the graph $G \setminus S$ (that is, the graph $G$ with the edges from $S$ removed) there exists a path from $s$ to $t$, which contradicts the definition of a $st$-cut..

(i) $\alpha \in K_-$, $\beta \in K_-$. Since $\alpha \in X$, we have $(\alpha, t) \in S$, while from the minimality of the cut $S$ in the graph $G \setminus S$ there is a path from $s$ to $\alpha$, since otherwise removing the arc $(\alpha, t)$ would be pointless. However, $(\alpha, \beta) \notin S$, and consequently there exists a path from $s$ to $\beta$ in $G \setminus S$. By the definition of the set $X$ and by the assumption $\beta \notin X$, we infer that $(\beta, t) \notin S$, and thus there exists a path from $s$ to $t$ in the graph $G \setminus S$: a contradiction.

(ii) $\alpha \in K_-$, $\beta \in K_+$. Similarly, as in the previous case, we infer that there exists a path from $s$ to $\alpha$ in $G \setminus S$. Analogously, from the minimality of the cut $S$ it follows that there exists a path from $\beta$ to $t$, and consequently there exists a path from $s$ to $t$ in $G \setminus S$: a contradiction.

(iii) $\alpha \in K_+$, $\beta \in K_-$. In this case the arcs $(s, \alpha)$ and $(\beta, t)$ do not belong to the set $S$, and therefore in the graph $G \setminus S$ there exists a path from $s$ to $t$: a contradiction.

(iv) $\alpha \in K_+$, $\beta \in K_+$. By the definition of the set $X$, the arc $(s, \beta)$ does not belong to the set $S$. From the fact that $S$ is a minimum cut, we infer that from $\beta$ there exists a path to $t$ in the graph $G \setminus S$, and since $(s, \alpha) \notin S$, there is a path from $s$ to $t$ in the graph $G \setminus S$: a contradiction. □

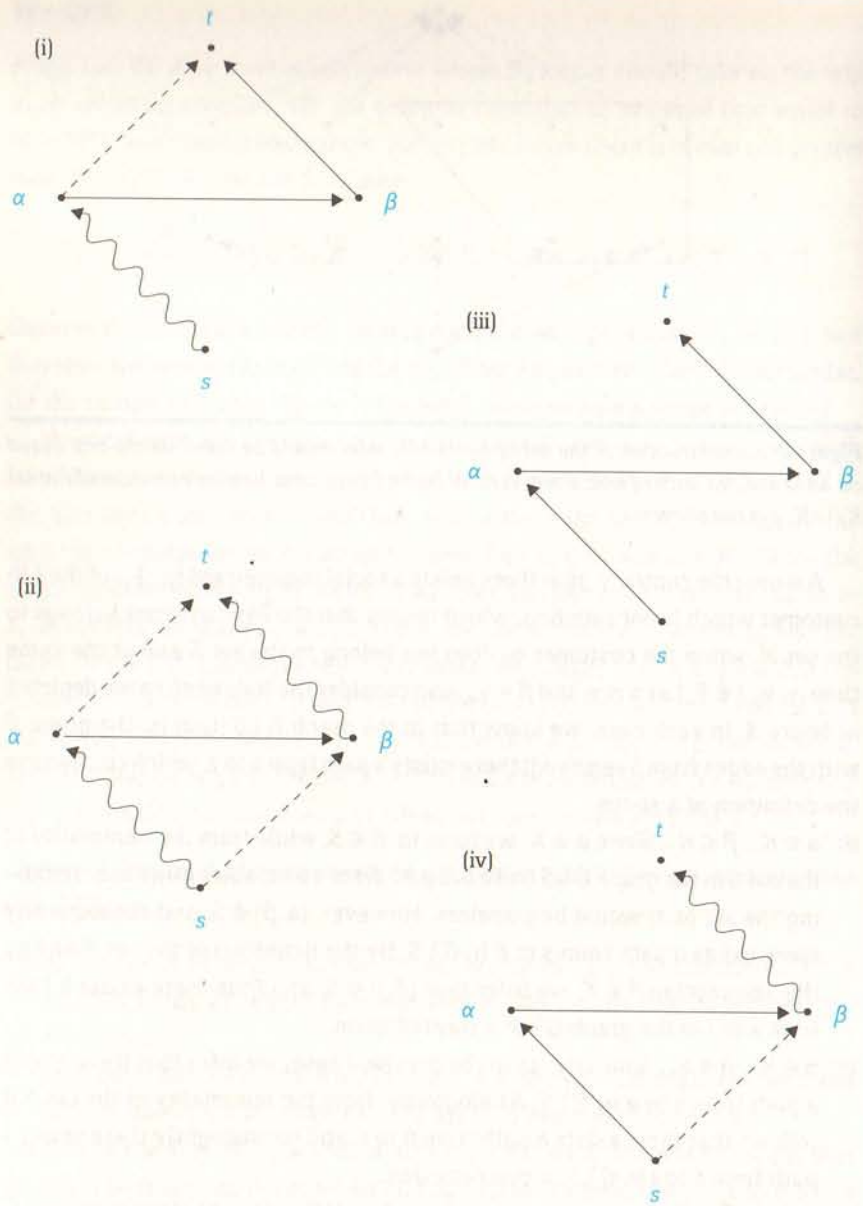



***Figure 3:*** *An illustration of the cases from the proof of Lemma 2. Dashed arcs belong to the set S, while the snake-shaped arcs denote paths that exist in the graph $G \setminus S$.*



By the maximum flow and minimum cut theorem we can effectively construct a minimum cut, which, by the construction given in the proof of Lemma 2, can be transformed into an optimum solution of Byteasar's problem. The complexity of such an algorithm depends on the implementation of the maximum flow algorithm. If we use Dinic's algorithm, the running time is $O(n^3)$.

There is still one outstanding issue. We should explain why we can assume that there are no customers with $x_i = 0$. If for all customers for which $x_i = 0$ we set $x_i := \varepsilon$, where $\varepsilon < \frac{1}{2n}$, the optimal profit of Byteasar changes by less than $\frac{1}{2}$ bytean dollars. It is easy to see that in order to obtain the solution for the original task, we can solve the modified one and take the computed group of customers on the trip.

Alternatively, we can define the set $K_+$ as the set of customers with nonnegative values $x_i$, and the only modification in the analysis would be an additional assumption in the proof of Lemma 2. Namely, if several minimum cuts exist, we consider a cut with the minimum number of removed arcs. The details, however, are left as an exercise for the reader.

**Epilogue**

I encountered an interesting application of the maximum flow and minimum cut theorem for the first time in the *Crazy Painter* problem set by Andrew Stankevich.

Furthermore, by a reduction to the minimum cut problem, one can solve the minimum weight vertex cover in bipartite graphs problem (e.g. the task *Intelligence Quotient* from the Polish Collegiate Programming Contest in 2011). An interesting task for the reader might be to give a reduction, showing that the minimum weight vertex cover in bipartite graphs is a special case of the *Travel Agency* problem. Hint: two independent sets of the graph are customers with positive and negative values of $x_i$.

**TOMASZ CZAJKA**

Tomasz Czajka is a graduate of the Faculty of Mathematics, Informatics and Mechanics, University of Warsaw and Purdue University. He has won two gold medals in the International Olympiad in Informatics and two silver medals in the International Mathematical Olympiad. He was a member of the team representing the University of Warsaw that won the ACM-ICPC World Finals for the first time in 2003. Later on, he won the TopCoder Collegiate Challenge, Top-Coder Open (three times), Google Code Jam Europe, Algorithmic Engagements and the online AI tournament, CodeCup (four times).

Tomasz Czajka currently works at the main Google campus in Mountain View, California, among other things preparing Google Code Jam competitions. In 2011 he became father to a son named Sebastian.

# / Mushrooms

**Contest:** Algorithmic Engagements 2010
**Task author:** Jakub Łącki
**Memory:** 32 MB
http://main.edu.pl/en/archive/pa/2010/grz

A heavy rain has come to the Bytean Forest. Byteasar—an eager mushroom hunter—has decided to take his chance and go mushrooming.

Byteasar knows a nice path leading through the forest, which contains several glades where lots of mushrooms may be found. Every two consecutive glades along the path are equidistant, and Byteasar needs 15 minutes to walk between any pair of adjacent glades. Having arrived at a glade, Byteasar—like any other self-respecting mushroom hunter—collects all the mushrooms in this glade in no time. It is a well-known fact that mushrooms spring up like—well, mushrooms!—and it takes them only half an hour to grow back after they are collected.

Knowing the numbers of mushrooms growing in the respective glades and the duration of Byteasar's walk, and assuming that he takes a route that maximizes his crops, find the number of mushrooms he will collect. Byteasar can finish his walk at any point on the path.

## Input

The first line of the input contains two integers, $n$ and $t$ ($1 \le n, t \le 1{,}000{,}000$), which specify the number of glades on the path and the duration of Byteasar's walk, in quarters. The second line contains a sequence of $n$ integers $a_i$ ($1 \le a_i \le 1{,}000{,}000$), representing the numbers of mushrooms in consecutive glades along the path. Byteasar starts his walk at the first of these glades. We assume that Byteasar can collect mushrooms both just before the first and just after the $t$-th quarter of mushrooming.

## Output

In the first and only line of the output, your program should write one integer: the maximum number of mushrooms that can be collected by Byteasar during his walk.

**Example**

For the input data:

5 4

3 4 3 5 1

the correct result is:

18

**Explanation of the example:** On Byteasar's optimal route, which takes him 60 minutes, he collects 18 mushrooms: 3 at time 0, 4 after 15 minutes, 3 after 30 minutes, 5 after 45 minutes and, finally, 3 after 60 minutes.



# / Solution

The problem *Mushrooms* appeared in the second online round of Algorithmic Engagements 2010, so it was one of the easier problems in that contest. Indeed, the optimal solution can be implemented in just a few lines of code. But coming up with it and proving that it is correct is not easy at all.

**Build a graph**

To start with, we should think about how to formalize the problem in the language of graph theory. We know that it takes 15 minutes to walk between adjacent glades, and that the mushrooms grow back within 30 minutes. But the problem does not require Byteasar to keep walking, or to walk at maximum speed. Might it sometimes be better to slow down or stop, in order to wait for mushrooms to grow back?

Fortunately, it is not that complicated. The key observation is that if Byteasar walks from glade $A$ to an adjacent glade $B$ then the mushrooms in $B$ have already reappeared. Why? Because even if Byteasar had collected mushrooms in $B$ before, it must have taken him at least 30 minutes to walk from $B$ to $A$ and back.

This means that, if we are moving, we can just as well walk at maximum speed. The only question remaining is whether, after we have collected mushrooms in a glade, it makes sense to stop and wait for 30 minutes until the mushrooms have grown back in the same place. The answer to that is also negative: it is not a good idea. After all, in 30 minutes we have enough time to walk to an adjacent place, collect the mushrooms there, and come back.

The only exception is when $n = 1$. Then, there is no adjacent spot to walk to, so we have no choice: we have to sit tight. This case has to be processed separately. Forgetting about simple edge cases like this is a common source of errors.

Let us summarize the findings in graph theory language. The glades are graph vertices, each vertex is assigned some *weight* (the number of mushrooms), and every two adjacent glades are connected with an edge. We are looking for a path starting in vertex 1, of length $t$ and maximum total weight.

*Figure 1: The graph correspoding to the example from the problem statement.*

**The first algorithm**

Let $f_k(x)$ be the maximum total weight of a path of length $k$, starting in vertex $x$ (vertices are numbered from 1 to $n$). In other words, it is the maximum number of mushrooms that can be collected in $k$ quarters, starting at glade $x$. Of course, $f_0(x) = a_x$, and our goal is to find $f_t(1)$. We compute the values of $f_k$ for $k > 0$ using $f_{k-1}$ and a recursive formula.

```
Algorithm Mushrooms1(n, t, a)
  if n = 1 then
    return (⌊t/2⌋ + 1) · a_1
  for x := 1 to n do
    f_0(x) := a_x
  for k := 1 to t do
    f_k(1) := a_1 + f_{k-1}(2)
    f_k(n) := a_n + f_{k-1}(n − 1)
    for x := 2 to n − 1 do
      f_k(x) := a_x + max(f_{k-1}(x − 1), f_{k-1}(x + 1))
  return f_t(1)
```

Notice that we do not have to store the whole two-dimensional table of values $f_k(x)$. When computing $f_k$ it is sufficient to store just two rows of the table: $f_k$ and $f_{k-1}$. Therefore the space required is $O(n)$. Time complexity, on the other hand, is $O(tn)$. That is too slow for the largest possible input ($n = t = 1{,}000{,}000$).

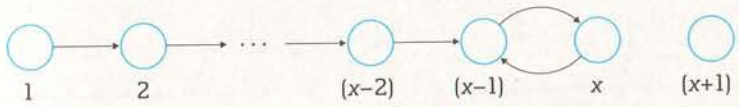
The above algorithm does not take advantage of the special form of our graph. With slight generalization, it would work for any graph. The given graph is quite simple: it is just one path. With a bit more work we can exploit that property to get a faster algorithm.



**Fast algorithm**

Let us imagine what the optimal walk looks like. A simple idea comes to mind: walk to the place with the greatest number of mushrooms in two adjacent glades, and then just walk back and forth between them. That would be the best strategy for a very large $t$, but unfortunately it is not optimal for a small $t$. For example, if $t = 7$, $n = 6$, $a = [4,4,1,1,5,5]$, the strategy would result in $4+4+1+1+5+5+5+5 = 30$ mushrooms. A better path is to walk back and forth between the first two glades, collecting $8 \cdot 4 = 32$ mushrooms. We collected a lot of mushrooms towards the end, but we lost many in trying to get to that fertile spot.

We do not give up just yet. It is not always best to go to the most fertile place, but maybe it is always a good idea to pick *some* two adjacent glades, go directly to them, and then walk back and forth between the two? That turns out to be true.




***Theorem.*** There is an optimal path of the form $1 \to 2 \to \ldots \to (x-2) \to (x-1) \to x \to (x-1) \to x \to (x-1) \to \ldots$

*Proof.* Suppose that the optimal path is not of that form. We are going to modify it so that it is, without reducing the number of mushrooms collected.

Byteasar visits, on his walk, some set of vertices $\{1,\ldots,z\}$. Find in that set two adjacent vertices $x-1$, $x$ with the largest total number of mushrooms; that is, $a_{x-1} + a_x \geq a_{y-1} + a_y$, for $y = 2,\ldots,z$.

Since the path is not of the desired form, it either ends in some vertex $y \notin \{x-1, x\}$ or Byteasar turns back at one such vertex.

If the path ends in a vertex $y \notin \{x-1, x\}$, and the last step is $(y' \to y)$, remove the last two vertices $y'$, $y$ from the path and replace them with $(x-1)$, $x$ by adding a cycle $(x-1) \to x \to (x-1)$ in an appropriate place.

Similarly, if the path turns back at some vertex $y \notin \{x-1, x\}$, i.e. Byteasar walks $(y' \to y \to y')$, remove these two steps, again adding a cycle $(x-1) \to x \to (x-1)$ instead.

In neither case do we decrease the total number of mushrooms collected, as $a_{x-1} + a_x \geq a_{y'} + a_y$. By repeating the procedure, we increase the number of times the vertices $x-1$ and $x$ are visited. After at most $t$ modifications this has to end; therefore we get an optimal path of the desired form. □

Thanks to the theorem we have a simple algorithm: just check the described path for every $x$ and pick the best one!

---

**Algorithm** Mushrooms2$(n, t, a)$
  **if** $n = 1$ **then**
    **return** $(\lfloor \frac{t}{2} \rfloor + 1) \cdot a_1$
  $s := a_1$ {Mushrooms on the direct path $1 \ldots x$}
  $r := 0$ {The best result}
  **for** $x := 2$ **to** $\min(n, t+1)$ **do**
    $s := s + a_x$
    $r := \max(r, s + \lceil \frac{t-x}{2} \rceil \cdot a_{x-1} + \lceil \frac{t-x}{2} \rceil \cdot a_x)$
  **return** $r$

---

The time complexity of this algorithm is $O(\min(n, t))$, whereas the space complexity is constant.

# / Sweets

**Contest:** Algorithmic Engagements 2010
**Task author:** Jakub Łącki
**Memory:** 128 MB
http://main.edu.pl/en/archive/pa/2010/cuk

To celebrate Children's Day, a mother of three brothers—Albert, Denis and Boris—gives them loads of sweets. The treats are packed in $n$ boxes, and there are $a_i$ sweets in the $i$-th box. The brothers want to share the boxes fairly, so they agree to the following rules:

- an older brother may not receive fewer sweets in total than a younger brother (Albert is older than Denis, and Denis is older than Boris),
- the difference between the total number of sweets given to Albert and the total number of sweets given to Boris should be minimal.

**Input**

The first line of the input contains an integer $n$ ($3 \leq n \leq 24$), specifying the number of boxes. The second line consists of $n$ positive integers $a_i$ ($1 \leq a_i \leq 1{,}000{,}000{,}000$), specifying the number of sweets in each box.

**Output**

In the only line of output you should write one integer: the difference between the numbers of sweets given to Albert and to Boris.

**Example**

For the input data:

4
5 4 7 6

the correct result is:

3

# / Solution

*Sweets* is one of the problems from the final round of Algorithmic Engagements 2010 in Zielona Góra, Poland. Its short problem statement and tiny input data may suggest that the problem is easy. Not so! The trivial algorithm, described below as Sweets1, is too slow.

The solution I implemented during the contest is Sweets2. It turned out to be fast enough. After the competition, a few people who had looked at my source code were disappointed that I had not implemented what I call Sweets3 here, and thus thought that it was just a slightly optimised version of the trivial algorithm, which was never supposed to have worked within the time limit.

I kept claiming that the solution was quite a bit faster than that. In the afternoon I described my algorithm to Eryk Kopczyński. We sat down together and analyzed its asymptotic complexity. It is not bad! The analysis follows below. Later, during the problem set discussion, Eryk described an even faster algorithm, Sweets3, that the organizers had in mind.

**The first try**

If there were only two brothers, this would be an optimization version of a classic problem called *Partition*. It is known to be NP-complete. Our problem can be treated as its generalization, and is also NP-complete. Hence, we should not hope for a polynomial solution.

On the other hand, since $n \le 24$, exponential algorithms are not out of the question. It would even seem that the easiest solution, which checks all possible ways of distributing the sweets, is good enough.

```
Algorithm Sweets1(a)
    r := ∞
    for aA, aD, aB form a partition of the multiset {a_1, ..., a_n} do
        A := ∑aA; D := ∑aD; B := ∑aB
        if A ≥ D ≥ B then
            r := min(r, A − B)
    return r
```

This results in time complexity $O(3^n n)$, since there are $3^n$ partitions of a set into three disjoint subsets. The running time can be reduced to $O(3^n)$, as the partial sums $A$, $D$, $B$ can easily be computed while generating the subsets.



$3^{24}$ is about $2.8 \cdot 10^{11}$, though. Assuming we could process a billion partitions per second, this gives 280 seconds. The time limit was 2 seconds, which means we have to think of something better.

**A better solution**

Let us return for a moment to the problem *Partition*, which consists in dividing the sweets between two brothers $A$ and $B$. We will show a known algorithm that is significantly faster than the trivial $O(2^n)$ solution.

The idea is as follows. Divide the boxes into two groups. In each group we will try all possible partitions and then merge the possibilities. But instead of trying to match every partition of the first group with every partition of the second group, which would result in the running time of $O(2^n)$, we will use a smarter approach. For each partition of the first group we can quickly find the best matching partition of the second group. In order to do that, first sort all possibilities in both groups by the difference in the number of sweets received by the two brothers.

```
Algorithm Partition(a)
    {Returns the optimal partition (A, B), A ≥ B of the multiset {a_1, ..., a_n}}
    K := multiset of all partitions (A_1, B_1) of the multiset {a_1, ..., a_⌊n/2⌋}
    L := multiset of all partitions (A_2, B_2) of the multiset {a_⌊n/2⌋+1, ..., a_n}
    {|K| = 2^⌊n/2⌋, |L| = 2^⌈n/2⌉}
    sort K decreasingly by A_1 − B_1
    sort L increasingly by A_2 − B_2
    r := ∞; i := 0; j := 0
    while i < |K| and j < |L| do
        (A_1, B_1) := K[i]
        (A_2, B_2) := L[j]
        d := A_1 + A_2 − B_1 − B_2
        if d ≥ 0 then
            if d < r then
                r := d
                (A, B) := (A_1 + A_2, B_1 + B_2)
            i := i + 1
        else
            j := j + 1
    return (A, B)
```

The running time of the Partition algorithm is dominated by sorting and is equal to $O(2^{n/2}\log(2^{n/2})) = O(\sqrt{2}^n n)$. A significant improvement!

Now we can directly use this algorithm for the *Sweets* problem. Note that at least one of the brothers has to receive not more than $\lfloor n/3 \rfloor$ boxes, and the remaining sweets should be split as evenly as possible. Let us check all such possibilities.

---

**Algorithm** Sweets2($a$)
  $r := \infty$
  **for** $aX \subseteq \{a_1, \ldots, a_n\}$, $|aX| \le n/3$ **do**
    $X := \sum aX$
    $(Y, Z) :=$ Partition($\{a_1, \ldots, a_n\} \setminus aX$)
    $r := \min(r, \max(X, Y, Z) - \min(X, Y, Z))$
  **return** $r$

---

We still have to compute the time complexity. This is not easy! Let us assume, for simplicity, that $n$ is divisible by 3. After all, we could always throw in an extra one or two empty boxes, which would not change the result.

There are $\binom{n}{k}$ subsets $aX$ of size $k$. Therefore, the time complexity is

$$T(n) = \sum_{k=0}^{n/3}\binom{n}{k} O(\sqrt{2}^{n-k}(n-k)) = O\Big(n\sum_{k=0}^{n/3}\binom{n}{k}\sqrt{2}^{n-k}\Big).$$

For $1 \le k \le n/3$,

$$\binom{n}{k-1} = \frac{n!}{(k-1)!(n-k+1)!} = \frac{k}{n-k+1}\binom{n}{k} \le \frac{n/3}{2n/3}\binom{n}{k} = \frac{1}{2}\binom{n}{k},$$

$$\binom{n}{k-1}\sqrt{2}^{n-(k-1)} \le \frac{1}{\sqrt{2}}\binom{n}{k}\sqrt{2}^{n-k}.$$

In other words, each term of the sum is not greater than $1/\sqrt{2}$ of the next one. Therefore

$$T(n) = O\Big(n\binom{n}{n/3}\sqrt{2}^{2n/3}\Big(1 + \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}^2} + \frac{1}{\sqrt{2}^3} + \cdots\Big)\Big) = O\Big(n\binom{n}{n/3}2^{n/3}\Big),$$

since the geometric series converges. We can estimate the binomial coefficient using Stirling's formula:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\big(1 + o(1)\big) = \Theta(n^{n+1/2}e^{-n}),$$



$$\binom{n}{n/3} = \frac{n!}{(n/3)!(2n/3)!} = O\Big(\frac{n^{n+1/2}e^{-n}}{(n/3)^{n/3+1/2}e^{-n/3}(2n/3)^{2n/3+1/2}e^{-2n/3}}\Big) =$$
$$= O\Big(n^{-1/2}e^0 3^{n+1}2^{-(2n/3+1/2)}\Big) = O\Big(n^{-1/2}3^n 2^{-2n/3}\Big),$$

$$T(n) = n\binom{n}{n/3}2^{n/3} = O\Big(n^{1/2}3^n 2^{-n/3}\Big) = O\Big((3/\sqrt[3]{2})^n\sqrt{n}\Big) = O(2.39^n).$$

Finally, we arrive at the time complexity of algorithm Sweets2: $O(2.39^n)$.

**Author's solution**

In the previous algorithm we used the Partition procedure directly for two of the brothers. Instead of that, we can get an even faster algorithm by applying the idea of Partition directly to all three brothers. We split the boxes into two groups $K$ and $L$, partition both groups between the three brothers in every possible way, and try to quickly merge the partitions of $K$ with appropriate partitions of $L$.

Then, we perform the following transformation. For every partition $(A_1, D_1, B_1)$ of $K$ define a point on the plane, $(x_1, y_1) = (A_1 - D_1, D_1 - B_1)$. For every partition $(A_2, D_2, B_2)$ of $L$ define the point $(x_2, y_2) = (D_2 - A_2, B_2 - D_2)$.

We are looking for a solution $A = A_1 + A_2$, $D = D_1 + D_2$, $B = B_1 + B_2$ that satisfies $A \ge D$, $D \ge B$ and minimizes the value of $A - B$. It is easy to verify that these correspond to $x_1 \ge x_2$, $y_1 \ge y_2$, and we are trying to minimize $(x_1 - x_2) + (y_1 - y_2)$.

The problem has thus been reduced to a purely geometric one! Given two sets $K'$, $L'$ of points on a plane, find two points satisfying the above conditions.

We can solve the problem using a sweep line algorithm. Sort all points by the $y$ coordinate (and then by $x$) and process them in order. Whenever we find a point of $L'$, we put it in a set $S$. When we find a point of $K'$, we find in $S$ the point $(x, y)$ with no larger $x$ coordinate and maximal sum $x + y$. This way, for each point of $K'$ we quickly find the "closest" point of $L'$, which allows us to find the optimal solution efficiently.

It remains to describe how to represent the set $S$. We store the points of $S$ in a balanced binary search tree (a red-black tree, for instance), sorted increasingly by $x$. Additionally, before we insert a point $(x, y)$ into the tree, we always check whether there is another point $(x', y')$ in there such that $x' \ge x$ and $x' + y' \le x + y$. If so, the point $(x', y')$ will no longer be needed in our algorithm, since the new point is "better". Hence, we delete all such points from the tree.

This way, our tree will always be sorted increasingly, both by $x$ and $x+y$. Thus, given a point $(x,y)$, we can quickly find in the tree the point $(x',y')$ with the largest $x' \le x$, and at the same time it will also be the point with the largest $x'+y'$.

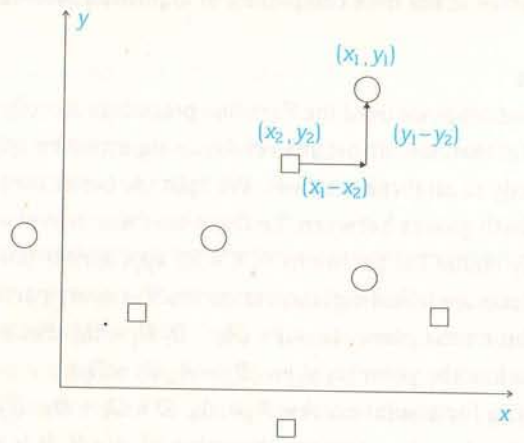



***Figure 1:*** *The circles are elements of $K'$, while the squares are elements of $L'$.*

**Algorithm** Sweets3($a$)

$K :=$ multiset of all partitions $(A_1, D_1, B_1)$ of the multiset $\{a_1, \ldots, a_{\lfloor n/2 \rfloor}\}$
$L :=$ multiset of all partitions $(A_2, D_2, B_2)$ of the multiset $\{a_{\lfloor n/2 \rfloor+1}, \ldots, a_n\}$
$\{|K| = 3^{\lfloor n/2 \rfloor}, |L| = 3^{\lceil n/2 \rceil}\}$
$K' :=$ multiset $\{(A_1 - D_1, D_1 - B_1) : (A_1, D_1, B_1) \in K\}$
$L' :=$ multiset $\{(D_2 - A_2, B_2 - D_2) : (A_2, D_2, B_2) \in L\}$
sort $K' \cup L'$ increasingly by $y$, then by $x$
$r := \infty$
$S := \varnothing$
**for** $(x, y) \in$ sorted $K' \cup L'$ **do**
  $\{S$ is sorted by both $x$ and $(x+y)\}$
  **if** $(x,y)$ comes from $K'$ **then**
    $(x', y') :=$ the point of $S$, such that $x'$ is largest among $x' \le x$
    **if** such a point exists **then**
      $r := \min(r, (x+y) - (x'+y'))$
  **else**
    **while** $S$ contains $(x', y')$, such that $x' \ge x$ and $x'+y' \le x+y$ **do**
      $S := S \setminus \{(x', y')\}$
    $S := S \cup \{(x,y)\}$
**return** $r$

The sizes of $K'$ and $L'$ are $O(3^{n/2}) = O(\sqrt{3}^n)$. All operations on $S$ work in logarithmic time and every point is added to and deleted from $S$ at most once. Therefore the time complexity of the whole algorithm is

$$O(\sqrt{3}^n \log \sqrt{3}^n) = O(\sqrt{3}^n n) = O(1.74^n).$$

**KRZYSZTOF DIKS**

Krzysztof Diks is a professor at the Institute of Informatics, University of Warsaw. He specializes in algorithmics and teaches Algorithms and data structures. Krzysztof Diks has been involved in the Polish Olympiad in Informatics since 1994, chairing its Main Committee since 1999. In 2005 he was the chairman of the Organizing Committee of the International Olympiad in Informatics that took place in Poland. He is the originator and co-organizer of the most popular algorithmic competition in Poland, Algorithmic Engagements. He is an author of many competition tasks and a keen promoter of computer science.

Since 1999, together with Professor Jan Madey, Krzysztof Diks has been mentoring teams representing the University of Warsaw in various programming competitions. In 2003 and 2007 he was a co-coach of world champions at the ACM-ICPC.

# / Coding *of Permutations*

**Contest:** 2nd Polish Olympiad in Informatics
**Task author:** Krzysztof Diks
**Memory:** 32 MB
http://main.edu.pl/en/archive/oi/2/kod

Every permutation $A = (a_1,\ldots,a_n)$ of numbers $1,\ldots,n$ can be coded by a sequence $B = (b_1,\ldots,b_n)$ in which $b_i$ equals the number of all $a_j$ such that $j < i$ and $a_j > a_i$, for $i = 1,\ldots,n$.

**Example**

The sequence $B = (0, 0, 1, 0, 2, 0, 4)$ is the code of the permutation $A = (1, 5, 2, 6, 4, 7, 3)$.

**Task**

Write a program that:

- reads from the input the length $n$ and the successive elements of the sequence $B$,
- examines whether the sequence is a code of some permutation of the numbers $1,\ldots,n$,
- if so, finds that permutation and writes it to the output, or otherwise writes to the output one word: NIE (Polish for *no*).

**Input**

In the first line of the input there is a positive integer $n$ ($n \leq 30{,}000$), representing the number of elements in the sequence $B$. Each of the following $n$ lines contains the successive elements of the sequence $B$ as nonnegative integers not greater than 30,000.

**Output**

The output should contain:

- on each of $n$ consecutive lines, one element of the permutation $A$, whose code is the sequence $B$ supplied in the input,
- one word, NIE, if the sequence $B$ is not a code of any permutation.

**Examples**

For the input data:

7
0
0
1
0
2
0
4

the correct result is:

1
5
2
6
4
7
3

whereas for the input:

4
0
2
0
0

the correct output is:

NIE



# / Solution

I would like to present two problems that I prepared for the Polish Olympiad in Informatics. At first glance, both of these problems may appear rather simple. Indeed, their straightforward solutions are easily found even by beginners. Most probably, however, such *ad hoc* approaches will run unacceptably slowly: the true challenge is in designing fast, efficient algorithms.

Both of the problems exemplify my philosophy of what constitutes good design for exercises in algorithmics. In proposing a new algorithmic task, my goal is to prepare a problem that is not only original and novel, but that also imparts important principles such as the need for proper algorithm design techniques and appropriate data structures to achieve efficiency. Additionally, a thoughtful solution to a well-designed problem should have the potential to become a useful pattern for tackling a range of tasks.

The problem *Coding of Permutations* demonstrates such characteristics. Although quite a typical problem by 2012 standards, this task embraces pedagogical values that were critical to teaching algorithmics in 1994, when access to educational resources (such as textbooks and web-based contest servers) was rather limited compared to what is available today. These values remain important even now.

The problem *Rooks* shares similar pedagogical values with *Coding of Permutations* and adds another important aspect: it encourages experimentation and playing with props, such as a chessboard or a sheet of paper.

The year 1994 was particularly important to me. In 1994, I started my adventures in popularizing algorithmics with Polish high school students; it was the beginning of my involvement with the Olympiad in Informatics. Working with algorithmically gifted youths has been very rewarding to me, and their successes have brought me tremendous enjoyment. I was very fortunate to have an opportunity to work with the future winners of the International Olympiad in Informatics and champions of the World Finals of the ACM International Collegiate Programming Contest. Watching the successful careers of past participants of programming competitions, who have become top researchers and employees of the world-leading IT companies, has brought me a lot of joy. And everything started with solving problems such as the two described below.

### An abstract point of view

A typical beginner's mistake is to look at algorithm design from the outset in implementational rather than abstract terms. Such an approach often immediately eliminates an array of potential solutions. It is much more productive to start thinking about the problem and its solution using abstract constructs (such as a set or a sequence) and abstract operations on such objects (e.g. "add an element to a set", "test if a given object is a member of a set", "return the $i$-th element in a sequence"). Expressing the algorithm in abstract terms opens a variety of options for selecting the best implementation for the problem at hand. Here, the "best implementation" is the one that results in the best running-time performance.

Let us try the approach advocated above on *Coding of Permutations*. Note that, for every $n$-element permutation $A$, the elements of its code satisfy $0 \le b_i < i$ for $i = 1, 2, \ldots, n$. This implies that there are at most $n!$ different codes, since $b_i$ can have at most $i$ different values. On the other hand, one can show that different $n$-element permutations $A'$ and $A''$ have different codes. To this end, consider the largest index $j$ such that $a'_j$ and $a''_j$ are different. Without loss of generality we can assume that $a'_j < a''_j$. Thus $b'_j > b''_j$, because all elements that are larger than $a''_j$ and are to the left of $a''_j$ in the permutation $A''$ are to the left of $a'_j$ in the permutation $A'$. Additionally, $a''_j$ is located to the left of $a'_j$. Since there are $n!$ permutations, the number of codes is also $n!$. Therefore, every sequence $B$ that satisfies the condition

$$0 \le b_i < i \quad \text{for all } i = 1, 2, \ldots, n \qquad (*)$$

is a (unique) code for some permutation.

It is not difficult to check in linear time whether a given sequence satisfies the condition $(*)$. Thus, we will henceforth assume that $B$ encodes some permutation, and our goal is to "decode" this permutation.

Let us start with the observation that $b_n$ unambiguously identifies an element in the set of *candidates* $\{n, n-1, \ldots, 1\}$ that is the last in the permutation $A$. Indeed, $b_n$ indicates that, in the permutation $A$, to the left of $a_n$ there are exactly $b_n$ elements larger than $a_n$. Since $a_n$ is at the last position in $A$, it means that $a_n$ is equal to the $(b_n + 1)$-th element in $\{n, n-1, \ldots, 1\}$, counting from the largest. After removing $a_n$ from the set of candidates, $a_{n-1}$ is equal to the $(b_{n-1} + 1)$-th largest element in the updated set of candidates $\{n, n-1, \ldots, 1\} \setminus \{a_n\}$. Continuing, $a_{n-2}$ is equal to the $(b_{n-2} + 1)$-th largest element in $\{n, n-1, \ldots, 1\} \setminus \{a_n, a_{n-1}\}$, etc.



Below is pseudocode for the entire abstract-level algorithm.

```
Algorithm DecodePermutation((b_1, ..., b_n))
    Candidates := {n, n − 1, ..., 1}
    for i := n downto 1 do
        k := b_i + 1
        e := k-th largest element of Candidates
        a_i := e
        Candidates := Candidates \ {e}
    return (a_1, ..., a_n)
```

Once we have this abstract algorithm, we can start thinking about its implementation. Both the code $B$ and permutation $A$ will be represented by $n$-element arrays, $B[1..n]$ and $A[1..n]$, respectively. Crucial to the solution, however, is the right choice of implementation for the set *Candidates*. What do we know about this set? At the beginning, *Candidates* includes all of the numbers $1, 2, \ldots, n$. In each iteration of the loop, we find an element $e$ that is the $k$-th largest in *Candidates*, where $k$ is a number between 1 and the current cardinality of *Candidates*. Next, $e$ is removed from *Candidates*. In our solution, we will combine these two operations into a single operation *FindAndDelete*($k$). It is clear that the running-time complexity of the algorithm will depend on the representation of the set *Candidates* and the implementation of the operation *FindAndDelete*($k$).

### Implementation

**Approach 1—an array.** In this approach, we represent the set *Candidates* in an $n$-element array *Cand*[1..$n$] of 0s and 1s. *Cand*[$e$] is equal to 1 if $e$ is in *Candidates*, and 0 otherwise. With this choice of representation, *FindAndDelete* can be simply implemented as a scan of the array *Cand* from $n$ down to 1 to find the $k$-th 1 value. Deleting the found element is accomplished by assigning 0 to its corresponding position in *Cand*.

```
Function: FindAndDelete(k)
    count := 0; e := n + 1
    while count < k do
        e := e − 1
        if Cand[e] = 1 then
            count := count + 1
    Cand[e] := 0
    return e
```

It is easy to see that finding the element $e$ with *FindAndDelete* requires $n - e + 1$ positions to be scanned in the array *Cand*. Since ultimately each of the elements in *Cand* is returned by *FindAndDelete*, a total of $\sum_{e=1}^{n}(n - e + 1) = n(n + 1)/2$ positions will be inspected to decode the permutation. Unfortunately, with this approach, the cost of decoding is always of order $n^2$, independent of the code $B$.

**Approach 2—a list.** The quadratic performance of the previous approach to searching for the $k$-th element is caused by scanning both the elements that are still in the set *Candidates* (1s in *Cand*) and those that were already removed from this set (0s in *Cand*). To alleviate this problem, we can represent the set of candidates as a decreasing list of its elements. In such a representation, finding the $k$-element in the set corresponds to finding the $k$-th element of the list, and requires only $k$ elements in the set *Candidates* to be scanned—the first $k$ elements in the list. An element can be deleted in constant time. We assume that the list is constructed with fields *next*[$e$] pointing to the next element in the list, with *next*[$e$] = 0 if $e$ is the last element in the list. Access to the list is given by a pointer *Cand* to $s$ (a sentinel) with *next*[$s$] pointing to the first element in *Candidates*. Thus *FindAndDelete* can be implemented as follows:

```
Function FindAndDelete(k)
    count := 1; f := Cand
    while count < k do
        f := next[f]
        count := count + 1
    e := next[f]; next[f] := next[e]
    return e
```



In this implementation, the cost of decoding depends on the code $B$. The total number of times elements of the set *Candidates* are scanned is $\sum_{i=1}^{n}(b_i + 1)$, which has a minimum of $n$ when all $b_i$ are equal to 0, and a maximum of $n(n + 1)/2$ if $B = (0, 1, 2, \ldots, n - 1)$. Thus, with the list representation, the running time of decoding varies from linear to quadratic. What is the average cost? It depends, of course, on the distribution of data. If we assume that $B$ is a code of a random permutation, and each permutation is equally likely (that is, with probability $1/n!$), then it is possible to show that the average number of elements scanned in the changing list *Cand* is $n(n + 2)/4$.

Can we solve our problem in better time?

**Approach 3—binary search and trees.** Let us return to the representation of the *Candidate* set as a 0-1 array *Cand*[1..$n$]. Assume now that we have an "oracle" function *HowManyOnes*($l, p$) that returns the number of 1s in the subarray *Cand*[$l$..$p$]. Now, the $k$-th 1 from the right end of the array can be found with a binary search. Divide *Cand*[$l$..$p$] into two equal (up to one element) parts *Cand*[$l$..$s$] and *Cand*[$s + 1$..$p$], where $s = \lfloor (l + p) / 2 \rfloor$. If *HowManyOnes*($s + 1, p$) $\geq k$, then the search continues for the $k$-th 1 from the right in the subarray *Cand*[$s + 1$..$p$]. If, on the other hand, *HowManyOnes*($s + 1, p$) $< k$, then we continue the search for 1 from the position $k -$ *HowManyOnes*($s + 1, p$) from the right in *Cand*[$l$..$s$], since all 1s in *Cand*[$s + 1$..$p$] are excluded from the search. The algorithm is presented as a recursive function *FindAndDelete*($k, l, p$) that finds and deletes the $k$-th 1 from the right in the subarray *Cand*[$l$..$p$]. The $k$-th 1 from the right in the entire array (that is, the $k$-th largest element) can be found by calling *FindAndDelete*($k$, 1, $n$).

```
Function FindAndDelete(k, l, p)
    if l = p then
        Cand[l] := 0
        return l
    else
        s := ⌊(l + p)/2⌋
        if HowManyOnes(s + 1, p) ≥ k then
            return FindAndDelete(k, s + 1, p)
        else
            return FindAndDelete(k − HowManyOnes(s + 1, p), l, s)
```

If we could execute the function *HowManyOnes* in constant time then the total cost of *FindAndDelete*$(k, 1, n)$ would be $O(\log n)$, because in each step of the recursive search the size of the subarrays is halved (plus/minus one element). But where do we get the values of *HowManyOnes*? Actually, they can be memorized. Assume that we have an array *HowMany*$[1..n]$ such that, for $l < p$ which are possible arguments for *FindAndDelete*, *HowMany*$[s]$ = *HowManyOnes*$(s+1, p)$, where $s = \lfloor (l+p)/2 \rfloor$. Then, calls to *HowManyOnes*$(s+1, p)$ in *FindAndDelete* can be replaced with calls to *HowMany*$[s]$. And how about deleting an element $e$ from the *Candidate* set? That is also simple. Whenever the search continues in the right subarray *Cand*$[s+1, p]$, we know that a 1 will be removed from this subarray. Consequently, *HowMany*$[s]$ should be decreased by 1. The above ideas are summarized in the following function:

```
Function FindAndDelete(k, l, p)
    if l = p then
        Cand[l] := 0
        return l
    else
        s := ⌊(l + p)/2⌋
        if HowMany[s] ≥ k then
            HowMany[s] := HowMany[s] − 1
            return FindAndDelete(k, s + 1, p)
        else
            return FindAndDelete(k − HowMany[s], l, s)
```

We still have to show how to initialize the array *HowMany*. Since at the beginning *Cand* contains only 1s, the task is not difficult. A natural recursive function to initialize *HowMany* is given below:

```
Function InitHowMany(l, p)
    if l < p then
        s := ⌊(l + p)/2⌋
        HowMany[s] := p − s
        InitHowMany(l, s)
        InitHowMany(s + 1, p)
```



We leave the proof that *InitHowMany*$(1, n)$ has linear cost as an exercise for the reader. The interested reader should have noticed that, in the last approach, the array *Cand* is not needed, except perhaps for clarity of presentation.

The problem presented above is one of the easiest problems in this book. It is not hard to notice the presence of inversions in this problem, which represent the difficulty of the insertion sort algorithm. An inversion in a sequence is a pair of numbers in the sequence that are not in order. The cost of the insertion sort algorithm is proportional to the length of the sequence and the number of inversions within it.

Despite being simple, the problem allowed us to demonstrate a strategy for tackling algorithmic problems to obtain efficient solutions (efficient running time). The strategy is to start with the development of an algorithm in abstract terms and then select the best implementation for the abstract functions and objects. Additionally, the problem can be used to introduce and illustrate a variety of data structures (arrays, lists, trees), search algorithms (linear, binary), recursion and the concept of the running-time complexity (worst-case and average).

**Reversing the problem**

Finally, let us discuss efficient solutions for the reversed problem: given a permutation $A$, find its code $B$. A natural approach is to scan the permutation $A$ from left to right and count, for each of its elements $a_i$, the number of elements in $A$ larger than $a_i$ and to the left of $a_i$ in $A$. A straightforward implementation of this approach has $O(n^2)$ running time. A better implementation could use balanced binary trees to obtain running time $O(n \log n)$. Below we will show yet another implementation with running time $O(n \log n)$. The solution, based on an idea proposed by Professor Wojciech Rytter, does not use any complex data structures.

Occasionally, it is easier to solve and prove the correctness of a generalized version of a problem then to do so for the original, more specific problem. Let us consider the following, more general task.

***Task.*** An array $A[1..n]$ of integers from the interval 0 to $n-1$, not necessarily distinct, is given. Find an array $B[1..n]$ such that $B[j]$ is the number of elements in $A$ that are larger than $A[j]$ and are to the left of $A[j]$ in $A$; that is,

$$B[j] = |\{1 \le k < j : A[k] > A[j]\}|.$$

Note that if $A$ is a permutation of $\{1,\ldots,n\}$ then, after subtracting 1 from each of the $A[j]$ and finding $B$, we obtain the code for the permutation $A$.

We will solve the above problem using the following observation:

***Observation.*** For any nonnegative distinct integers $a < b$ the following holds true: $\lfloor a/2 \rfloor = \lfloor b/2 \rfloor$ if and only if $a = 2k$ and $b = 2k + 1$, for some integer $k \geq 0$.

What does this observation imply? If we divide (integer division) each $A[j]$ by 2, then the number of pairs $(A[l], A[p])$ such that $A[l] > A[p]$ and $l < p$ will not change, not counting pairs that, before the division operation, were $A[l] = 2k + 1$ and $A[p] = 2k$, for some integer $k \geq 0$. This observation can be used in our solution.

While the array $A$ contains at least one element larger than 0, for each even $A[j]$ compute the number of elements (they are odd) that are larger by 1 than this element and are to its left. Add this number to the total number of elements larger than the element at position $j$ in the original array $A$ and located to its left (in positions with indices smaller than $j$). Next, divide (integer division) each element in $A$ by 2 and repeat the above process. The program below summarizes the above ideas. The array $Odd[0..n]$ stores the count of odd numbers that are present in the (changing) array $A$ during the scanning of this array from left to right.



```
Algorithm EncodePermutation(A)
    for i := 1 to n do
        B[i] := 0
    HowManyZeros := 0
    while HowManyZeros < n do
        for i := 0 to n do
            Odd[i] := 0
        HowManyZeros := 0
        for i := 1 to n do
            if 2 ∤ A[i] then
                Odd[A[i]] := Odd[A[i]] + 1
            else
                B[i] := B[i] + Odd[A[i] + 1]
            A[i] := A[i] div 2
            if A[i] = 0 then
                HowManyZeros := HowManyZeros + 1
    return B
```

Each iteration of the **while** loop takes $O(n)$ time. Since the number of iterations is at most $\lceil \log(n + 1) \rceil$, the resulting running time of the entire algorithm is $O(n \log n)$.

# / Rooks

**Contest:** 3rd Polish Olympiad in Informatics
**Task author:** Krzysztof Diks
**Memory:** 32 MB
http://main.edu.pl/en/archive/oi/3/wie

On a chessboard of size $n \times n$ we place $n$ rooks. The arrangement of the rooks should satisfy the following rules:

- For each $i = 1,\ldots,n$, the $i$-th rook can be placed within a rectangle specified by two pairs of coordinates: $(a_i, b_i)$, $(c_i, d_i)$, where $(a_i, b_i)$ are the coordinates of the square in the upper left corner of the rectangle (row, column), $(c_i, d_i)$ are the coordinates of the square in the lower right corner of the rectangle, $1 \le a_i \le c_i \le n$ and $1 \le b_i \le d_i \le n$. The square in the upper left corner has coordinates $(1, 1)$; the square in the lower right corner has coordinates $(n, n)$.
- No two rooks can attack each other, i.e. they cannot appear in the same row or the same column.

## Task

Write a program that:

- reads from the input the size of the chessboard $n$ and for each $i = 1,\ldots,n$ the coordinates of the rectangle in which the $i$-th rook can be placed,
- verifies whether the rooks may be placed in appropriate rectangles such that no two of them may attack each other and, if so, finds one such arrangement,
- writes to the output one arrangement of rooks that satisfies the given conditions, or the single word NIE (Polish for *no*) if no such arrangement exists.

## Input

In the first line of the input there is a positive integer $n$ ($1 \le n \le 3000$). In each of the following $n$ lines appear four positive integers not greater than $n$, separated by single spaces. The numbers in the $i$-th line are the coordinates of the rectangle in which the $i$-th rook may be placed ($a_i$, $b_i$, $c_i$ and $d_i$ respectively).



## Output

The output should comprise either one word, NIE, or a series of $n$ lines on each of which are written two integers separated by a single space. The numbers in the $i$-th line should specify the position of the $i$-th rook (row, column). Each such position should be within the rectangle specified by the coordinates in the $(i + 1)$-th line of the input. Pay attention to the fact that the positions of the rooks should be written in the same order as the coordinates of the rectangles are specified in the input.

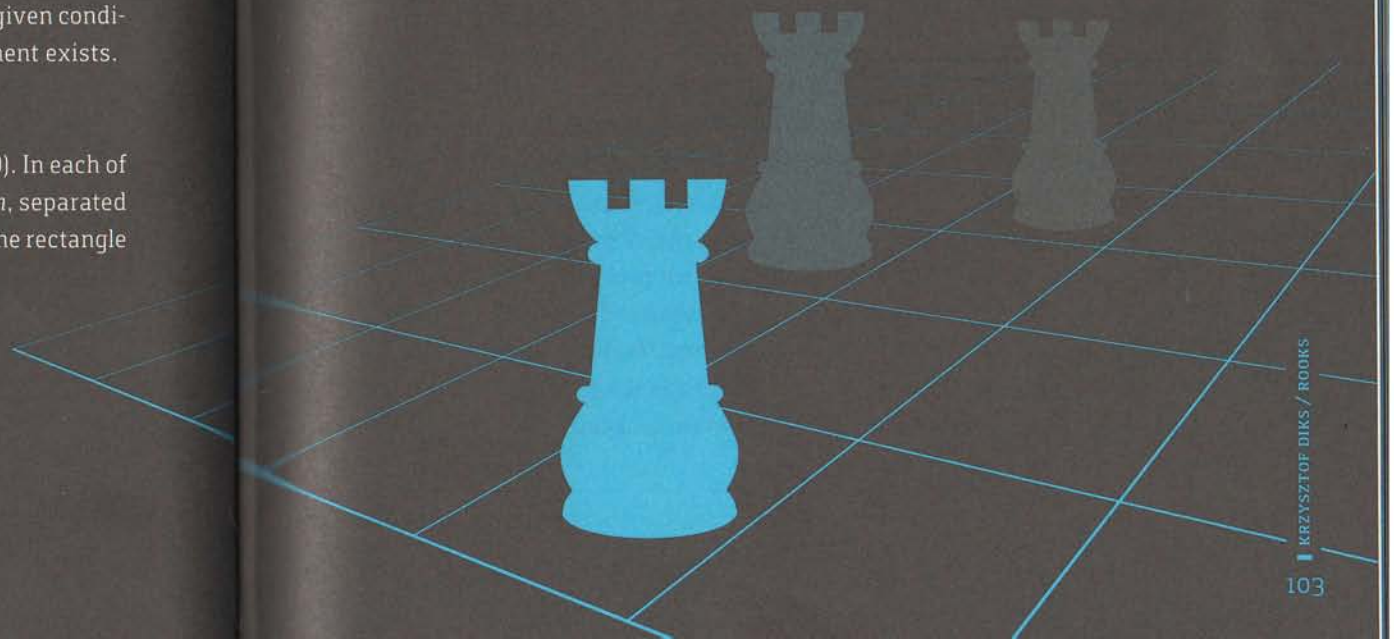
## Example

For the input data:

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

the correct result is:

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



# / Solution

Although the problem is formulated in two dimensions, we will show that it can be reduced to one dimension. Note that if a desired arrangement of rooks exists—call it an *admissible* arrangement—then there is at most one rook in each row and, similarly, at most one in each column. Let $W_i$ denote the $i$-th rook. Indices of the rows where $W_i$ can be placed belong to the closed interval $[a_i..c_i]$ of integers, and indices of columns where $W_i$ can be placed are in the interval $[b_i..d_i]$. Let $[w_1, w_2, \ldots, w_n]$ be an arbitrary permutation of the row indices (numbers $1,\ldots,n$) such that, for each $i$, $w_i \in [a_i..c_i]$; that is, an admissible choice of rows for the rooks. Similarly, let $[k_1,k_2,\ldots,k_n]$ be an arbitrary permutation of the column indices (numbers $1,\ldots,n$) such that $k_i \in [b_i..d_i]$; that is, an admissible choice of columns for the rooks. Then a sequence of pairs $[(w_1,k_1),(w_2,k_2),\ldots,(w_n,k_n)]$ is an admissible arrangement of the rooks on a chessboard. Moreover, every admissible arrangement of the rooks on a chessboard is a combination of an admissible choice for rows and an admissible choice for columns. Therefore, it is sufficient to solve the following problem:

***Problem.*** A positive integer $n$ and $n$ intervals $J_i = [l_i..r_i]$, $1 \le l_i \le r_i \le n$, of integer numbers are given. The task is to check whether there is a permutation $[p_1, p_2, \ldots, p_n]$ of $1, 2, \ldots, n$ such that each $p_i \in [l_i..r_i]$. The number $p_i$ will be called a *representative of the interval* $J_i$. In other words, we are asking if there is a sequence of different representatives for the intervals $J_i$. Provide one such sequence if any exists.

Instead of assigning representatives to intervals, let us reverse the assignment. Each representative $1, 2, \ldots, n$ will be assigned an interval to represent. What interval can safely be assigned to 1, and what does "safely" mean? The answer is simple. The number 1 can be assigned to an interval such that the remaining intervals can be assigned to the representatives $2, \ldots, n$. Firstly, the interval has to contain 1. There may be many such intervals, however, so which one to choose? It is not hard to see that the safest choice is to select the shortest interval; that is, the one with the smallest right endpoint. Why? Assume that $J = [1..r]$ is the shortest interval containing the element 1 and, in some admissible assignment of intervals to the representatives, $J$ is assigned to $x > 1$. Now, let $J' = [1..r']$ be an interval assigned to 1. Since $J$ is the shortest, $r' \ge r$. After swapping the assignments for $J$ and $J'$, we obtain an admissible assignment with $J$ assigned to 1.



Having an interval for the representative 1, we turn to 2. To 2 we assign the shortest interval containing 2, not counting the one selected earlier for 1. We repeat this process for 3, 4 and so on, selecting from the intervals not previously assigned.

This process can be formalized as follows:

```
Algorithm ArrangeRooks
  S := ∅
  for i := 1 to n do
    P := a set of intervals with the left endpoint in i
    S := S ∪ P
    if S = ∅ then
      print NO; exit
    else
      I := an interval in S with the smallest right endpoint
      set i as the representative for I
      S := S \ {I}
```

It remains to be decided how to implement this algorithm. In essence, there are two issues to address: how to compute the set $P$ quickly, and how to implement the dynamic set $S$ and operations on it.

Computing $P$ is not a complex task. It is sufficient to sort all the intervals with respect to the left endpoints and group together intervals with the same left endpoint. Since the endpoints are integers from $[1..n]$, such sorting can be done in $O(n)$ time.

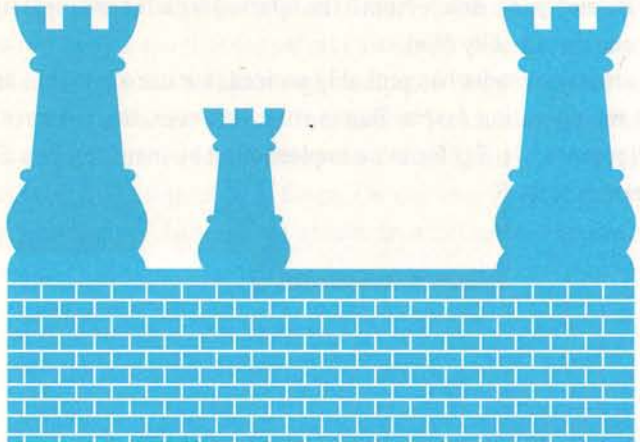
Now let us take a look at the set $S$. It consists of intervals with integer endpoints from $[1..n]$. The following operations are performed on $S$:

- *Init*($S$): initialize $S$ as an empty set,
- *Insert*($S, I$): insert interval $I$ to $S$,
- *Min*($S$): return a pointer to the interval in $S$ with the smallest right endpoint,
- *DeleteMin*($S$): delete from $S$ the interval with the smallest right endpoint (the one returned by *Min*).

An attentive reader has probably noticed that our algorithm does not explicitly call the operation *Insert*. This is true; however, the abstract operation in the assignment $S := S \cup P$ can be implemented by inserting into $S$ the consecutive intervals from $P$.

A set $S$ with the operations *Init*, *Insert*, *Min* and *DeleteMin* is nothing more than a priority queue organized with respect to the right endpoints of the intervals. Such a priority queue can either be implemented directly with a heap (a data structure used, for example, in the heapsort algorithm) or using an implementation from STL. The running time of the *Insert* and *DeleteMin* operations in such implementations is $O(\log k)$, where $k$ is the number of elements in the priority queue; and the operation *Min* requires constant time. Consequently, the worst-case running time of the above implementation of the algorithm for finding interval representatives is $O(n \log n)$. Since we reduced our original problem to two independent tasks for intervals, we have obtained a solution for the *Rooks* problem with $O(n \log n)$ running time.

Finally, the *Rooks* problem offers yet another advantage: the task can be viewed as a good introduction to the highly interesting problem of finding representatives for families of finite sets that are not necessarily intervals of integers. This general problem—known as a *system of distinct representatives*—can be solved, for example, with graph matching algorithms; but that's another story.



**ANDRZEJ GĄSIENICA-SAMEK**

Andrzej Gąsienica-Samek graduated from the Faculty of Mathematics, Informatics and Mechanics, University of Warsaw with a degree in informatics. He is a three-time winner of the Polish Olympiad in Informatics, and won one silver and three gold medals at the International Olympiad in Informatics.

In 2003 Andrzej Gąsienica-Samek and his team ranked first in the ACM-ICPC World Finals in Beverly Hills. Two years earlier, at the ACM-ICPC World Finals in Vancouver in 2001, he and his team had achieved sixth place.

For five years Andrzej Gąsienica-Samek worked at Comarch SA, one of the largest Polish IT business solutions providers, where he held positions of department director and R&D center director. During his time there he was responsible for managing departments covering R&D and software deployment in the public administration sector. Since 2008 he has been CEO at Atinea, a company that deals primarily with tailor-made IT solutions and currently employs 30 programmers. Its core activity encompasses R&D in the field of flexible database systems.

Andrzej Gąsienica-Samek is interested in digital intelligence. Even though he claims that the current state of scientific knowledge in this sphere has little to do with actual intelligence, his long-term goal is to create a program that would be capable of "thinking" on its own.

# / Monkeys

**Contest:** 10th Polish Olympiad in Informatics
**Task author:** Andrzej Gąsienica-Samek
**Memory:** 32 MB
http://main.edu.pl/en/archive/oi/10/mal

There are $n$ monkeys on a tree, numbered from 1 to $n$. Monkey number 1 clutches a branch by its tail. The remaining monkeys are held by other monkeys, or hold on to other monkeys, or both. Each monkey can use two hands and can hold at most one other monkey in each hand (gripping its tail). Starting from moment 0, every second, one monkey releases the grip of one hand. This may cause some monkeys to fall down to the ground, where they can continue releasing their grips (the time taken to fall is negligibly small).

### Task

Write a program that:

- reads from the input the description of how the monkeys hold together, and in what order they release their grips,
- for each monkey, computes the time when it falls down to the ground,
- writes the result to the output.

### Input

The first line of the input consists of two positive integers, $n$ and $m$ ($1 \le n \le 200{,}000$, $1 \le m \le 400{,}000$). The number $n$ specifies the number of monkeys and the number $m$ specifies the time (in seconds) when we observe the monkeys. The next $n$ lines contain the description of the initial situation. In the $(k+1)$-th line ($1 \le k \le n$) there are two integers specifying the numbers of the monkeys that are held by monkey number $k$. The first integer is the number of the monkey that is held by the left hand; the second is the number of the monkey held by the right hand. A value of $-1$ indicates that the monkey's hand is free. The following $m$ lines record the result of the observation of the monkeys. In the $i$-th of those lines ($1 \le i \le m$) there are two integers. The first is the number of the monkey and the second is the number of the hand (1 = left, 2 = right) with which the monkey releases its grip in the moment $i - 1$.

### Output

Your program should write to the output exactly $n$ integers, one per line. The number in the $i$-th line should specify the moment the $i$-th monkey fell down to the ground, or should be $-1$ if the monkey did not fall to the ground during the observation.

### Example

For the input data:

```
3  2
-1  3
3  -1
1  2
1  2
3  1
```

the correct result is:

```
-1
1
1
```

# / Solution

The problem *Monkeys* was inspired by the mechanism of *garbage collection* that was made popular by the introduction of the Java programming language in the late nineties. Garbage collection frees memory that is no longer referenced by the running program.

Many programmers confuse memory management with resource management. The fundamental difference between these two mechanisms is in the requirements: unlike the release of memory, which may happen with a delay unknown to and unpredictable by a programmer, files or network connections must be released at precisely determined points.

A good understanding of the problem *Monkeys* helps to explain why the mechanism of garbage collection is not efficient for general resource management. In addition, the problem throws light on the misconception that garbage collection solves the issue of memory leaks.

**Monkeys and garbage collectors**

Let us try to model garbage collection using the terminology of object-oriented languages. Specifically, we have:

- a stack of activation records,
- objects,
- references (to objects), which can be object fields or variables allocated on the stack.

A program has access to only those objects that are referenced through a path of references that originates in the stack. Other objects are no longer accessible to the program and should be released (for example, see objects *D* and *E* in figure 1).

What is the connection between the above model and the *Monkeys* problem? The monkeys represent objects, the stack corresponds to the tree on which monkey 1 is hanging, and references correspond to the hands with which monkeys hold the tails of other monkeys.

The problem *Monkeys* asks for the exact moment when a group of monkeys falls. This question is analogous to asking when an object becomes unavailable to a program. In a similar way to knowing the exact moment when each of the monkeys drops down, we would like to release memory as soon as it becomes unreferenced. This would allow us to apply garbage collection to general resource management.



**Garbage collection and resource management**

Attempts to employ a garbage collector in general resource management lead immediately to efficiency issues, since finding which objects may be released is not a simple task. This stems from the fact that a change to a single reference may potentially allow us to release an arbitrarily large number of other objects. For example, if a stack frame for some function contains a list object that, in turn, contains a thousand other objects, then upon a return from this function the only reference to the list will be lost and the list object, as well as all the objects in this list, should be released.

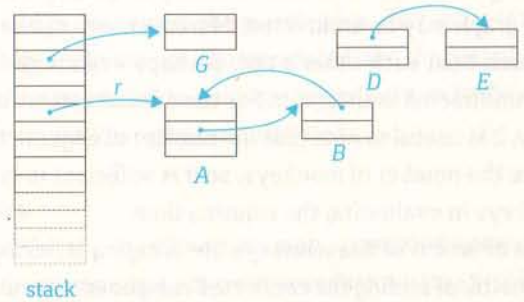



***Figure 1:*** *A schematic view of garbage collection. Rectangles represent objects; arrows represent references.*

The first idea that comes to mind is to release the object as soon as the reference to the object is popped from the stack, or if this reference is modified and starts pointing to another object. Of course, such a strategy would be effective for the aforementioned example of the list, but in general more than one reference can point to the same object.

Refining the above idea, we could associate with each object a *reference counter* to store the number of references pointing to that object, and update the counter each time references change. Thanks to reference counters, we would be able to release memory bound to objects as soon as the last reference disappeared, and a similar approach would also work for resource management. In fact, this strategy is used by some programming languages; for example, Perl.

Unfortunately, chains of references may form cycles. Object *A* may refer to some object *B* that, in turn, has a reference to *A*. With reference counters, we would never release objects *A* and *B* (there are references to both of them) even if they become unavailable to the program (in the example from figure 1, this would happen after reference *r* had been popped from the stack).

Identification of objects that are no longer available to the program requires all the objects and references to be checked. Such a process is costly and is therefore performed by garbage collectors only rarely, which leads to delays in releasing memory.

Let us go back to the problem *Monkeys* to see how much time it takes to identify monkeys that fall.

**Monkeys and graph theory**

The monkeys from our problem can be modeled as a graph, with the monkeys represented as vertices and the hands that hold tails as edges. Note that it does not make any difference whether a monkey is holding a tail or having its tail held, so the graph may be undirected. Moreover, one can imagine that two monkeys can each hold each other's tail, perhaps even with both hands, so the graph is an undirected multigraph. For the sake of estimating the running time complexity, it is useful to note that the number of edges in this graph does not exceed twice the number of monkeys, so it is sufficient to consider just the number of monkeys in evaluating the running time.

The question of which of the monkeys are hanging is, using graph theory terminology, a matter of finding the connected component that includes monkey 1. Less formally, starting with monkey 1, we want to visit all monkeys reachable through hands holding tails. This can simply be accomplished with a depth-first search: that is, recursively traversing the hands and memorizing all the visited monkeys.

The running time complexity of such a traversal is proportional to the size of the connected component containing monkey 1. This number is $O(n)$ since, as we noted above, the number of hands is proportional to $n$. Unfortunately, this operation is repeated each time any monkey releases its grip on a tail, resulting in an $O(n^2)$ cost. From the description of the input size, it is clear that an $O(n^2)$ solution is unacceptable for the *Monkeys* problem.

**Returning to a garbage collector**

In view of the above discussion, it makes sense to analyze how garbage collectors do their job, knowing that detecting objects unreachable from the program is so expensive. Firstly, garbage collection does not happen after each change of references, which may unfortunately result in substantial delays in releasing memory. Therefore, if the object has methods such as *close* or *dispose*, the programmer should always use them before losing the last reference to the object. Otherwise, nondeterministic results, depending on the garbage collector, are possible.



Secondly, developers of garbage collectors are focused more on efficiency for cases that can occur in practice than on the asymptotic complexity. Therefore, there are scenarios where the time taken by garbage collection is hard to predict. However, a similar comment applies to the *malloc* and *free* (*new* and *delete*) operations, where attaining good asymptotic efficiency is also not the primary concern.

On the other hand, taking a theoretical rather than empirical view, it is valid to ask about the best running time complexity for garbage collection. Is good efficiency at all possible? It appears that garbage collection can be accomplished in linear time with respect to the size of allocated memory. A simple argument to this end is that, assuming the availability of twice as much memory as is needed for processing and performing garbage collection after space is fully exhausted, the cost of garbage collection can be amortized into the cost of allocation. More interesting problems related to garbage collection can be found by browsing information about *real-time garbage collection*.

**A better solution**

Reasons that inhibit the use of memory management for general resource management have been presented in the above sections. In the problem *Monkeys*, we are asked to identify the exact moments when monkeys fall, and we know that a $O(n^2)$ solution is unacceptable. Let us then take another look at the assumptions of *Monkeys*, with an eye on what makes the problem different from memory management.

The first observation is that monkeys have two hands, unlike the multiple reference fields that are possible for objects. Even so, each object could be split into several pairs (in a similar way as in the programming language Lisp) and not much would change in the conditions for garbage collection. Similarly, we can imagine collapsing two monkeys into one monkey with two tails and three hands, which does not change the problem either.

The second observation is that, during the process, after releasing its grip, a monkey never catches any other tail with its hands. This observation is more interesting than the previous one. If we stop considering falling monkeys, and instead start thinking in terms of a graph from which edges are being removed, then the scenario becomes that of a graph that is slowly decomposing into smaller connected components. Our concern is with the connected component that includes vertex 1. It is computationally costly to find what part of this connected component becomes detached after removing an edge, as it requires all vertices connected to 1 to be visited. They are visited every time an edge is removed, and that contributes to the cost.

However, if instead of splitting the graph we try to build it, the process becomes much more efficient. If all the vertices in the connected component containing vertex 1 are properly marked, we can easily detect if a newly added edge attaches anything new to this component. In constant time, we can check whether both ends of the edge belong to the connected component of vertex 1. If so, nothing else needs to be done. Otherwise, in time proportional to the number of vertices being attached, we mark them as members of the connected component of vertex 1. There is no need to revisit vertices already belonging to this component. Consequently, the operation of adding an edge takes time proportional to the number of attached vertices.

Now, if the entire process is executed backwards—that is, monkeys on the ground are consecutively attached to the tree—then each step takes time proportional to the number of monkeys attached in that step. In the result, this simple solution has $O(n)$ running time, the best possible in view of the input size.

It is important to note that a similar trick is not applicable to memory management, which is much less deterministic than the sequence of events with monkeys. For that reason, programmers should always remember to call explicit *close* or *dispose* methods on objects that provide them. The garbage collector has no way to predict the precise moment when the last reference will disappear. Similarly, we must be careful with a single individual monkey that carries the entire group of monkeys on its tail. In such a case, not even a garbage collector would protect us from memory leaks.

**TOMASZ IDZIASZEK**

Tomasz Idziaszek is a research assistant at the Faculty of Mathematics, Informatics and Mechanics, University of Warsaw. His research focuses on automata theory, with emphasis on connections between logic and tree automata. He has been involved in programming competitions for eleven years. He is a finalist of several individual contests (TopCoder, Google Code Jam, Algorithmic Engagements) as well as team competitions (with Szymon Acedański and Jacek Jurewicz he won the ACM-ICPC Central European Regional Contest in 2004 and represented the University of Warsaw at the ACM-ICPC World Finals). Currently he is more involved in organizing events such as the Polish Olympiad in Informatics, Polish Olympiad in Informatics Training Camp, Polish Collegiate Programming Contest and Algorithmic Engagements (the latter giving him the opportunity to write algorithmic problems for the most challenging round of the contest, which is what he enjoys best).
Moreover, Tomasz Idziaszek is an author and editor of computer science articles in the Polish popular science monthly, *Delta*. In his spare time he enjoys playing the piano.

# / Plotter

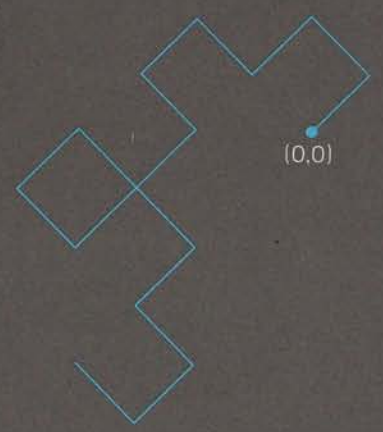
**Contest:** Algorithmic Engagements 2011
**Task author:** Tomasz Idziaszek
**Memory:** 32 MB
http://main.edu.pl/en/archive/pa/2011/plo

To test the functionality of his newly-purchased plotter, Byteasar decided to plot a few bytecurves. A bytecurve of order $n$ consists of $2^n$ segments of length $\sqrt{2}$ each. The first of them connects the points with coordinates $(0,0)$ and $(1,1)$. A bytecurve of order $n$ can be described with a word $L_n$ of length $2^n - 1$ based on a two-letter alphabet {**L**, **R**}. The word's $i$-th letter indicates that, after the $i$-th segment has been printed, the pen turns and moves orthogonally to the left (letter **L**) or to the right (letter **R**) to print the next segment. In other words, the pen changes its direction to the left or to the right by 90°.

$L_1$ consists of a single letter **L** (left turn) and $L_2$ consists of three letters **LLR**, indicating two left turns followed by one right turn. Inductively, $L_n$ is generated from $L_{n-1}$ in the following way: separate all letters in $L_{n-1}$ with spaces and place an additional space in front and at the end of $L_{n-1}$. Then, in the newly-created spaces, place alternating letters **L** and **R**, beginning with **L**. For example, $L_3$ is generated as follows:

$$L_2 = \mathbf{LLR} \rightarrow _\mathbf{L}_\mathbf{L}_\mathbf{R}_ \rightarrow \underline{\mathbf{L}}\mathbf{L}\underline{\mathbf{R}}\mathbf{L}\underline{\mathbf{L}}\mathbf{R}\underline{\mathbf{R}} = L_3.$$

Similarly, $L_4$ = **LLRLLRRLLLRRLRR** (see the figure below).




It takes exactly one second to draw one segment of the bytecurve. While waiting for the plotter to complete printing, Byteasar ponders the following question: given a point $(x, y)$ on the plane, when will the pen be positioned at that point? For example, for the bytecurve of order 4 in the above figure, the pen will be at point $(-3, -1)$ seven seconds after the beginning of plotting, and again eleven seconds after the beginning of plotting. Your task is to answer Byteasar's question.

### Input

The first line of the input contains two integers, $n$ and $m$ ($1 \le n, m \le 2000$), where $n$ indicates that the bytecurve is described by $L_n$ and $m$ indicates that there are $m$ query points. The next $m$ lines each contain two integers, $x_i$ and $y_i$ ($-10^9 \le x_i, y_i \le 10^9$), indicating the coordinates of the $i$-th query point. Note that the coordinates in any particular input line are not guaranteed to represent a point on the bytecurve. Moreover, no point appears more than once in the input.

### Output

The output should consist of $m$ lines containing answers to the consecutive queries. Each answer consists of a nonnegative integer $k_i$, specifying the number of visits the pen makes to the query point $(x_i, y_i)$; followed by $k_i$ nonnegative integers indicating the times of those visits in increasing order, in seconds since the beginning of plotting. Numbers should be separated by single spaces.

### Example

For the input data:

```
4  3
-3  -1
1  1
-1  0
```

the correct result is:

```
2  7  11
1  1
0
```

# / Solution

The bytecurve described in the problem is in fact a fractal, known to a wider audience as a *dragon curve*. It was first studied in 1966 by two physicists from NASA, John Heighway and William Harter, but it was popularized by Martin Gardner in his magazine column *Mathematical Games* and by writer Michael Crichton in his novel *Jurassic Park*.

Figure 1 illustrates how the dragon curve earns its name by showing a few curves of low orders. It can be proven that the dragon curve does not intersect itself (though it can touch itself); therefore the plotter's pen will visit every point at most two times. We denote by $S_n$ the dragon curve of order $n$. We also assume that the first segment of the curve joins points (0,0) and (1,0); thus the points from the input have to be transformed using the formula

$$(x',y') = \left(\frac{x+y}{2}, \frac{y-x}{2}\right).$$

The dragon curve has many equivalent definitions; in the problem statement we encounter just one of them. Another definition states that the word $L_n$ describing the curve satisfies the following recurrence relation:

$$L_1 = \mathbf{L}, \quad L_n = L_{n-1}\,\mathbf{L}\,r(L_{n-1}), \tag{1}$$

where $r$ denotes an operation that reverses the order of letters in the word and simultaneously interchanges letters **L** and **R**; e.g. $r(L_2)=$ **LRR**. We leave the proof of equivalence of both definitions as an exercise for the reader.

Let us denote by $(x_n, y_n)$ the last point on the curve of order $n$. It is easy to show the recurrence that calculates it:

$$x_1 = y_1 = 1, \quad x_n = x_{n-1} - y_{n-1}, \quad y_n = x_{n-1} + y_{n-1}.$$

The formula (1) draws our attention to a very important property of the dragon curve, namely that the curve of order $n$ is obtained by joining two curves of order $n-1$. Speaking more precisely: the curve $S_n$ consists of a curve $S_{n-1}$ and its copy, fixed at point $(x_n, y_n)$ and rotated by 90° clockwise (see again figure 1).



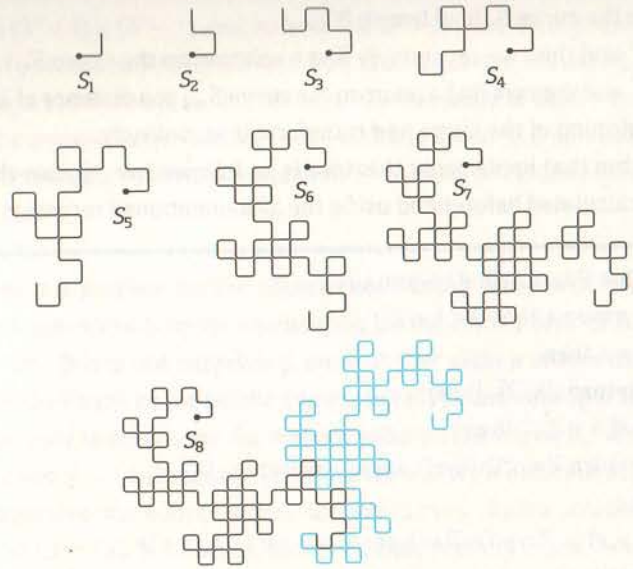



*Figure 1: The family of dragon curves; $S_n$ is the curve of order n. For $S_8$ we have marked out its two component $S_7$ curves.*

**Why do we like fractals?**

Writing a program that generates a picture of a fractal is an ideal task for a beginner programmer. Although the rules of generating fractals are simple, they lead to complicated patterns. With relatively little programming effort one can achieve quite ambitious visual effects. Moreover, the self-similarity of fractals allows us to exercise one of the most fundamental programming concepts—recursion.

It is no wonder that authors of programming contest problem sets like to look to the world of fractals for inspiration. One of the popular kinds of tasks is generating space-filling curves, such as the Peano curve, Hilbert curve or their variants. The size of such curves grows exponentially in the order of the curve, therefore the task is usually stated as follows: for the curve of order $n$ and a number $k$, calculate the coordinates of a point in the plane that lies at a distance of $k$ from the start of the curve. It is expected that the program will run in time complexity $O(n)$.

Usually, such tasks are not conceptually difficult; for example, if we wish to solve such a problem for the dragon curve $S_n$, we need to consider two cases (recall that the curve $S_n$ is of length $2^n$):

(a) $k < 2^{n-1}$ and then we recursively find a solution on the curve $S_{n-1}$,

(b) $k \geq 2^{n-1}$ and then we find a point on the curve $S_{n-1}$ at a distance of $2^n - k$ from the beginning of the curve and transform it accordingly.

The algorithm that implements this idea is as follows (we assume that values $x_i, y_i$ were calculated beforehand using the aforementioned recursion):

```
Algorithm PointOnTheDragonCurve(k, n)
    {we assume that 0 ≤ k ≤ 2^n}
    if n = 1 then
        return (⌈k/2⌉, ⌊k/2⌋)
    else if k < 2^(n-1) then
        return PointOnTheDragonCurve(k, n − 1)
    else
        (x, y) := PointOnTheDragonCurve(2^n − k, n − 1)
        return (x_n + y, y_n − x)
```

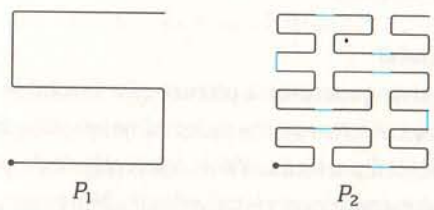



*Figure 2: A Peano curve of order n is constructed by joining nine curves of order $n-1$.*

**Converse problem**

The task can be stated in reverse: given a point in the plane $(x, y)$ that lies on the curve, we are asked at what distance from the start of the curve it lies. Before we show how to solve this problem for the dragon curve, we look at the solution for another fractal: the Peano curve (also known as a second Hilbert curve; see figure 2).

This task appeared in the ACM-ICPC World Finals in 2003 (see the task entitled *Riding the Bus*). Its solution is fairly easy: a Peano curve $P_n$ of order $n$ fits on a grid of size $(3^n - 1) \times (3^n - 1)$, and its length is $9^n - 1$. Considering values $\lfloor x/3^{n-1} \rfloor$ and $\lfloor y/3^{n-1} \rfloor$, we are able to decide to which of nine smaller curves $P_{n-1}$ the point $(x, y)$ belongs. If it belongs to the $i$-th curve (in the order of their appearance in $P_n$) then the answer is the sum of $i \cdot 9^{n-1}$ and a solution to a subtask for curve $P_{n-1}$ and accordingly rotated point $(x \bmod 3^{n-1}, y \bmod 3^{n-1})$. The algorithm runs in time $O(n)$.

The converse problem for the dragon curve is the essence of the task *Plotter*. The task proved to be quite challenging for the contestants of Algorithmic Engagements. This is not surprising, since at first sight it seems that the idea we used for the Peano curve will be useless here. We lack an important piece of knowledge: how to determine, for a given point on the curve $S_n$, which of two smaller curves $S_{n-1}$ it belongs to. Therefore we will try a different approach: let us call a recursive procedure for both smaller curves. Such a solution will run, of course, in time $O(2^n)$. Observe, however, that in many cases, when we look for a point on the wrong curve of smaller order, after a few recursive descents our point will lie so far from the currently examined part of the curve that we will be able to detect it easily and prune the search.

Let us see how this idea works in practice. Denote by $d(x, y, n)$ a function that returns a set of distances at which the point $(x, y)$ lies from the start of the curve $S_n$ (the given point can lie at no more than two distances from the start of the curve). The function will make two recursive calls. Its value is a sum of two sets:

$$d(x, y, n) = d(x, y, n-1) \cup \{2^n - k : k \in d(y_n - y, x - x_n, n-1)\}.$$

For $S_1$ we have

$$d(x, y, 1) = \begin{cases} \{x + y\} & \text{for } 0 \leq y \leq x \leq 1, \\ \varnothing & \text{otherwise.} \end{cases}$$

When can we prune the search? We cannot easily check *whether* a point belongs to the curve $S_n$, but for many points we can say that they are, without doubt, *not* on the curve. Indeed, consider a bounding box $R_n$, i.e. the smallest rectangle with sides parallel to the axes of the coordinate system that contains all points of the curve $S_n$. Every time the function $d(x, y, n)$ is called with a point

$(x, y)$ that lies outside the bounding box $R_n$, we can immediately return $\varnothing$. Let $r_n$, $t_n$, $l_n$, $b_n$ denote the distances of point $(0,0)$ from the sides of the rectangle $R_n$ (right, upper, left and lower side respectively). We can calculate them recursively:

$$
\begin{aligned}
r_1 = t_1 = 1, \quad l_1 = b_1 = 0,\\
r_n &= \max(r_{n-1}, t_{n-1} + x_n),\\
t_n &= \max(t_{n-1}, l_{n-1} + y_n),\\
l_n &= \max(l_{n-1}, b_{n-1} - x_n),\\
b_n &= \max(b_{n-1}, r_{n-1} - y_n).
\end{aligned}
$$

We still need to analyze how the above optimization changes the running time of the algorithm. Empirical tests suggest that the effect is quite significant. This is the reason why some contestants implemented similar ideas but were unable to show why their solutions worked so quickly. We shall now show that the above optimization improves the time complexity of the algorithm from exponential to linear!

Let us draw the curve $S_n$ and mark out all bounding boxes of $2^{n-m}$ curves $S_m$ that form the curve $S_n$. Let us estimate how many, at most, drawn bounding boxes can enclose point $(x, y)$. This number will reveal how many times recursive calls to $d(\cdot, \cdot, m)$ will not be pruned. If this number is a constant then, consequently, at every level of recursion we will have a constant number of calls; thus the whole algorithm will run in time $O(n)$.

It can be proven by induction that $x_m$ and $y_m$ are divisible by $M = 2^{\lfloor m/2 \rfloor}$, thus every curve $S_m$ begins at one of the points from the set $Z_M = \{(Mi, Mj) : i, j \in \mathbb{Z}\}$. It can be also shown that

$$r_m + l_m, t_m + b_m < 2^{\lfloor m/2 \rfloor + 1},$$

therefore every drawn bounding box is enclosed in a square of side $2M - 1$. That means that bounding boxes that contain a point $(x, y)$ cover in total at most 16 points from the set $Z_M$. At each of these points, at most four curves $S_m$ can begin; thus $(x, y)$ belongs to no more than $16 \cdot 4$ bounding boxes. Therefore our algorithm does indeed run in linear time.



# / Termites

**Contest:** Algorithmic Engagements 2010
**Task author:** Tomasz Idziaszek
**Memory:** 256 MB
http://main.edu.pl/en/archive/pa/2010/ter

Two termites are eating an old wooden fence. It consists of $n$ planks of possibly different heights. Termites have already eaten some of them, and they thought that they should make their meal more interesting. They have decided to play a game and take turns to eat the planks, one by one. During one turn, a termite may only choose to eat a plank which is next to a plank that has already been consumed. Assuming that each termite chooses the planks in such a way that, after the whole game, the sum of the heights of all the planks eaten by that termite is as large as possible, compute the amount of wood that each of them will have eaten.

## Input

In the first line of the input there is an integer $n$ ($1 \le n \le 1{,}000{,}000$), specifying the number of planks in the fence. The second line contains a sequence of $n$ integers $l_i$ ($0 \le l_i \le 1{,}000{,}000{,}000$), describing the heights of the consecutive planks. If $l_i = 0$, then the corresponding plank has already been eaten. Plank number $i$ (for $1 < i < n$) is adjacent to planks $i - 1$ and $i + 1$. The only plank adjacent to plank number 1 is plank number 2, and the only plank next to plank number $n$ is plank number $n - 1$. At least one number $l_i$ is equal to zero.

## Output

You should write two integers to the output. The first one should be the sum of the heights of the planks that will be consumed by the termite that starts the game, whereas the second should specify the amount of wood that will be eaten by the other termite.

## Example

For the input data:

```
8
1 2 0 3 7 4 0 9
```

the correct result is:

```
17 9
```

**Explanation of the example:** The fence consisted of eight planks, of which two have already been eaten. The first termite, on its first turn, can choose between planks of height 2, 3, 4 or 9. During an optimal game, in consecutive turns, the termites will eat planks of height 9, 2, 1, 4, 7 and 3.

# / Solution

If anyone asked me which of my own problems I like the most, I would answer *Termites* without hesitation. It's not every day that one is lucky enough to come up with a problem with a simple, neat statement that causes disbelief among experienced contestants, who are certain that the author has made a mistake in the constraints of the input, since the problem apparently cannot be solved effectively.

Yet *Termites* does have a solution: a solution, moreover, that is short and quite elegant. My experience with this problem proves that everybody learns throughout their whole lives, and even now there are algorithmic pearls that are unknown to us, still waiting to be discovered.

A considerable amount of time passed between the original conception of this task and its presentation, in its current form, at the weekend round of Algorithmic Engagements in 2010. Its history began in 2005 at the International Olympiad in Informatics, which was held in Poland in the city of Nowy Sącz. During this event, Jakub Radoszewski shared with me a problem that he could not solve effectively.

***Problem 1.*** We have $n$ stacks of coins. Every coin is either a golden ducat or a worthless penny. There are two players in the game who collect coins alternately. In one move, a player can grab one coin from the top of any stack. We are to determine what the outcome of the game will be if both players play optimally in an attempt to maximize the number of collected ducats.

As we will see later, assumptions that occur in the statement of *Termites*—that the heights of the planks are positive and that at most two sides of the fence could be inaccessible—are of little importance. Therefore, the above problem is a simplified version of *Termites*. Since the problem looked interesting to me, I decided to attack it. Unfortunately, over the next few years I was not able to tackle it, except for a few hypotheses, which alone did not draw me any nearer to the solution (e.g. that it does not pay off to take a penny when one could take a ducat), and several not-so-clever tricks that decrease the exponent in an obvious exponential solution based on dynamic programming.

However, five years later, when motivated by the approaching Algorithmic Engagements, I sat down with this task and decided to work for as long as I could to make appreciable progress. My persistence was rewarded: a polynomial algorithm was developed in a week, and I spent the following month proving its correctness. What is more, I noticed that the algorithm is general enough that it also works in the case where we can grab coins from both ends of a stack. This observation was a little surprising to me, since I had encountered a similar problem before...

**From mathematical folklore: Coins in a Row**

This problem is surely known to many readers. I ask for a little more patience: for now, we are only presenting new problem statements, but soon we will start to solve them.

***Problem 2 (Coins in a Row).*** There are $n$ coins in a row, each with a value. There are two players in the game, Annie and Bertie, who alternately take one coin from the left or right end of the row. Annie makes the first move. What will be the outcome of the game (in terms of the sum of collected values) when both children play optimally?

As we can see, this problem is also a simplified version of *Termites* in which we have only one part of the fence, accessible from both sides. Let us try to solve this variant before we move on to the general case.

Firstly, let us make certain that the greedy strategy—which consists of always taking the most valuable available coin—does not work. Indeed, consider the following sequence of coins:

5 7 10 6

If Annie takes 6 in her first move, then the sum of values of coins collected by her, reduced by the sum accumulated by Bertie, will be equal to $6 - 10 + 7 - 5 = -2$. On the other hand, grabbing 5 gives Annie an advantage over Bertie of $5 - 7 + 10 - 6 = 2$. In the following discussion we use the term *value of the game* to denote the difference between the sum of values collected by Annie and the sum accumulated by Bertie when they play optimally. It is easy to see that Annie's goal is to maximize the value of the game, while Bertie tries to minimize it.

So, the naïve greedy strategy does not work. It turns out, however, that in the case where the number of coins is even, there is a very simple strategy that the first player can use to ensure that he does not lose (i.e. the first player will get at least the same value of coins as the opponent). Moreover, the existence of this strategy does not depend on the values of the coins. Now, for even $n$, Annie can be sure to take all the coins that are in the odd positions in the row,

or all the coins in the even positions. Thus she can choose whichever variant yields the bigger sum. Let us look at the example:

$$\underline{5}\ 2\ \underline{1}\ 4\ \underline{9}\ 8$$

In this case, the sum of the coins' values in the odd positions (underlined) is $s_1 = 5 + 1 + 9 = 15$, and the sum of the even positions (not underlined) is $s_2 = 2 + 4 + 8 = 14$. Annie could employ the strategy of taking coins from non-underlined positions, which gives her an advantage of $s_1 - s_2 = 1$. The strategy exists because the following invariant holds: before each of Annie's moves, exactly one end of the row has an underlined coin, and before each of Bertie's moves, both ends have non-underlined coins.

Therefore, we see that, for even $n$, Annie never loses. Let us observe, however, that the aforementioned strategy is not necessarily optimal. Indeed, in the above example Annie could get an advantage of 3. The optimal strategy is to take 5 in the first move and then play greedily. This leads to the optimal play of sum $5 - 8 + 9 - 4 + 2 - 1 = 3$. Thus the value of the game in this case is 3.

So what do we have to do if we want to compute the value of the game? We can calculate it in time $O(n^2)$ using dynamic programming. It is sufficient to calculate the value of the game for every continuous subsequence of coins. If we denote by $d[i,j]$ Annie's advantage for a sequence of coins in positions $i, i+1, \ldots, j$ $(1 \le i \le j \le n)$, then the values in the array $d$ could be computed by using the following recursive formula:

$$d[i,j] = \begin{cases} a[i] & \text{for } i = j, \\ \max(a[i] - d[i+1,j], a[j] - d[i,j-1]) & \text{for } i < j. \end{cases}$$

I first encountered this problem at the beginning of my experience with programming competitions. It was a classic example that illustrated the dynamic programming method, and the above $O(n^2)$ algorithm was presented as its exemplary solution. I wondered at the time whether a faster solution could be found, but nobody I talked to about the problem knew a better one. Nobody claimed that a better solution did not exist, though...



**An optimal solution for Coins in a Row**

Finally, after my many years of uncertainty, it turned out that a better algorithm could indeed be developed. We show now how to solve this problem in the optimal time complexity of $O(n)$.

For a given row of coins $G$ we denote by $\mathrm{val}(G)$ the value of the game. Assume, for simplicity, that in the row $G$ there is exactly one coin of the greatest value $M$. Since both players try to maximize the sum of collected values, it is clear that both of them would like to grab the coin $M$. Let us consider who has the chance to do it.

It turns out that, although the naïve greedy strategy does not work in general, in situations of extreme greediness it is correct:

***Theorem 1 (Greedy Move Principle).*** Let the coin of the greatest value $M$ be on the end of a row $G$. Let $G'$ be the row after removing this coin. Then $\mathrm{val}(G) = M - \mathrm{val}(G')$.

Thus, every time we can take the biggest value, it is profitable to do so—such a move will be optimal. Moreover, we could strengthen this theorem: this will be the only optimal move.

However, most of the time we will not be lucky enough to utilize the Greedy Move Principle directly. Thus, we need to consider what to do when the greatest value $M$ lies in the middle of the row. At some point, one of the players (say Bertie) during their move will uncover the value $M$. Then, from the strong Greedy Move Principle, we know that $M$ must be taken immediately by Annie. Observe, though, that in these two moves Bertie gains fewer points than Annie. The reason why he would make such a move in the first place is a desire to take the coin that is uncovered after taking $M$. This rationale lies behind the key theorem, which will allow us to calculate the value of the game:

***Theorem 2 (Fusion Principle).*** Let $x$, $M$, $y$ be adjacent coins (in this order) in the row $G$, satisfying the condition $x, y \le M$. Let $G'$ be the row in which these three coins have been replaced by one coin of value $x - M + y$. Then $\mathrm{val}(G) = \mathrm{val}(G')$.

Observe that the statement of the above theorem is much stronger than the presented insight. We do not assume that $M$ is the greatest value in the row, but only that it is greater than the values of adjacent coins. The action of replacing coins $x$, $M$, $y$ by one coin of value $x - M + y$ corresponds to the situation where two outermost coins are taken by one player and the middle one is collected by the opponent.

The Fusion Principle leads us to the following algorithm: as long as we can find three adjacent coins in the row that satisfy the premises of the theorem, we

replace them by one coin. That will shorten the sequence and will not change the value of the game. Let us look at an example in which the theorem is applied three times:

$$\underbrace{5\ 7\ 10\ \underbrace{6\ 9\ 9}_{6}\ 1}_{3}\ \underbrace{7\ 11\ 8}_{4}$$

$$\downarrow$$

$$5\ 3\ 1\ 4$$

If we cannot make fusions any more, we know that the coins' values form a bitonic sequence, i.e. a sequence which is decreasing to some point and then increasing. In particular, it has the nice property that no matter how we take coins from the sequence, the current greatest value will be at one end. Thus we can make use of the Greedy Move Principle, which means that calculating the value of the game for such a sequence is very easy: if $a_1,\ldots,a_m$ are the values in increasing order from the bitonic sequence $G$, then the value of the game is equal to

$$\mathrm{val}(G) = \sum_{i=1}^{m} (-1)^{i+1} a_i.$$

In the above example it would be $5-4+3-1=3$.

The last step is to implement the algorithm effectively. We can perform fusions in an arbitrary order. One approach is to put subsequent values on a stack and check whether a fusion can be made for the topmost three elements on the stack. This leads to an algorithm running in $O(n)$ time. Determining the increasing order of values boils down to merging two sorted sequences that comprise a bitonic sequence; thus it can also be done in linear time. The algorithm in pseudocode is as follows:

```
Algorithm CoinsInARow((a_1, ..., a_n))
    m := 0
    s := ∅
    for i := 1 to n do
        m := m + 1
        s[m] := a_i
        while m ≥ 3 and s[m − 2] ≤ s[m − 1] and s[m − 1] ≥ s[m] do
            s[m − 2] := s[m − 2] − s[m − 1] + s[m]
            m := m − 2
    value := 0
    sign := 1
    i := 1; j := m
    while i ≤ j do
        if s[i] ≥ s[j] then
            value := value + sign · s[i]
            i := i + 1
        else
            value := value + sign · s[j]
            j := j − 1
        sign := −sign
    return value
```

**Solution for an arbitrary number of rows of coins**

Let us try to generalize Problem 2 and think about a solution for the case where there is more than one row of coins. Using dynamic programming, it is easy to come up with a solution that works in time complexity $O(kn^k)$, where $k$ is a doubled number of rows. Obviously this solution is not acceptable.

It turns out, however, that it is easy to modify our optimal algorithm for the single row of coins in order to enable it to cope with an arbitrary number of sequences of coins. This is because the Greedy Move Principle and the Fusion Principle still work in the general case, i.e. their statements are valid even after changing the word "row" to "collection of rows".

Therefore, we are very close to the full solution. We only need to consider the case of rows in which we can only grab coins from one end. Observe that if the greatest value $M$ is at the inaccessible end of the row, and it is adjacent to some value $x$, then it would be fruitless for a player to take $x$, since then the opponent

would grab $M$ (from the strong Greedy Move Principle) and the sequence will be emptied. Thus the following theorem is true:

***Theorem 3 (Worthless Move Principle).*** Let $M$, $x$ be adjacent coins in the collection of rows of coins $G$, where the value $M$ is at the inaccessible end of sequence and $x \leq M$. Let $n$ be the total number of coins in $G$. Let $G'$ be the collection of rows after removing these two coins. Then $\mathrm{val}(G) = \mathrm{val}(G') + (-1)^n(x - M)$.

Again, this theorem is also true in the case where $M$ is simply not smaller than $x$. Which player is forced to make a worthless move depends on the parity of the number of coins $n$—for even $n$ it is Annie.

Ultimately, the algorithm is as follows. For as long as we can, we apply the Fusion Principle, which causes every sequence of coins to become unimodal. Then, for as long as we can, we apply the Worthless Move Principle. This means that we will be able to apply the Greedy Move Principle until the end of the game.

This algorithm can be implemented with time complexity $O(n \log n)$ if we use sorting to obtain the order of coins in the second phase of algorithm, or $O(n \log k)$, where $k$ is the number of accessible ends, if we use an algorithm that merges $k$ sorted lists.

**Practice and theory**

At this point, readers will certainly have realized that the generalized version of the *Coins in a Row* problem, after adding an entomological setting, is indeed the task *Termites*.

Practical readers can implement the above algorithm and check for themselves that it works. However, to fully justify its correctness, we still need to prove three theorems. Unfortunately, though their proofs are not very difficult, they are quite long and technical. Therefore we have decided not to include the proofs in the book. Any reader who is interested in seeing the formal proofs can find them on the author's website*.

* http://www.mimuw.edu.pl/~idziaszek/termity

**GRZEGORZ JAKACKI**

Grzegorz Jakacki graduated from the Faculty of Mathematics, Informatics and Mechanics, University of Warsaw. Over the years he has taken part in diverse software development projects in Poland, the United States and China. In 2001 he was one of the original authors of the System Verilog Assertions, a protocol description language that became the industry standard for VLSI design tools. In 2005, working on a massive hiring campaign in Beijing, China, he invented a way of automating the screening of software developers based on methods derived from programming competitions. This experience eventually led him to establishing his own start-up business, which subsequently gained worldwide recognition under the name Codility.com. Grzegorz Jakacki was a bronze medalist in the International Olympiad in Informatics in 1993, and has also co-organized multiple programming contests. He occasionally teaches classes in advanced programming at the University of Warsaw. He is a founding member of the Warsaw Hackerspace.

# / Window

**Contest:** 5th Polish Olympiad in Informatics
**Task author:** Wojciech Guzicki
**Memory:** 32 MB
http://main.edu.pl/en/archive/oi/5/okn

We have a polygon, chosen in the Cartesian coordinate system, whose sides are all parallel to the axes of the coordinates. Every two consecutive sides are perpendicular, and the coordinates of every vertex are integers. We also have a window, which is a rectangle whose sides are parallel to the axes of the coordinates. The interior of the polygon (but not its periphery) is colored blue. What is the number of separate blue fragments of the polygon that can be seen through the window?

We assume that the boundary of the polygon is a simple polygonal chain; i.e. every vertex of the polygon belongs to exactly two (consecutive) sides, and every other point of the boundary belongs to exactly one side.

**Example**

Look at the figure below:

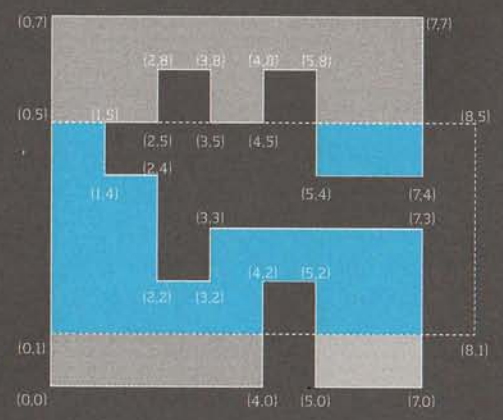



*Figure: There are two separate fragments of the polygon that can be seen through the window.*



**Task**

Write a program that:

- reads descriptions of a window and a polygon from the input,
- computes the number of separate blue fragments of the polygon that can be seen through the window,
- writes the result to the output.

**Input**

In the first line of the input there are four integers, $x_1, y_1, x_2, y_2$ ($1 \le x_1, y_1, x_2, y_2 \le 10{,}000$), separated by single spaces. The numbers $x_1, y_1$ are the coordinates of the top-left corner of the window. The numbers $x_2, y_2$ are the coordinates of the bottom-right corner of the window.

The next line of the input contains one integer $n$ ($4 \le n \le 5000$), which equals the number of vertices of the polygon. In the following $n$ lines, the coordinates of the polygon's vertices are given in counterclockwise direction; i.e. the interior of the polygon is on the left side of its periphery when we move along the sides of the polygon according to the given order. Each line contains two integers $x$, $y$ separated by a single space ($0 \le x \le 10{,}000$, $0 \le y \le 10{,}000$). The numbers in the $(i+2)$-th line ($1 \le i \le n$) are the coordinates of the $i$-th vertex of the polygon.

**Output**

In the first and only line of the output there should be one integer—the number of separate blue fragments of the polygon that can be seen through the window.

**Example**

For the input data (split into two columns):

```
0 5 8 1
24
0 0
4 0
4 2
5 2
5 0
7 0
7 3
3 3
3 2
2 2
2 4
1 4
1 5
2 5
2 6
3 6
3 5
4 5
4 6
5 6
5 4
7 4
7 7
0 7
```

the correct result is:

2

# / Solution

I learned about this interesting task by Wojciech Guzicki back in 1998 when I was preparing its model solution and test data for the Polish Olympiad in Informatics. This seemingly typical problem in computational geometry suggests a pretty mundane solution using a standard technique—a sweep line. It turns out, however, that a much easier solution exists, which skips the complexities of advanced data structures suggested by the canonical approach.

**Solution**

Imagine that the polygon occurs in the real world and that it is large—as large as the mysterious Nazca Lines—and the window is as large as a soccer field. Due to these enormous dimensions, we cannot keep the whole image in sight (except if looking from an airplane), but we are still able to count the fragments by following the boundaries of the polygon and the window.

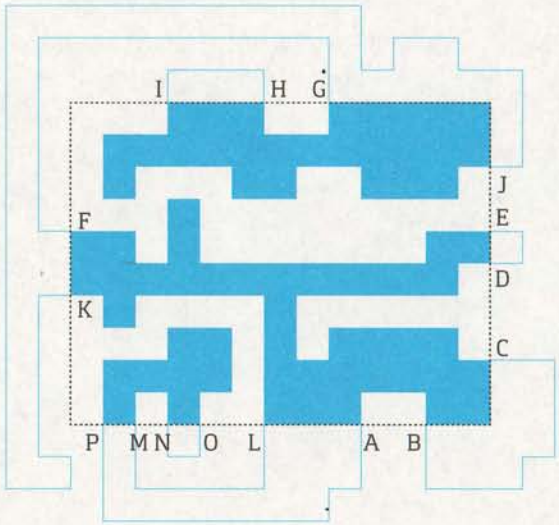



***Figure 1:*** *A polygon and a window. Three fragments are visible.*



If the boundary of the polygon intersects the boundary of the window, then by alternating between the consecutive boundaries of the polygon and the window we can count the fragments. Each polygon fragment consists of the fragments of window and polygon boundaries (see figure 1). In order to count the polygon fragments we shall traverse each fragment counterclockwise along its border.

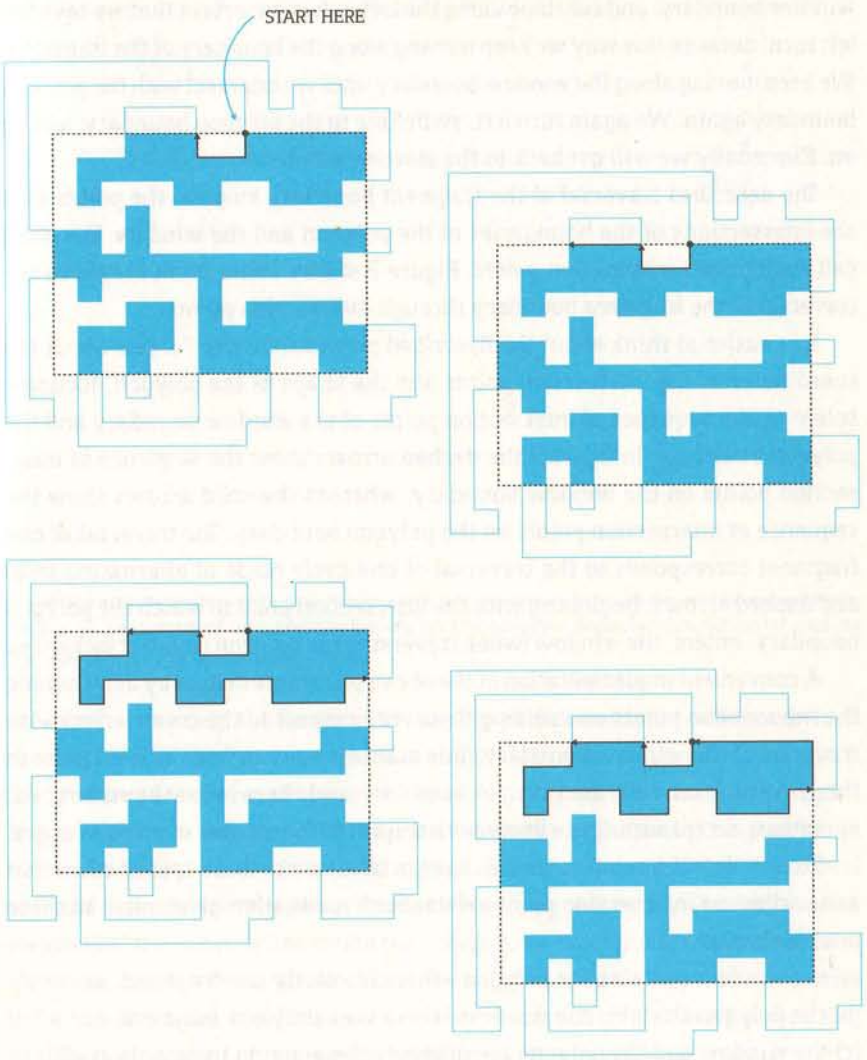



***Figure 2:*** *Subsequent steps of fragment traversal.*

We begin with an arbitrary polygon vertex external to the window and we follow the polygon boundary counterclockwise, up to the intersection with the window boundary (it is an intersection point where the polygon edge "enters" the window). This is when the traversal of one visible fragment begins. We continue to move along the polygon boundary until we find the spot where the polygon boundary crosses the window boundary and leaves the interior of the window. At this intersection point we turn left, switch from the polygon boundary to the window boundary, and continue along the latter. It is important that we take the left turn, because this way we keep moving along the boundary of the fragment. We keep moving along the window boundary until we intersect with the polygon boundary again. We again turn left, switching to the polygon boundary, and so on. Eventually we will get back to the starting point.

The described traversal of the fragment boundary stops at the points that are intersections of the boundaries of the polygon and the window. We shall call such points *intersection points*. Figure 2 shows subsequent stages in the traversal of the fragment boundary through intersection points.

It is easier to think about the described procedure if one forgets about the coordinates of the intersection points and the shape of the polygon, focusing solely on the sequence of intersection points of the window boundary and the polygon boundary. In figure 3 the dashed arrows show the sequence of intersection points on the window boundary, whereas the solid arrows show the sequence of intersection points on the polygon boundary. The traversal of one fragment corresponds to the traversal of one cycle made of alternating solid and dashed arrows, beginning with the intersection point in which the polygon boundary "enters" the window (when traversing the polygon counterclockwise).

A convenient implementation of the above procedure begins by determining the intersection points and sorting them with respect to the counterclockwise traversal of the window boundary. This makes it easy to find the next point in the traversal, on both the polygon boundary and the window boundary. All operations except sorting are linear with respect to the number of polygon edges.

To complete the solution we still have to take care of three special cases that arise when no intersection point exists. Such a situation gives raise to three possible cases:

(a) the window contains the polygon—there is exactly one fragment, or
(b) the polygon contains the window—there is exactly one fragment, or
(c) the window and the polygon are disjoint—there are no fragments at all.



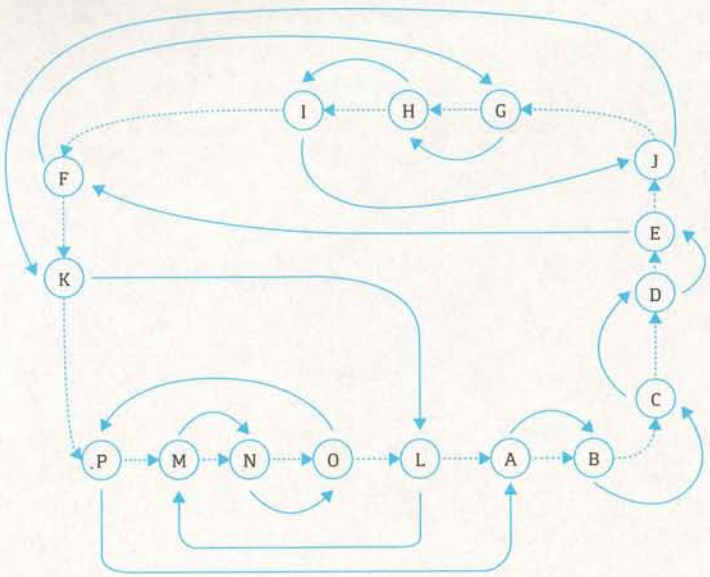



***Figure 3:*** *Sequence of intersection points on the polygon boundary (solid lines) and on the window boundary (dashed lines).*

If any vertex (e.g. the first one) of the polygon lies inside the window (which is easy to determine), then we are in case (a). Otherwise, we need to check whether any window vertex (e.g. the lower left corner) lies inside the polygon. This is a bit more complex. We draw an arbitrary ray (half-line) originating in the said window vertex (it is most convenient to use a vertical or horizontal ray) and count how many times it stabs the polygon boundary. An odd number means that the window lies inside the polygon, so exactly one fragment exists. However, cases when a whole side of the polygon is contained in the ray need to be handled more carefully; we can also avoid such situations by choosing as the starting point of the ray any window point of non-integer coordinates.

# / Altars

**Contest:** 6th Polish Olympiad in Informatics
**Task author:** Grzegorz Jakacki
**Memory:** 32 MB
http://main.edu.pl/en/archive/oi/6/olt

According to Chinese folklore, evil spirits can only move in a straight line. This is of great importance when temples are built. The temples are constructed on rectangular planes with sides parallel to the north-south or east-west directions. No two rectangles have common points. An entrance is situated in the middle of one of four walls, and its width is equal to half of the length of the wall. An altar appears in the center of the temple, where the diagonals of the rectangle intersect. If an evil spirit appears at this point, the temple will be profaned. This may happen only if there exists a ray that runs from an altar, through an entrance, to infinity and neither intersects nor touches the walls of any temple (on a plane parallel to the plane of a construction area); i.e. if one can draw, in a construction area, a line that starts at the altar and runs to infinity without touching any wall.

**Task**

Write a program that:

- reads descriptions of the temples from the input,
- verifies which temples could be profaned,
- writes their numbers to the output.

**Input**

In the first line of the input appears one integer $n$, the number of temples ($1 \le n \le 1000$).

In each of the following $n$ lines there is a description of one temple (the $i$-th line contains a description of the $i$-th temple). The description of a temple consists of four nonnegative integers, not greater than 8000, and a letter: E, W, S or N. The first two numbers are the coordinates of the temple's northwestern corner, and the next two numbers are the coordinates of the opposite southeastern corner. In order to specify the coordinates of a point, first we give its geographical longitude, which increases from the west to the east, and then its latitude, which increases from the south to the north. The fifth element of the description indicates the wall with the entrance (E = eastern, W = western, S = southern, N = northern). The elements of the temple's description are separated by single spaces.

**Output**

In the consecutive lines of its output, your program should write, in ascending order, the numbers of any temples that may be profaned by an evil spirit. Each number is placed on a separate line. If there are no such temples, you should write one word: BRAK (Polish for *lack*).

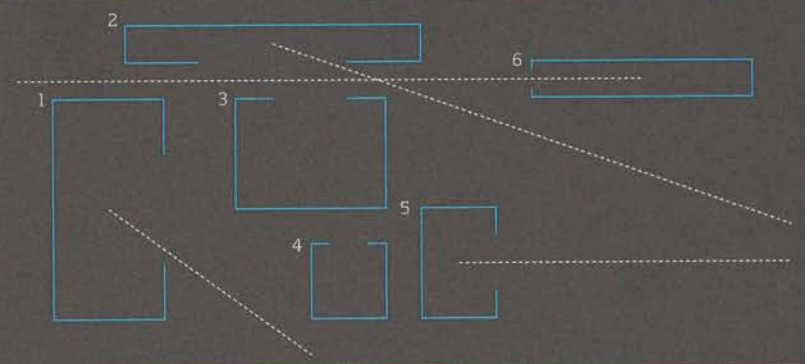
**Example**

For the input data:

```
6
1  7  4  1  E
3  9  11  8  S
6  7  10  4  N
8  3  10  1  N
11  4  13  1  E
14  8  20  7  W
```

the correct result is:

```
1
2
5
6
```




**Explanation of the example:** The picture shows the temples described in the sample input. The dashed lines show the possible routes of evil spirits.

# / Solution

In southeastern China, near the border with Burma, Laos and Vietnam, lies the Yunnan province, famous for its exquisite tea plantations. In the very heart of Yunnan lies Kunming, the City of Eternal Spring, as it is called for its thriving vegetation and mild climate. On the outskirts of Kunming sits the Purple Bamboo Temple (Qiongzhu Si), where the idea for the task *Altars* popped into my head.

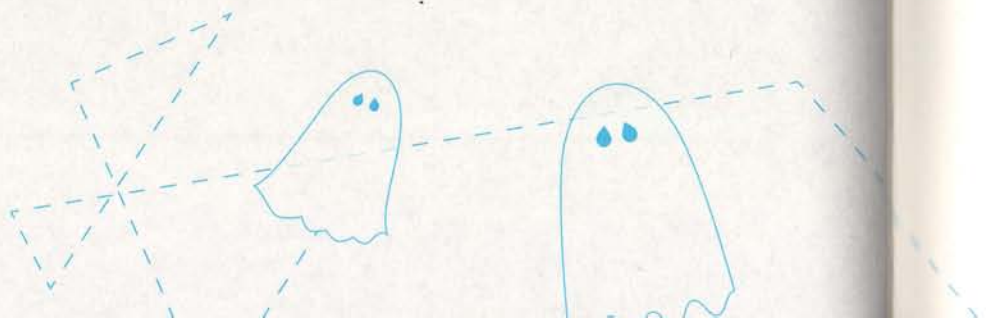
The Chinese believe that *influences* (bad influences in particular) propagate only along straight lines. *Spirit Walls* are traditionally built in front of entrances to the temples to protect the insides by blocking straight-line entry. For similar reasons, many houses are accessible only through winding passages.

One may imagine, if standing at the temple's altar, i.e. in the center of the rectangle circumscribed by the temple's walls, that the specifications of the temple may allow it to become desecrated if, in one's field of vision, there is a gap revealing the outside world, not just the other surrounding temples. The model solution is based on this analogy.

**The first solution**

Imagine a ray (half-line), pinned in the altar, that scans the field of vision like an airport radar. The scan begins at one door frame and ends at the other. Observing the count of walls that intersect the scanning ray, we notice that moving the ray onto a new wall increases the count by one and leaving the wall behind decreases the count by one. A fragment of the field of vision in which the count is equal to zero is a gap to the outside world.

The ray scanning and wall counting can be implemented by sorting all walls' vertices on their angular coordinates in the polar coordinate system originating at the altar point of the temple in question. Sorting must be implemented in such a way as to preserve the correspondence between both endpoints of the wall.





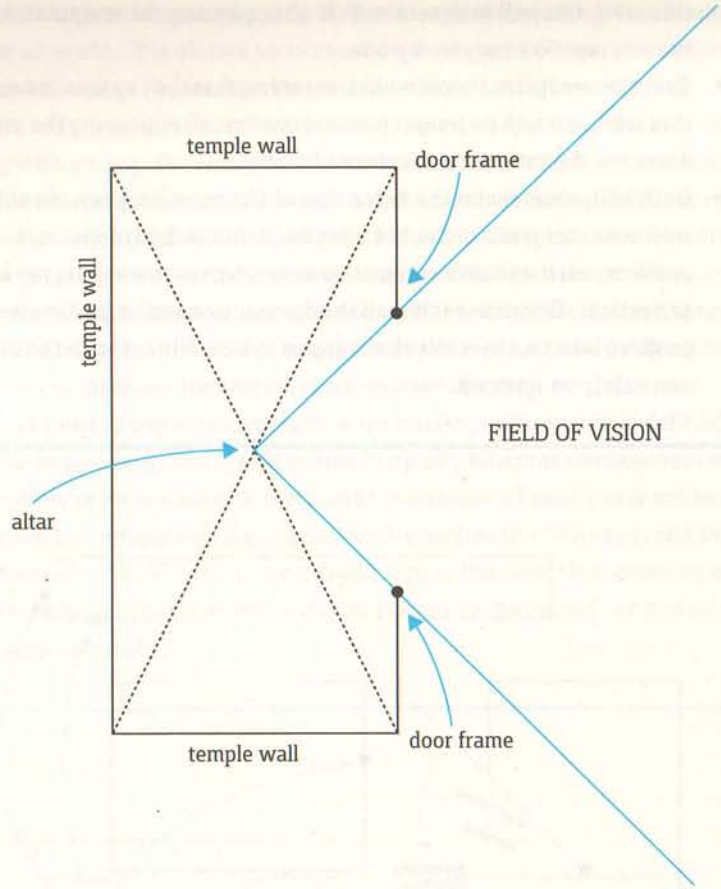



***Figure 1:*** *Layout of a temple.*

Imagine that we rotate the scanning ray such that it is always pinned to the altar point of the selected temple, and such that it stops at the endpoint of each wall (with the proper order of the wall endpoints being determined by the angular sorting). Each position of the rotating ray divides the plane into two open semiplanes*: "downwind", i.e. the semiplane being left behind as a result of the rotation, and "upwind", i.e. the one which is being entered by the rotating ray. Visiting the sorted wall endpoints, we face one of three cases:

* An open semiplane is a semiplane excluding its boundaries.

- The other endpoint is upwind, meaning that the ray did not intersect with this wall, but will intersect with it after passing the endpoint. We increase the intersection counter by one.
- The other endpoint is downwind, meaning that the ray was intersecting with this wall, but will no longer intersect with it after passing the endpoint. We decrease the intersection counter by one.
- Both endpoints lie on the extension of the sweeping ray, meaning that the wall obscures just a point, not a segment, in the field of vision. In the original problem, such a situation can arise only when the sweeping ray is horizontal or vertical. Because each wall is adjacent to another perpendicular wall of positive length, the walls that happen to be collinear with the sweeping ray can safely be ignored.

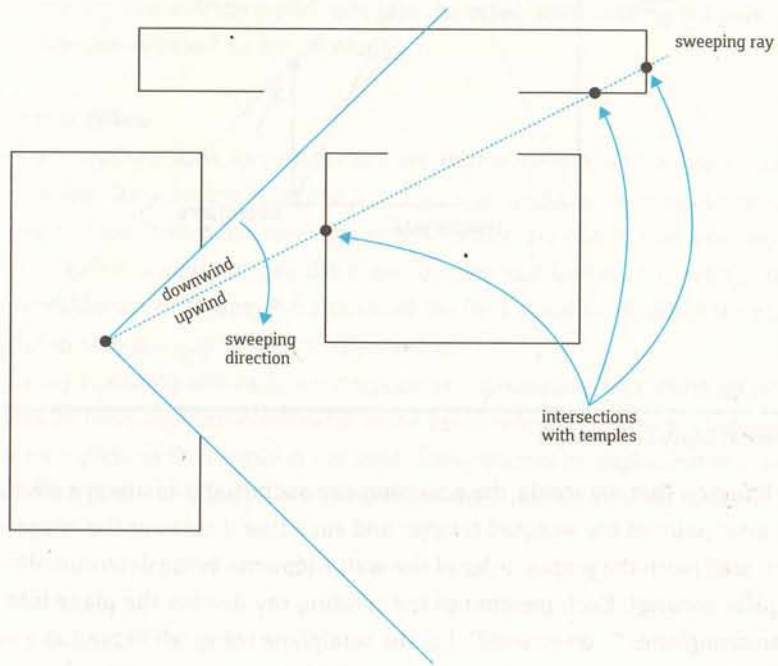



***Figure 2:*** *The sweeping ray and its intersections with the temples' walls.*



Three issues are still outstanding. First, we need to start properly. In order to do that, we can set the sweeping ray to go through the starting door frame and traverse all walls. This allows us to count how many walls are intersected by the ray and initialize the intersection counter. But there's a simpler way.

The initial set of walls (being just line segments) can be transformed by intersecting with an angular section of the plane determined by the angle with a vertex at the altar and rays going through door frames (see figure 3). This way, we ignore irrelevant segments outside the field of vision and also the irrelevant fragments of segments that are visible only partially. This operation contributes to a reduction of the sorting time, and we also guarantee that the sweeping ray anchored at the door frame does not intersect any of the (transformed) walls. This enables us to initialize the intersection counter to zero.

Second, we need to pay attention to how the sorting relationship is defined. Logically, the sequence of the swept points is cyclic, whereas employment of angular coordinates (or any other projection of points into a linearly ordered set) causes the need for "wrapping". E.g. a point with coordinate 1° may succeed the point with coordinate 359°, which is contradictory to the canonical ordering of numbers. An easy solution is to include door frames in the sorted set and rely on them as loop sentinels.

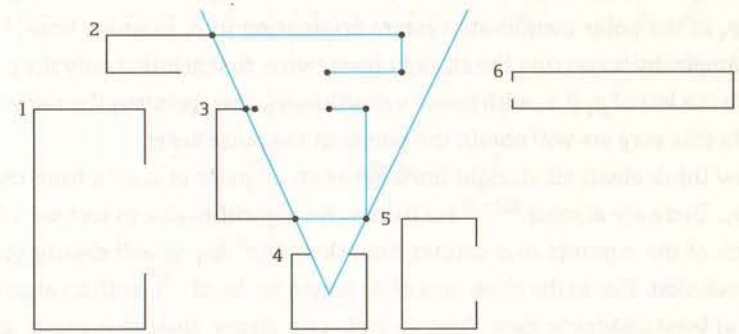



***Figure 3:*** *Set of walls restricted to the intersection with the field of vision.*

Third, we have to deal with angular coordinates carefully. Translating Cartesian coordinates (as given in the input data) into polar coordinates involves a danger of rounding errors, due to the use of floating-point arithmetic. Luckily the computation of the sign of appropriate vector products can entirely replace the need for trigonometry.

The algorithm shown above determines whether one particular temple can be desecrated. In order to determine this property for all the temples, one needs to repeat this procedure $n$ times. The algorithm's dominant operation for one temple is sorting (running time $O(n \log n)$), thus the total running time is $O(n^2 \log n)$. Such an approach seems extravagant, because substantial amounts of computation are performed for each temple, but their results are not used when processing the following temples. This can be remedied.

**How to perform angular sorting**

The algorithm's main operation is the angular sorting of the set of points with respect to one of them. There exists an algorithm that performs this step in $O(n^2)$ total time. It is interesting and difficult, and we will summarize it briefly.

Think about the set of points $p_1, \ldots, p_n$. Let us choose one of them. It does not matter which one, so let us take $p_1$. Run a straight line through $p_1$ so that it gets connected with all other points $p_2, \ldots, p_n$ (see figure 4). Now, for each of those remaining points, there is one straight line. If we know how to sort these straight lines with respect to their slopes (i.e. the $a$ in the equation $y = ax + b$ describing the line), we would be able to infer the angular order of the points $p_2, \ldots, p_n$ in the polar coordinate system originating in $p_1$ in linear time. How? For example, by traversing the straight lines twice, first printing only the points lying to the left of $p_1$ (i.e. with lower $x$ coordinate), then printing the remaining ones. In this way we will obtain the points in the polar order.

Now think about all straight lines between all pairs of points from the set $p_1, \ldots, p_n$. There are at most $\frac{n(n-1)}{2}$ such lines. An algorithm able to sort $n-1$ lines for each of the $n$ points in a shorter time than $\Theta(n^2 \log n)$ will clearly yield a better solution. Due to the sheer size of its output (order of $n^2$), such an algorithm needs at least quadratic time. And, in fact, a quadratic algorithm exists and is based on an extremely interesting concept.



**Dualization.** Imagine a geometric transformation $T$, which maps a point $p$ into a straight line $T(p)$, and a straight line $l$ into a point $T(l)$ as follows:

$$p = (a, b) \rightarrow T(p) : y = ax - b$$
$$l : y = kx + d \rightarrow T(l) = (k, -d).$$

Thus a point $(a, b)$ is mapped into a straight line $y = ax - b$. The straight line $y = kx + d$ is mapped into a point $(k, -d)$. Both domain and co-domain of the above transformation are the two-dimensional plane $\mathbb{R}^2$. We shall call the domain a primary space and the co-domain a dual space. The transformation just described has an interesting property. A point $p$ lies on the straight line $l$ if, and only if, the straight line $T(p)$ intersects the point $T(l)$.

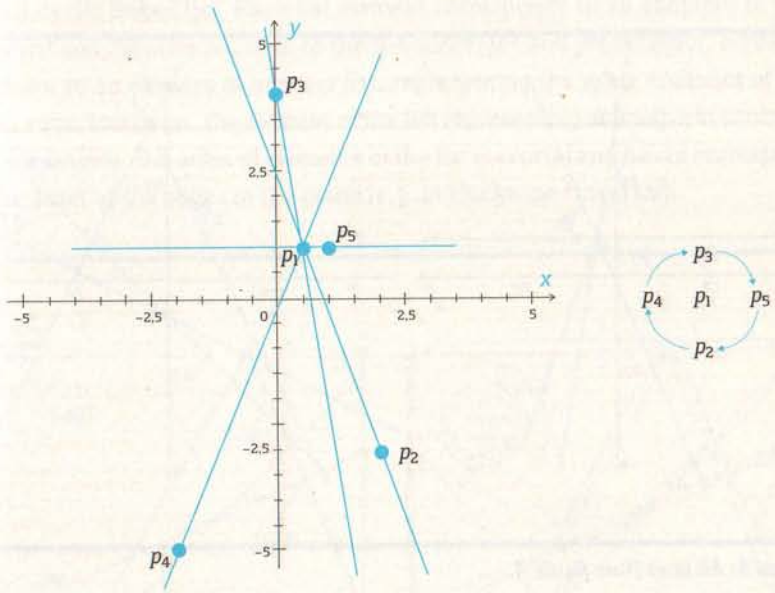



*Figure 4: Straight lines from $p_1$ and the angular order of points with respect to $p_1$.*

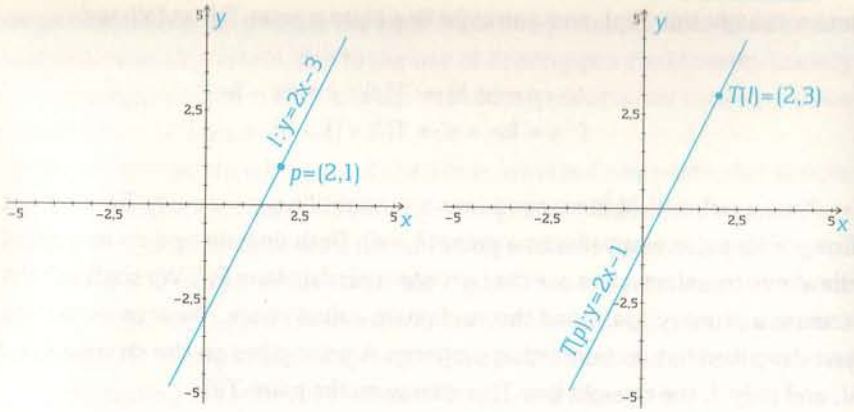



***Figure 5:*** *Primary space (on the left) and its corresponding dual space (on the right).*

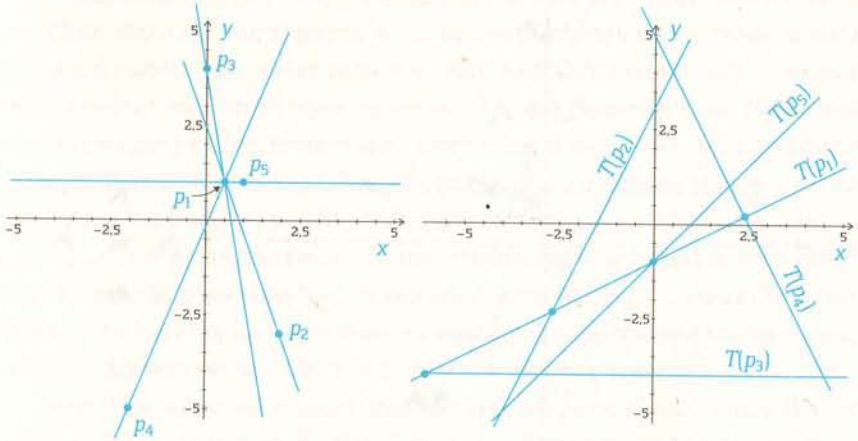



***Figure 6:*** *All lines from figure 4.*

The set of points $p_1,\ldots,p_n$ and the set of straight lines running through the pairs of these points can be mapped into the set of straight lines $T(p_1),\ldots,T(p_n)$. In the dual space, the straight lines connecting $p_1$ with other points correspond to the intersections of the straight line $T(p_1)$ with the straight lines $T(p_2),\ldots,T(p_n)$. However, there is a small exception. If some point $p_i$ lies over or under $p_1$, in the dual space, the lines $T(p_1)$ and $T(p_i)$ are parallel. We will handle this in a moment.



We have arrived at the following problem: $n$ straight lines in the plane are given. For each of them, find the order of the intersection with the remaining straight lines. It follows from the definition of the dual space that no straight line will be vertical. From the ordering of the intersections of the line $T(p)$ in the dual space we can recreate the ordering of the straight lines intersecting the point $p$ in the primary space and, further, the angular ordering of the remaining points with respect to $p$. The same procedure will work for any point $p_1,\ldots,p_n$. Should there be any points above and under $p_1$, they must be considered separately. After sorting all other points in the angular order with respect to $p_1$, we need to insert the points lying above or under $p_1$ into this ordering. This can easily be done in linear time. From now on we assume that no such points exist.

**Finding line intersections.** Let us take a look at figure 7, showing the data structure representing the set of endpoint-connected segments. To be precise, this is a set of segments such that the segments intersect only at endpoints, and each endpoint belongs to at least two segments. Each endpoint is represented with a cyclic linked list. Each list element corresponds to an endpoint of one segment and, besides pointers to the list successor and predecessor, contains a pointer to an element of another list, representing the other endpoint of the same edge. Moreover, the element of the list representing an endpoint contains its coordinates. The order of elements in the list is crucial and has to correspond to the order of the edges in the plane (e.g. in clockwise traversal).

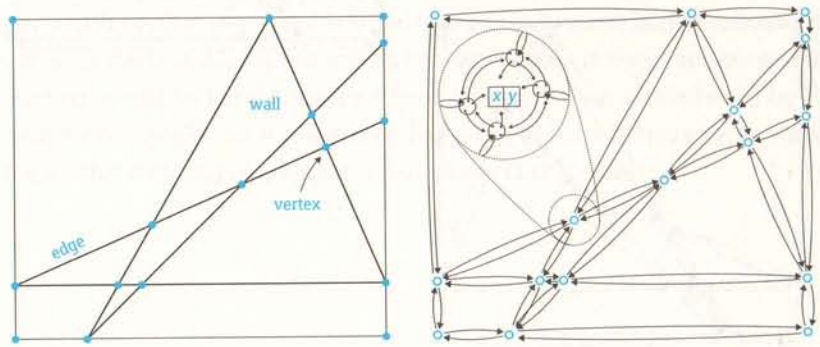



***Figure 7:*** *Data structure (on the right) representing a set of segments (on the left). The set of segments was built by drawing a rectangular frame around the lines from figure 6.*

This data structure is a variant of a so-called *doubly-connected edge list*. It enables the quick insertion of new edges into an existing set. Notice also that, thanks to the pointers to adjacent edges and to the proper ordering of the elements in the cyclic linked lists corresponding to endpoints, one can easily traverse the boundaries of faces made up of the edges. Figure 8 shows several steps in the process of adding line segments to the set.

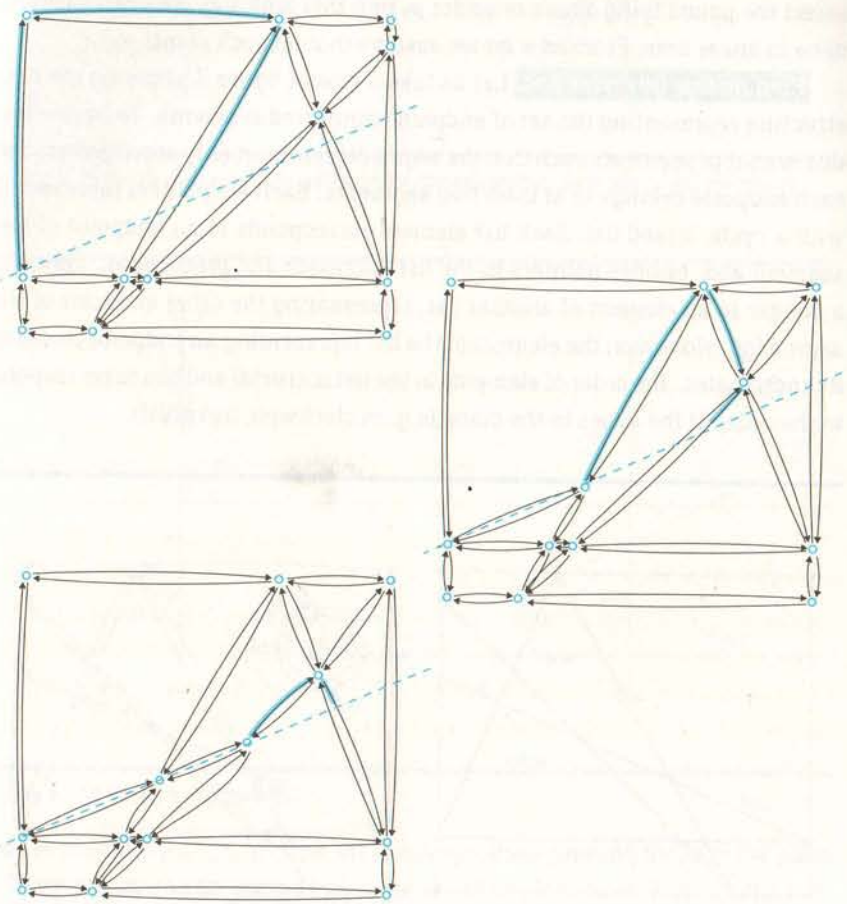


***Figure 8:*** *Adding a new line to the data structure.*

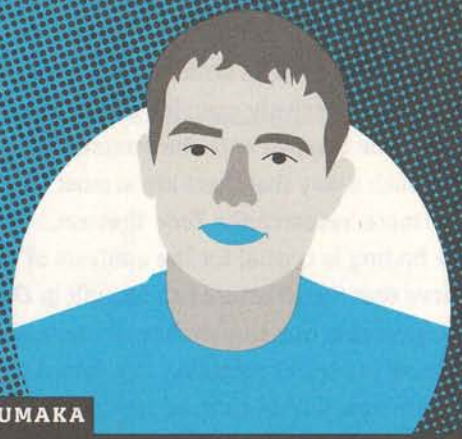


The operations on this data structure seem quite straightforward. We are interested in how many of them may need to be performed (with respect to $n$), because this is crucial to the time complexity of this phase of the algorithm. It is clear that individual new faces may have many edges (the order of $n$). It turns out, however, that the *total* number of edges that need to be visited in order to add one straight line to the data structure is also of the order of $n$. We omit the proof, but it is not excessively complex and is based on the observation that visited edges can be mapped onto the intersection points of the straight line being added in such a way that there are at most 10 edges per one intersection point. To learn more, research the *Zone Theorem*.

The above finding is crucial for the analysis of the algorithm's time complexity. Observe that the structure can be built in $O(n^2)$ time. Traversal of the subsequent segments of one straight line can be realized in $O(n)$ time. Observe that if we dualize the set of points $p_1,\ldots,p_n$ into the set of lines $T(p_1),\ldots,T(p_n)$ and contain these lines within an envelope*, we will arrive at a set of segments similar to the one in figure 7. The algorithm for building the doubly-connected edge list described above can determine the intersection points for all lines in $O(n^2)$ total time. Now, if we wish to determine the angular order of all the points $p_1,\ldots,p_n$ with respect to one chosen point (e.g. $p_1$), we can read the ordering of the intersection of the straight line $T(p_1)$ with the remaining straight lines. The indices of the remaining straight lines will determine the ordering of the points $p_2,\ldots,p_n$ in the angular order with respect to $p_1$, which has already been discussed above.

The wholesale angular sorting of the set of $n$ points speeds up our initial algorithm from $O(n^2 \log n)$ to $O(n^2)$. This speed-up does not matter much in the solution of the task *Altars*, because its implementation is very complex and the speed-up would be hardly observable on the input data sets allowed by the problem description. Such a speed-up has merit in more serious applications, such as the rendering of graphics or integrated circuit designs.

* A rectangular envelope containing all intersection points of $n$ straight lines can be determined as follows. Sort (in $O(n \log n)$) all lines with respect to their slopes and consider only the intersection points of the lines that have adjacent slopes (including the pair consisting of the first and last lines). All other intersection points lie inside the convex hull circumscribed around these.



**TOMASZ KOCIUMAKA**

Tomasz Kociumaka is a student of a double degree program in Computer Science and Mathematics at the Faculty of Mathematics, Informatics and Mechanics, University of Warsaw. He started to take part in algorithmic competitions during his sophomore year at high school. A year later he won the Polish Olympiad in Informatics and earned a gold medal at the International Olympiad in Informatics. During his university years, his team was a runner-up in the Polish Collegiate Programming Contest. He has been successful in mathematical competitions as well: he earned a silver medal at the International Mathematical Olympiad, and later on celebrated—twice—a first place at its counterpart for university students, the International Mathematical Competition. Recently, he has been engaged in research into text algorithms. His greatest achievement in this area is designing a linear-time algorithm for finding seeds of a string. This problem was used at Algorithmic Engagements 2009 as the task *Quasi-template*. Tomasz Kociumaka also works as a jury member of the Polish Olympiad in Informatics, Central European Olympiad in Informatics (in 2011) and the Polish Olympiad in Informatics Training Camp. He also organizes yearly algorithmic workshops of the Polish Children's Fund. As a high school student, he was a participant in the Fund's program. He likes spending his free time hiking in the mountains.

# / Axes *of Symmetry*

**Contest:** 14th Polish Olympiad in Informatics
**Task author:** Piotr Stańczyk
**Memory:** 32 MB
http://main.edu.pl/en/archive/oi/14/osi

Little Johnny—a well-respected young mathematician—has a younger sister, Justina. Johnny likes his sister very much and he gladly helps her with her homework, but, like most scientific minds, he does object to solving the same problems repeatedly. Unfortunately, Justina is a very diligent pupil, and so she asks Johnny to review her assignments many times, for the sake of certainty. One sunny Friday, just before the famous Long May Weekend*, the math teacher set many exercises consisting of finding the axes of symmetry of various geometric figures. Justina is most likely to spend a considerable amount of time solving these tasks. Little Johnny had arranged himself a trip to the seaside long before; nevertheless, he feels obliged to help his little sister. Soon, he has found a solution—it would be best to write a program that would ease the checking of Justina's solutions. Since Johnny is a mathematician rather than a computer scientist, and you are his best friend, it falls to you to write it.

## Task

Write a program that:

- reads the descriptions of the polygons from the input,
- determines the number of axes of symmetry for each one of them,
- writes the result to the output.

## Input

In the first line of the input there is one integer $t$ ($1 \le t \le 10$), specifying the number of polygons for which the number of axes of symmetry is to be determined. Next, $t$ descriptions of the polygons follow. The first line of each description contains one integer $n$ ($3 \le n \le 100{,}000$) specifying the number of

* In Poland, there is an accumulation of public and national holidays at the beginning of May. They are: May 1—May Day, the International Workers' Day; May 2—the Day of the Polish Flag; May 3—the Anniversary of the Establishment of the First Constitution of Poland (dating back to the year 1795). Though the middle, quite recent, feast is not actually a holiday, it is, however, customary to make it a day off from work too. Now imagine that May 1 is on a Monday, or May 3 on a Friday, and you will get a feeling for what the Long May Weekend is like!

vertices of the polygon. In each of the following $n$ lines there are two integers, $x$ and $y$ ($-100{,}000{,}000 \le x, y \le 100{,}000{,}000$), representing the coordinates of subsequent vertices of the polygon. The polygons need not be convex, but they have no self-intersections; any two sides have at most one common point: their common endpoint, if they actually share it. Furthermore, no pair of consecutive sides is parallel.

**Output**

Your program should output exactly $t$ lines, with the $k$-th line containing a single integer—the number of axes of symmetry of the $k$-th polygon.

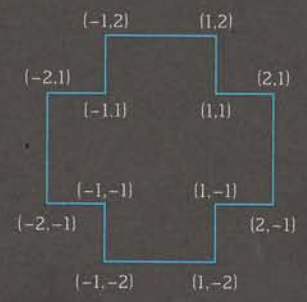
**Example**

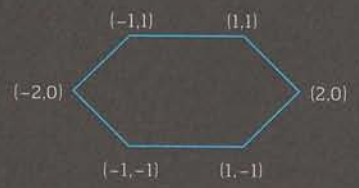
For the input data:

```
2
12
1  -1
2  -1
2  1
1  1
1  2
-1  2
-1  1
-2  1
-2  -1
-1  -1
-1  -2
1  -2
6
-1  1
-2  0
-1  -1
1  -1
2  0
1  1
```

the correct result is:

```
4
2
```






# / Solution

The vast majority of algorithmic problems can be classified according to the area they concern. As such, we may easily distinguish graph problems, text problems, geometry problems, number theory problems, game theory problems etc. Usually, a problem is quickly classified in this way—generally as early as during the initial reading of the task statement. Very often, the solution uses techniques and algorithms specific to the problem's area, sometimes with the aid of data structures that are fundamental to the whole area of algorithms.

Far fewer tasks have solutions that are based on reductions to problems in different areas. Such solutions tend to be simultaneously tricky and easy to code. As we will see in a moment, this is the case in *Axes of Symmetry*, which is definitely a geometrical problem but which turns out to have a solution that utilizes text algorithms.

**Problem analysis—a load of geometry**

Let us find out when a line is a polygon's axis of symmetry. First, notice that such a line must cross the perimeter of the polygon. This is intuitively clear—if the line does not cross the border, the whole polygon needs to lie on the same side of the line*, which is impossible for an axis of symmetry. Hence let us consider only lines crossing the polygon's border.

Let us conduct a thought experiment. Imagine we have two robots, which are represented as points on the plane. Assume that they can follow polygonal paths with a constant velocity, which is common to both of them. Let us put the robots at a single starting position at a crossing point of the line and the polygon's perimeter. Now, make them move along the perimeter in opposite directions until they reach the starting point again. The polygon has no self-intersections, so their movement is well defined. What is more, both robots finish at the same time and they pass each other exactly once, in the middle of their way.

Robots follow the whole perimeter, so if their positions remain symmetric with respect to the line, then this line is an axis of symmetry of the perimeter, and consequently of the whole polygon. Clearly, the opposite implication is also true (see figure 1). Note that the symmetry axis crosses the perimeter at exactly two points: the starting point and the point where the robots passed each other.

* We do not provide rigorous proofs of intuitively clear geometrical statements like this. In order to do so, we would need to build a formal setting with appropriate definitions and axioms. This way, descriptions would be harder to read and key ideas would not be made easier to understand.

How can we benefit from this discussion to check whether a given line is an axis of symmetry? We know that it is enough to find out whether the robots' positions stay symmetric with respect to the line. In order to invent a way of doing so, imagine that the robots cannot follow the perimeter on their own, but need us to give them a sequence of commands. Assume that each robot is oriented; that is, it has a "forward" direction. The commands are of two types:

- rotate by an angle of $180° - \alpha$ (denoted as $\alpha$),
- move forward by $d$ units (denoted as $d$).

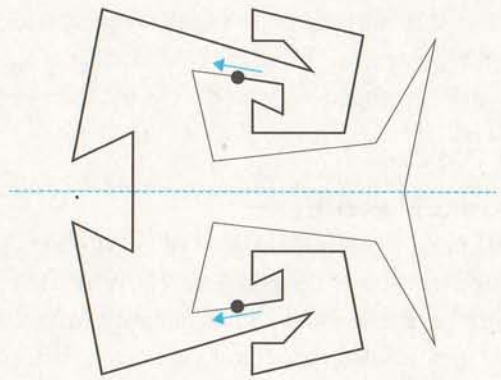


***Figure 1:*** *If a line is an axis of symmetry, the robots' positions are always symmetric with respect to that line.*

Additionally, assume that the robot that goes around the polygon clockwise rotates clockwise, and the other one counterclockwise. Moreover, at the starting point, let the robots be oriented in opposite directions perpendicular to the line under investigation. The sequence of commands we give, if necessary, begins with an appropriate rotation to the starting directions. Then, moving forward to the next vertex alternates with the rotation at that vertex.

If both robots are given the same sequence of commands, their positions remain symmetrical with respect to the line. What is more, if we never ask them to go forward by 0 units or rotate by 0°, the opposite implication is also true. The commands we give using the polygon's description to make the robots follow the perimeter satisfy this condition; that is because, as noted in the problem statement, no two consecutive sides are parallel.



The observations we have already made allow us to limit the set of lines that are worth considering as candidates for axes of symmetry. First of all, the initial rotations must be equal for both robots. Hence, the starting point may either be a vertex, with the line being the bisector of the corresponding angle, or a point of an edge, with the line being perpendicular to that edge. In the latter case, the point must be the edge's midpoint and the line the bisector of the edge, since the first movements forward must be equal.

In this way we have limited the set of candidates to $2n$ lines: $n$ for either case. Note that actual axes of symmetry (and possibly some other lines) are counted twice here: once for each crossing point with the perimeter.

The only thing we need to do for each candidate is to compare the sequence of commands. Note that we can omit the initial ones, which we used to prove that the axis of symmetry is a bisector of an angle or of an edge. This way, we just compare sequences consisting alternately of edge lengths and angle values (see figure 2).

### The solution—a shift to text algorithms

How can we compare such sequences of commands? If we proceed in a naïve way, we obtain an $O(n^2)$-time algorithm. However, the sequences we are comparing are very regular, and this makes it possible to design a much faster algorithm.

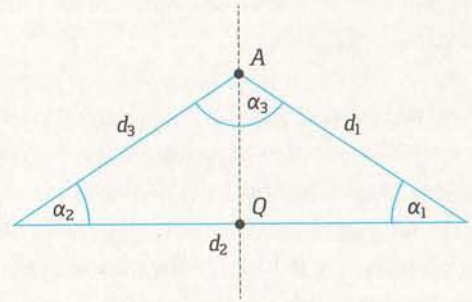



***Figure 2:*** *When A is the starting point we need to compare* $(d_1,\alpha_1,d_2,\alpha_2,d_3)$ *with* $(d_3,\alpha_2,d_2,\alpha_1,d_1)$. *If it is Q, the sequences are* $(\alpha_2,d_3,\alpha_3,d_1,\alpha_1)$ *and* $(\alpha_1,d_1,\alpha_3,d_3,\alpha_2)$.

Consider a cyclic sequence of consecutive edge lengths alternating with the values of angles between them:

$$(d_1, \alpha_1, d_2, \alpha_2, \ldots, d_n, \alpha_n).$$

Each check we make can now be described as starting from some position $i$ in this sequence and, for each $j = 1, 2, \ldots, 2n-1$, comparing the values at positions $i+j$ and $i-j$. The candidate is an axis of symmetry if and only if, for a corresponding $i$ and for each $j$, the values are equal. Hence, the final result is half of the number of such positions in the cyclic sequence.

In other words, we want to find out for which $i$ a contiguous subsequence of length $4n-1$, with its midpoint in $i$, is a *palindrome*—i.e. the same whether read backwards or forwards. We can solve the problem for the standard linear (i.e. not cyclic) string by unwinding the cycle three times and considering just palindromes with midpoints among the middle $2n$ positions. For example, for point $Q$ of figure 2, we check whether, in the sequence below, the first underlined word read forwards is equal to the second one read backwards:

$$d_1, \alpha_1, d_2, \underline{\alpha_2, d_3, \alpha_3, d_1, \alpha_1}, d_2, \underline{\alpha_2, d_3, \alpha_3, d_1, \alpha_1}, d_2, \alpha_2, d_3, \alpha_3.$$

Again, the final answer is half of the number of palindromes of length $4n-1$ centered at such positions.

In this way, we have reduced the geometrical problem of finding the axes of symmetry to a text problem: in a word of length $6n$, find the midpoints of the palindromes of length $4n-1$.

In order to solve this task, as well as most of the problems related to palindromes, we may use the classical text algorithm named after Glenn Manacher. Given a word, for each position this algorithm finds the maximum length of the palindrome centered at this position. For example, for *abaaaba* it outputs a sequence 1, 3, 1, 7, 1, 3, 1. Recall that Manacher algorithm works in linear time.

In our problem we just need to identify the positions for which the values computed by Manacher algorithm are at least $4n-1$, since whenever a position is the center of an odd palindrome of given length, it is the center of the palindromes of all smaller odd lengths.



# / Leonardo *Numbers*

**Contest:** Polish Olympiad in Informatics Training Camp 2008
**Task author:** Tomasz Kulczyński
**Memory:** 32 MB
http://main.edu.pl/en/archive/ontak/2008/leo

The famous Fibonacci numbers are not the only discovery of Leonardo of Pisa (also known as Fibonacci); he is also known for a similar sequence called the Leonardo numbers. Let us define the sequence of Leonardo numbers $(L_i)$ as $L_0 = L_1 = 1$ and $L_{i+1} = L_i + L_{i-1} + 1$ for $i \geq 1$. Today, 800 years after Leonardo Fibonacci's death, we would like to find the value of the following expression:

$$L_0^k + L_1^k + L_2^k + \cdots + L_n^k.$$

**Task**

Write a program that:

- reads from the input the integers $n$ and $k$,
- evaluates the expression,
- writes the last nine digits of the result to the output.

**Input**

The first and only line of input contains two positive integers, $n$ and $k$ ($1 \leq k \leq 13$, $n$ fits in a 64-bit unsigned integer type).

**Output**

The only line of output should contain the last nine digits of the decimal representation of the result of the expression.

**Example**

For the input data:

3 2

the correct result is:

000000036

**Explanation of the example:** The expression corresponding to this example is $1^2 + 1^2 + 3^2 + 5^2 = 36$.

# / Solution

This task appeared at the Polish Olympiad in Informatics Training Camp in 2008. I was a participant at this camp, but at the time I had only been interested in algorithmics for about a year, and the problem was too hard for me. When I learned the solution, I found its techniques very interesting. I started analysing those techniques, and by combining them with some observations I had made during the contest, I managed to create a solution that was not only faster but also easier to code than the original. Moreover, when I came back to the problem in order to write this text more than three years later, I was able to design an even better algorithm. The character of number-theory problems like this makes me believe that an even faster algorithm might be devised; possibly even a not very complicated one.

**Initial remarks on residues.** We are to compute only the last nine digits of the result, i.e. its residue modulo $M = 10^9$. The formula for the sum of powers of Leonardo numbers involves only addition, multiplication and exponentiation. Those operations—and hence also determining the value of Leonardo numbers—can be done on residues modulo $M$. Addition and multiplication of such residues can easily be implemented using 64-bit integer variables.

In the following text, we usually operate on residues modulo $M$. For brevity, we will use standard notations for operations both on integers and on residues. In particular, $L_n$ often denotes not the Leonardo number itself, but its remainder in division by $M$. In terms of abstract algebra, we perform most of the calculations in the $\mathbb{Z}_M$ ring.

### How to compute the Leonardo numbers

In this task we need to deal with Leonardo numbers that are really far along the sequence. The only assumption we make about $n$ is that it fits into a 64-bit unsigned integer variable, i.e. $n < 2^{64}$. This makes even determining the value of $L_n$ a nontrivial problem. The simple algorithm, using the recursive formula and written so that it doesn't compute the same number multiple times, has $\Theta(n)$ time complexity. That is far from acceptable. As a warm-up, let us significantly improve this running time.

**Fibonacci numbers.** We begin by solving the same problem for the famous relative of the Leonardo numbers—the Fibonacci numbers, defined recursively as follows:

$$F_0 = 0, \quad F_1 = 1, \quad F_{n+1} = F_n + F_{n-1}.$$



The Fibonacci numbers are broadly known and well studied. In particular, in many books and other sources one can find the closed-form formula for the $n$-th Fibonacci number:

$$F_n = \frac{\varphi^n - \psi^n}{\sqrt{5}},$$

where $\varphi = \frac{1+\sqrt{5}}{2}$ and $\psi = \frac{1-\sqrt{5}}{2}$. Unfortunately, $\varphi$ and $\psi$ are not integers (not even rational numbers). Although this is not a problem when we are to find the approximate values of Fibonacci numbers (or their leading digits; floating point variables can be used), this issue makes the formula useless when we are to find the remainders of Fibonacci numbers (in particular, their trailing digits).

The formula that proves most useful for our purpose is as follows. It uses the basic notions of linear algebra, i.e. matrices and vectors:

$$\begin{bmatrix} F_1 \\ F_0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad \begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix}$$

This clearly gives a different closed-form expression of Fibonacci numbers:

$$\begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

To compute $F_n$, one can just use the formula we have obtained; that is, compute the $n$-th power of the appropriate $2 \times 2$ matrix and multiply it by a vector. The key part is exponentiation—it involves $n$, which might be really large. To perform this operation quickly, one can use a method called *exponentiation by squaring*, which allows the $n$-th power of a matrix to be found using $O(\log n)$ matrix multiplications. (This technique is much more general: it is commonly used to compute powers in the $\mathbb{Z}_M$ ring and in many other circumstances.) This method is based on the following simple observation:

$$A^{2n} = (A^2)^n,$$
$$A^{2n+1} = A \cdot (A^2)^n.$$

Direct application of these equalities leads to a recursive algorithm. To improve its practical performance (reduce the constant factor in time cost) and the memory consumption (to constant), the algorithm is usually implemented in the iterative form:

```
Algorithm Power(A, n)
  P := I {identity matrix}
  while n > 0 do
    if n mod 2 = 1 then
      P := P · A
    A := A · A
    n := n div 2
  return P
```

This algorithm gives an $O(\log n)$ time and constant space algorithm for computing $F_n$ in $\mathbb{Z}_M$.

**Leonardo numbers.** The recursive formula for Leonardo numbers is

$$L_0 = L_1 = 1, \quad L_{n+1} = L_n + L_{n-1} + 1, \tag{1}$$

which is slightly more complicated. In particular, this formula cannot be encoded in a $2 \times 2$ matrix, as we have done before. Speaking in terms of linear algebra, $L_{n+1}$ is not a linear combination of $L_n$ and $L_{n-1}$*.

This means that we need to use the matrix exponentiation technique in a more subtle way. The key observation is that $L_{n+1}$ is a linear function of $L_n$, $L_{n-1}$ and 1 (a constant sequence). Hence, we can use a larger (but still constant-sized) matrix. This leads to the following recursive formula:

$$\begin{bmatrix} L_1 \\ L_0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \quad \begin{bmatrix} L_{n+1} \\ L_n \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} L_n \\ L_{n-1} \\ 1 \end{bmatrix}$$

Again we get an $O(\log n)$ time and constant-space solution.

### The original solution by the problem's author

We have learned how to find $L_n$ in $O(\log n)$ time. However, our task is harder: we are to compute a sum $\sum_{i=0}^{n} L_i^k$. Let us start by solving it for $k = 1$ and $k = 2$. This will give us some insight for the general case.

* When writing of linear combinations, we abuse the notation—so, for example, by $L_{n+1}$ we mean a sequence whose $n$-th term is $L_{n+1}$. If we were formulating this statement without the notion of linear algebra on sequences, we would need to say that $L_{n+1}$ cannot be written as a linear combination of $L_n$ and $L_{n-1}$ with coefficients independent of $n$.



**Sum of Leonardo numbers.** The current task is to find the $n$-th term of the sequence $S_{1,n} = \sum_{i=0}^{n} L_i$. Directly from the definition one gets

$$S_{1,0} = L_0 = 1, \quad S_{1,n} = S_{1,n-1} + L_n.$$

Writing the recursive formula in matrix form led us to the effective algorithms, so let us try it again. Clearly, $S_{1,n}$ can be expressed as a linear combination of the $S_{1,n-1}$ term and $L_n$. This is the reason why we extend the previous matrix:

$$\begin{bmatrix} S_{1,0} \\ L_1 \\ L_0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \quad \begin{bmatrix} S_{1,n} \\ L_{n+1} \\ L_n \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} S_{1,n-1} \\ L_n \\ L_{n-1} \\ 1 \end{bmatrix}$$

This matrix allows us simultaneously to find both Leonardo numbers and their sums. Note that this is the first time in this solution where two coordinates of the left-hand side vectors are not merely rewritten coordinates of the right-hand side vector. The only assumption we need in order to use the matrix exponentiation technique is that a vector $[S_{1,n}, L_{n+1}, L_n, 1]$ can be expressed as a linear function of $[S_{1,n-1}, L_n, L_{n-1}, 1]$ (with the linear function independent of $n$).

Hence, we have devised an algorithm that performs $O(\log n)$ multiplications of $4 \times 4$ matrices, which gives $O(\log n)$ time and constant space complexity. Thus, let us go a step further.

**Sum of the Leonardo numbers' squares.** Now we are to determine the values $S_{2,n} = \sum_{i=0}^{n} L_i^2$. The recursive formula is

$$S_{2,0} = L_0^2 = 1, \quad S_{2,n} = S_{2,n-1} + L_n^2.$$

This time the problem is not so easy, $L_n^2$ should be an element of the vector, and it is not $L_n^2$ but $L_n$ for which we have a linear recursive formula. However, the formula we have gives a more complicated but still linear recursive formula for $L_{n+1}^2$:

$$L_{n+1}^2 = (L_n + L_{n-1} + 1)^2 = L_n^2 + L_{n-1}^2 + 2L_nL_{n-1} + 2L_n + 2L_{n-1} + 1.$$

The "new" term here is $L_nL_{n-1}$. Hence we need one more formula:

$$L_{n+1}L_n = (L_n + L_{n-1} + 1)L_n = L_n^2 + L_nL_{n-1} + L_n.$$

Summing up, the matrix recursive formula can be written as follows:

$$\begin{bmatrix} S_{2,0} \\ L_1^2 \\ L_1L_0 \\ L_0^2 \\ L_1 \\ L_0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} S_{2,n} \\ L_{n+1}^2 \\ L_{n+1}L_n \\ L_n^2 \\ L_{n+1} \\ L_n \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 1 & 2 & 2 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} S_{2,n-1} \\ L_n^2 \\ L_nL_{n-1} \\ L_{n-1}^2 \\ L_n \\ L_{n-1} \\ 1 \end{bmatrix}$$

Once again we get a solution of time complexity $O(\log n)$, but this time the constant hidden in the $O$ notation is quite large.

**The general case.** Having obtained some experience by analyzing the special cases of $k = 1$ and $k = 2$, we are finally ready to tackle the problem in its general form. We still want to solve it by matrix exponentiation. The elements of this matrix, and even its size, will depend on $k$. Hence, this time our program will need to compute this matrix before it exponentiates it.

For the sequence $S_{k,n} = \sum_{i=0}^{n} L_i^k$ we have a trivial formula:

$$S_{k,0} = L_0^k = 1, \quad S_{k,n} = S_{k,n-1} + L_n^k.$$

The linear recursive formula that lets us find $L_{n+1}^k = (L_n + L_{n-1} + 1)^k$ is a consequence of Newton's trinomial formula, and is a rather complicated one:

$$L_{n+1}^k = (L_n + L_{n-1} + 1)^k = \sum_{r=0}^{k} \sum_{s=0}^{k-r} \binom{k}{r,s,k-r-s} L_n^r L_{n-1}^s 1^{k-r-s}.$$

The coefficients that occur above are the trinomial coefficients defined as

$$\binom{p}{r,s,t} = \frac{p!}{r!\,s!\,t!} \quad \text{for } r,\ s,\ t \geq 0 \text{ and } p = r + s + t. \tag{2}$$

Despite being complicated, this formula allows us to express $L_{n+1}^k$ as a linear function of $L_n^r L_{n-1}^s 1^{k-r-s} = L_n^r L_{n-1}^s$, where $r$, $s$ and $k - r - s$ are all nonnegative integers. Fortunately (as we will show in a moment), for any such $r$, $s$ the term $L_{n+1}^r L_n^s$ is also expressible as the linear function of the same terms.

This means that the vector that we will compute is

$$[S_{k,n}, L_{n+1}^k, L_{n+1}^{k-1}L_n^1, \ldots, L_{n+1}^1 L_n^{k-1} L_n^k, L_{n+1}^{k-1}, \ldots, L_n^{k-1}, \ldots, L_{n+1}^1, L_n^1, 1].$$



Its length is $1 + \binom{k+2}{2} = \Theta(k^2)$. This is also the size of the square matrix whose $n$-th power we will compute. The time complexity of multiplying two square matrices of this size can in practice be bounded by $O(k^6)$. For theoretical considerations, one could use a better bound of $O(k^{2\omega})$ to express this cost*.

What remains to complete the solution is the first phase of the algorithm— we need to compute the elements of the matrix. For $k \leq 2$ we did this manually; now we need an algorithm. From Newton's trinomial formula we get:

$$L_{n+1}^p L_n^q = (L_n + L_{n-1} + 1)^p L_n^q = \sum_{r=0}^{p} \sum_{s=0}^{p-r} \binom{p}{r,s,p-r-s} L_n^{q+r} L_{n-1}^s.$$

This means that the first row of the matrix is 1, 1, 0, ..., 0, while—due to the above formula—the terms of all other rows are zeros or trinomial coefficients. If we know the values of the trinomial coefficients, the matrix can be computed in $O(k^4)$ time, which is good since the number of entries in the matrix is also $O(k^4)$. (The details of this purely technical problem are left as an exercise.)

The definition of trinomial coefficients stated in (2) is unfortunately not appropriate for computation in the $\mathbb{Z}_M$ ring. That is because it involves division, which is not well defined in this ring. Thus, we need a better definition. To simplify the formulas, let us extend the definition and assume that $\binom{p}{r,s,t} = 0$ if $p \neq r + s + t$ or there is a negative number among $r$, $s$, $t$. In this notation it is easy to verify that for $p \geq 1$:

$$\binom{p}{r,s,t} = \binom{p-1}{r-1,s,t} + \binom{p-1}{r,s-1,t} + \binom{p-1}{r,s,t-1}$$

Obviously the border case is $\binom{0}{0,0,0} = 1$.

Hence, all the trinomial coefficients we need (i.e. those with $p \leq k$) can be found in $O(k^3)$ time, so the whole matrix can be found in $O(k^4)$. Unfortunately, the procedure is technically complicated.

This concludes the description of the algorithm. Its overall time complexity is $O(k^{2\omega} \log n) = O(k^6 \log n)$ and it consumes $O(k^4)$ memory. For $k \leq 13$ and $\log n \leq 64$, and with a small constant hidden in $O$ notation (matrices have size $\frac{k^2}{2} + O(k)$), this solution is fast enough to earn the perfect score.

* The naïve algorithm multiplying two square matrices of size $m \times m$ works in $\Theta(m^3)$. A number $\omega$ is defined as the smallest real number such that two such matrices can be multiplied in $O(m^\omega)$. The algorithms used in practice reach $O(m^{\log_2 7}) = O(m^{2.808})$. Coppersmith and Winograd devised a very complicated algorithm in 1990, which achieves $O(m^{2.3755})$. At the end of 2011, this long-standing record was beaten: Virginia Vassilevska Williams published a paper with an algorithm that uses $O(m^{2.3727})$ time. In our case, this leads to $O(k^{4.7452})$ time complexity.

**A faster and simpler solution**

Analysing the previous solution, most readers are probably not fond of the huge sums that appear there because of the trinomial formula. If it were binomial—that is, if we computed the power of the sum of two terms—the formulas would be much simpler.

This would be the case if we somehow got rid of the constant term in the recursive formula (1) for Leonardo numbers. Here is how we can do that.

Let us start with the following lemma (which we state in a generalized version in order to give a better insight into the technique we use). Its proof is very simple and we leave it as an exercise for the reader.

***Lemma 1.*** Let $x_n$ be the sequence defined by the recursive formula of the following form:

$$x_n = a_1x_{n-1} + a_2x_{n-2} + \cdots + a_lx_{n-l} + b,$$

where $a_1 + a_2 + \cdots + a_l \neq 1$. Then the sequence

$$y_n = x_n + \frac{b}{a_1 + \cdots + a_l - 1}$$

satisfies the formula

$$y_n = a_1y_{n-1} + a_2y_{n-2} + \cdots + a_ly_{n-l}.$$

In our case, $x_n$ is $L_n$, so from (1) we get $l = 2$, $a_1 = a_2 = b = 1$. Hence we obtain a sequence $K_n = L_n + 1$, which can be defined by

$$K_0 = 2, \quad K_1 = 2, \quad K_{n+1} = K_n + K_{n-1}.$$

An attentive reader has probably already noticed that

$$K_n = 2F_{n+1}, \tag{3}$$

but this is not important for us—the crucial thing is that we have a recursive formula without the constant term.

Our situation is unfortunately not as good at it may seem. In order to find $S_{k,n} = \sum_{i=0}^{n} L_i^k$ we need to use the formula



$$S_{k,n} = \sum_{i=0}^{n} L_i^k = \sum_{i=0}^{n} (K_i - 1)^k = \sum_{i=0}^{n}\sum_{j=0}^{k} \binom{k}{j} K_i^j (-1)^{k-j} =$$

$$= \sum_{j=0}^{k} \left( \binom{k}{j} (-1)^{k-j} \left( \sum_{i=0}^{n} K_i^j \right) \right).$$

Let us therefore define $T_{j,n} = \sum_{i=0}^{n} K_i^j$. In order to compute $S_{k,n}$ it is enough to determine $T_{j,n}$ for all $0 \le j \le k$ and spend $O(k^2)$ time on computing the binomial coefficients (for example, by generating an appropriate part of the Pascal triangle).

**Computing $T_{k,n}$.** Let us start in the same way as we did for $S_{k,n}$. Again we have a trivial recursion:

$$T_{k,0} = K_0^k = 2^k, \quad T_{k,n} = T_{k,n-1} + K_n^k.$$

Moreover,

$$K_{n+1}^k = (K_n + K_{n-1})^k = \sum_{r=0}^{k} \binom{k}{r} K_n^r K_{n-1}^{k-r}.$$

Note that this time we obtain a linear function of just $k + 1$ terms. What is more,

$$K_{n+1}^p K_n^{k-p} = (K_n + K_{n-1})^p K_n^{k-p} = \sum_{r=0}^{p} \binom{p}{r} K_n^{k-p+r} K_{n-1}^{p-r},$$

so the vector

$$v_{n+1} = [T_{k,n}, K_{n+1}^k, K_{n+1}^{k-1}K_n, \ldots, K_{n+1}K_n^{k-1}, K_n^k]$$

can be expressed as a linear function of the similar vector $v_n$.

This time, the matrix in the recursive formula is simple enough to be written down below. Its size is $(k+2) \times (k+2)$ and its main part is the Pascal triangle.

$$\begin{bmatrix} T_{k,0} \\ K_1^k \\ K_1^{k-1}K_0 \\ \vdots \\ K_1K_0^{k-1} \\ K_0^k \end{bmatrix} = \begin{bmatrix} 2^k \\ 2^k \\ 2^k \\ \vdots \\ 2^k \\ 2^k \end{bmatrix} \quad \begin{bmatrix} T_{k,n} \\ K_{n+1}^k \\ K_{n+1}^{k-1}K_n \\ \vdots \\ K_{n+1}K_n^{k-1} \\ K_n^k \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & k & \cdots & k & 1 \\ 0 & 1 & k-1 & \cdots & 1 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 1 & 1 & \cdots & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 & 0 \end{bmatrix} \begin{bmatrix} T_{k,n-1} \\ K_n^k \\ K_n^{k-1}K_{n-1} \\ \vdots \\ K_nK_{n-1}^{k-1} \\ K_{n-1}^k \end{bmatrix}$$

Of course, this matrix can be computed in $O(k^2)$, which is the number of its entries. The values of $T_{k,n}$ can thus be found in $O(k^\omega \log n) = O(k^3 \log n)$ time using $O(k^2)$ space. Therefore we get a new algorithm for our main problem. Its time complexity is $O(k^{\omega+1} \log n) = O(k^4 \log n)$ and it uses $O(k^2)$ space. Thus we have created a solution that is simultaneously simpler, faster and less memory-consuming than the previous one.

**Even faster algorithm**

It turns out that the algorithm we have just presented is still not the best that we can do. That is rather typical for such problems of computational number theory. The time complexity depends heavily on the formulas we use to build an algorithm. Often, some interesting and unexpected ones lead to fast, highly nontrivial, and sometimes even simple solutions. In a way, this is what happened in the previous section. In order to improve this result, we need to use more complicated mathematical tricks.

Our main aim is very simple: obtain a new, better recursive formula for $k$-th powers of Leonardo numbers.

**Fibonacci numbers once more.** We noticed in (3) that $K_n^k = 2^k F_{n+1}^k$. Moreover, we have seen the closed-form for Fibonacci numbers

$$F_n = \frac{\varphi^n - \psi^n}{\sqrt{5}}, \tag{4}$$

where $\varphi = \frac{1+\sqrt{5}}{2}$ and $\psi = \frac{1-\sqrt{5}}{2}$. Note that $\varphi$ and $\psi$ are the roots of the polynomial $x^2 - x - 1$. This in particular means that $\varphi\psi = -1$ and $\psi^2 = \varphi^{-2}$. Thus

$$5^{\frac{k}{2}} F_n^k = (\varphi^n - \psi^n)^k = \sum_{i=0}^{k} \binom{k}{i} (-1)^{k-i} (\varphi^i \psi^{k-i})^n.$$

Hence, the sequence $F_n^k$ turns out to be a linear combination of sequences $\{(\varphi^i \psi^{k-i})^n : 0 \le i \le k\}$. Similarly, using (3) we conclude that $K_n^k$ is such a combination as well. Since

$$L_n^k = (K_n - 1)^k = \sum_{i=0}^{k} \binom{k}{i} (-1)^{k-i} K_n^i,$$

the sequence $L_n^k$ is a linear combination of $K_n^i$ for $0 \le i \le k$. Combining these two facts, we find that $L_n^k$ is a linear combination of sequences

$$\{(\varphi^i \psi^{j-i})^n : 0 \le i \le j \le k\}.$$



**The main trick.** The equality $\varphi\psi = -1$ turns out to have deep and useful implications.

***Lemma 2.*** All elements of the set $\{\varphi^i \psi^{j-i} : 0 \le i \le j \le k\}$ are roots of the polynomial

$$P(x) = \prod_{t=-k}^{k} (x^2 - \varphi^{2t}). \tag{5}$$

*Proof.* Let us consider a fixed element $x_{ij} = \varphi^i \psi^{j-i}$ from the set. We have

$$x_{ij}^2 = (\varphi^i \psi^{j-i})^2 = \varphi^{2i} \psi^{2(j-i)} = \varphi^{2i-2j+2i} = \varphi^{2(2i-j)}.$$

Moreover, $-k \le 2i - j \le k$, so for $t = 2i - j$ in the product we get $x_{ij}^2 - \varphi^{2t} = \varphi^{2(2i-j)} - \varphi^{2(2i-j)} = 0$. □

Let us write the polynomial $P(x)$ in the power basis:

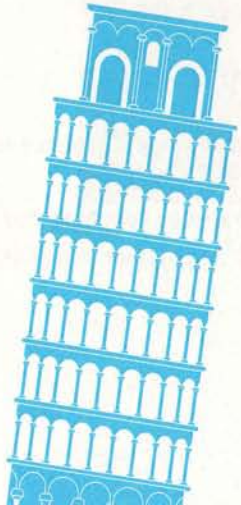
$$P(x) = x^{4k+2} + p_{2k} x^{4k} + p_{2k-1} x^{4k-2} + \cdots + p_1 x^2 + p_0.$$

Clearly, if $P(x) = 0$ then also $P(x) x^{n-4k-2} = 0$, which means that

$$x^n = -p_{2k} x^{n-2} - p_{2k-1} x^{n-4} + \cdots - p_1 x^{n-4k} - p_0 x^{n-4k-2}.$$

Values $x_{ij} = \varphi^i \psi^{j-i}$ are all roots of $P$, which by the above reasoning means that the geometrical progressions $a_n = (\varphi^i \psi^{j-i})^n$ all satisfy the following recursive formula:

$$a_n = -p_{2k} a_{n-2} - p_{2k-1} a_{n-4} + \cdots - p_1 a_{n-4k} - p_0 a_{n-4k-2}. \tag{6}$$



Being a linear combination of such geometric progressions, $L_n^k$ must also satisfy this formula.

This is very good news for us, since we can solve our problem by naïvely computing the first few elements of $L_n^k$ and then using the following recursive formula:

$$\begin{bmatrix} S_{k,4k+1} \\ L_{4k+2}^k \\ L_{4k+1}^k \\ L_{4k}^k \\ \vdots \\ L_1^k \end{bmatrix} = \cdots \begin{bmatrix} S_{k,n+4k+1} \\ L_{n+4k+2}^k \\ L_{n+4k+1}^k \\ L_{n+4k}^k \\ \vdots \\ L_{n+1}^k \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & -p_{2k} & \cdots & 0 & -p_0 \\ 0 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 0 \end{bmatrix} \begin{bmatrix} S_{k,n+4k} \\ L_{n+4k+1}^k \\ L_{n+4k}^k \\ L_{n+4k-1}^k \\ \vdots \\ L_n^k \end{bmatrix}$$

Note that in order to compute $a_n$ using the formula (6), we need to know the previous values up to $a_{n-4k-2}$, which means that the size of our matrix is $(4k+3) \times (4k+3)$.

The only thing that remains is to find the $p_i$ values. What is more, these numbers need to be integers, so we can perform the computations in $\mathbb{Z}_M$. Note that from (4) we get

$$5F_t^2 = (\varphi^t - \psi^t)^2 = \varphi^{2t} - 2\varphi^t\psi^t + \psi^{2t} = \varphi^{2t} - 2(-1)^t + \varphi^{-2t},$$

so $\varphi^{2t} + \varphi^{-2t} = 5F_t^2 + 2(-1)^t$.

Let us in (5) join the factors for $t$ of the same absolute value:

$$(x^2 - \varphi^{2t})(x^2 - \varphi^{-2t}) = x^4 - (\varphi^{2t} + \varphi^{-2t})x^2 + 1 = x^4 - (5F_t^2 + 2(-1)^t)x^2 + 1.$$

Finally, we get

$$P(x) = (x^2 - 1)\prod_{t=1}^{k}\left(x^4 - (5F_t^2 + 2(-1)^t)x^2 + 1\right).$$

This means that the coefficients of this polynomial are integers and we can find them (or rather their residues modulo $M$) in $O(k^2)$ time.

The matrix we exponentiate is of $\Theta(k)$ size, so we can conclude that we have invented a solution that works in $O(k^\omega \log n) = O(k^3 \log n)$ time using $O(k^2)$ space.

**ERYK KOPCZYŃSKI**

Eryk Kopczyński learned to program in BASIC at the age of six, back in the days of Amstrad CPC computers. In 1994 he was a finalist of the first Polish Olympiad in Informatics. He also received gold medals at the International Mathematical Olympiad and the International Olympiad in Informatics. He has represented the University of Warsaw in ACM-ICPC World Finals twice, in 2000 and 2002. Later on, he completed his Ph.D. degree and is currently working at the Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, doing research in theoretical computer science.

Eryk Kopczyński still actively participates in international online programming contests, such as TopCoder (he won TopCoder Open in 2005 and is currently ranked 18th) and Google Code Jam. He is also an author of programming tasks and a coach at programming competitions.

Eryk Kopczyński enjoys biking, mountain trekking, music and all kinds of games.

# / Ritual

**Contest:** Polish Olympiad in Informatics Training Camp 2007
**Task author:** Eryk Kopczyński
**Memory:** 32 MB
http://main.edu.pl/en/archive/ontak/2007/ryt

BAB (Bytean Algorithmic Bureau) is a private agency, employing a team of computer scientists, that helps wealthy people solve their confidential problems with the aid of powerful computer systems. Recently, BAB was invited to meet the Grand Master of the Bytoo Worship. On the way to the Bytoo Temple the BAB team managed to collect some information about the Worship.

*The Bytoo Worship started when the Prophet received the great silicon slab with two numbers written on it: the Great Sacred Number and the Small Sacred Number. [...] As a part of the ritual, each morning, priests write the Great Sacred Number on the inner side of the wall of the Temple. The priests erase some of its digits during the day, according to a secret rule, and leave the remaining digits on the wall for the night. Next morning, the Great Sacred Number is written on the wall again and the ritual continues.*

The Grand Master of the Bytoo Worship gave BAB a preliminary test: they were to figure out the secret rule of erasing digits. On the first evening, BAB noticed that the remaining digits formed a palindromic number: the first digit is the same as the last one, the second digit is the same as the last but one, and so on. That was, however, only a part of the secret rule. After a few days, BAB used Euclid's algorithm to discover the second part of the rule: the number that remains after the digits are erased must be divisible by the number of days in the Bytoo week, which is equal to the Small Sacred Number, 666. The remaining number may have leading and trailing zeros, but it must have at least one digit (the priests cannot erase all digits at a time).

The Grand Master congratulated BAB on successfully completing the test and then revealed their real mission. Each day, the priests must erase a different configuration of digits. The end of the world will occur on the first day when the priests are unable to find any new configuration. The Grand Master would like to know which day of the week will be the last day (priests erased the first configuration of digits on the first day of the Bytoo week). He asked BAB to write a program to compute this information.



**Task**

Write a program that:

- reads the Great Sacred Number from the input,
- computes the day of the week after which the world will end (assuming the perfect accuracy of the priests),
- writes the result to the output.

**Input**

The first line of the input contains one integer $G$ ($1 \le G < 666^{36}$), the Great Sacred Number.

**Output**

The first and only line of the output should contain one number from the set {1, 2, ..., 666}, specifying the day of the week after which the world will end.

**Example**

For the input data:
6666666
the correct result is:
42

**Explanation of the example:** There are 35 ways to erase four digits of the Great Sacred Number so that the number 666 remains on the wall and seven ways to erase one digit leaving 666666 on the wall. Obviously, 666 and 666666 satisfy both requirements of the ritual.

# / Solution

This problem appeared in the Polish Olympiad in Informatics Training Camp in 2007. My idea was to train the participants in dynamic programming algorithms by combining their two relatively standard applications.

However, this combination turned out not to be as easy and straightforward as it seemed. There are lots of subtleties here. The problem features two cult numbers: 666 and 42, which suggests that the author cared about style and wanted to use interesting numbers, instead of mundane powers of 10 and 2. However, this was not the only reason: the number 666 was not chosen arbitrarily; we had to take advantage of one of its special properties. Otherwise, one would obtain a solution with significantly worse time and space complexities, which would not run within the time and space limits imposed.

As far as I remember, these subtleties meant that none of the participants was able to solve the problem during the contest at the training camp. Moreover, I arrived at the camp later, after the problem had been used, and found that the solution presented by the trainers was the suboptimal one (they did not take advantage of the properties of 666). I think that the idea of using a special property of an innocent number is quite original and interesting. With hindsight I regret that this problem was used at a training camp, as it would probably have been appropriate for a more serious competition.

**Solution**

Let's first consider the problem in the case where we do not care about divisibility, and only have to compute the number of ways of erasing digits so that the remaining digits form a palindrome. Experienced coders quickly discover that this can be done with dynamic programming.

Let $l_1 l_2 \ldots l_N$ be the digits of the Great Sacred Number. First, note that each subpalindrome of length greater than 2, having $l_a$ and $l_b$ as its first and last digits, consists of these two digits, two sequences of erased digits $\cancel{l_{a+1} \ldots l_{c-1}}$ and $\cancel{l_{d+1} \ldots l_{b-1}}$, and a *heart* (lying between the two erased sequences), which is a subpalindrome with $l_c$ and $l_d$ as its first and last digits.

For each pair $(a, b)$, where $a \le b$, let $P_{a,b}$ denote the number of palindromes obtained by erasing some digits in the sequence $l_a l_{a+1} \ldots l_b$, except for $l_a$ and $l_b$, which have to remain. From the property described above we find that $P_{a,b}$ can be calculated from the values $P_{c,d}$ for $(c, d)$ such that $a < c \le d < b$:

$$P_{a,b} = \begin{cases} 1 & \text{for } a = b, \\ 0 & \text{for } l_a \ne l_b, \\ 1 + \sum\limits_{c,d:a<c\le d<b} P_{c,d} & \text{for } a \ne b \text{ and } l_a = l_b. \end{cases}$$

The formula above leads to an algorithm with complexity $O(N^4)$. For each pair $(a, b)$ we have to test all the positions of the heart $(c, d)$, so we have four nested loops. We start with pairs where $a = b$ and then proceed to greater and greater values of $b - a$.

How can we improve the running time of this algorithm? Maybe we should redefine $P_{a,b}$ and no longer force $l_a$ and $l_b$ to remain? In this case, we would get the following formula: $P_{a,b} = P_{a+1,b} + P_{a,b-1} + c_{a,b}$, where $c_{a,b}$ equals 0 for $l_a \ne l_b$ and $1 + P_{a+1,b-1}$ for $l_a = l_b$. The complexity is reduced to only $O(N^2)$!

Unfortunately, this solution is not correct. For example, consider 161. The palindrome $\cancel{161}$ is counted twice: both by extending $\cancel{16}$ and $\cancel{61}$. This is because, when erasing digits surrounding the heart of our palindrome, the formula allows us to choose arbitrarily whether we extend to the left or to the right. Thus, if we consider 61111611116, its subpalindrome 666 is counted $\binom{8}{4} = 70$ times, since there are 70 ways of ordering four erasures of 1s to the left, and four erasures of 1s to the right.

How do we fix this? We simply force digits to the right of the heart to be erased before the digits to the left are erased. Let $P_{a,b}$ be the number of subpalindromes of $l_a \ldots l_b$ that do not have to include $l_a$ and $l_b$, and let $Q_{a,b}$ be the number of subpalindromes that include $l_a$. We extend the palindrome in the following way. Suppose that we have a subpalindrome with digits $l_a$, $l_c$, $l_d$ and $l_b$ (all of which have not been erased); all digits between $l_{a+1}$ and $l_{c-1}$, and between $l_{d+1}$ and $l_{b-1}$, are erased. Then:

- The heart $l_c \ldots l_d$ is counted in $Q_{c,d}$.
- We extend it to the right, thus we count $l_c \ldots l_d \cancel{l_{d+1} \ldots l_{b-1}}$ in $Q_{c,b-1}$.
- The same subpalindrome will be counted in $P_{c,b-1}$.
- We extend it to the left, thus $\cancel{l_{a+1} \ldots l_{c-1}} l_c \ldots l_d \cancel{l_{d+1} \ldots l_{b-1}}$ will be counted in $P_{a+1,b-1}$.
- We extend it by $l_a$ and $l_b$, and count the resulting palindrome in $Q_{a,b}$.

The following formulae describe this process:

$$Q_{a,b} = \begin{cases} 1 & \text{for } a = b, \\ Q_{a,b-1} + P_{a+1,b-1} + 1 & \text{for } a \neq b \text{ and } l_a = l_b, \\ Q_{a,b-1} & \text{for } a \neq b \text{ and } l_a \neq l_b, \end{cases}$$

$$P_{a,b} = Q_{a,b} + P_{a+1,b}.$$

The formulae above force the erased digits to be added first to the right from the heart, then from the left. After we compute all these values, the total number of palindromes is stored in $P_{1,N}$.

Now, how should we take divisibility into account? Again, let's assume first that we only want to know how many subnumbers of the Great Sacred Number are divisible by $M$. This can also be done with dynamic programming. Let $V_{a,d}$ be the number of subnumbers of $l_1 \ldots l_a$ that are congruent to $d$ modulo $M$. We can express $V_{a,d}$ with a recursive formula:

$$V_{a,d} = V_{a-1,d} + \sum_{c:10c+l_a \equiv d} V_{a-1,c}.$$

The first component describes subnumbers where $l_a$ is erased, and the second describes subnumbers where it remains. At the beginning we set $V_{0,0} = 1$ and $V_{0,d} = 0$ for $d > 0$. By the formula, $V_{N,0}$ is equal to the number of all subnumbers of the Great Sacred Number that are divisible by $M$ (including the one in which we have erased all the digits). Of course, we don't calculate the sum in the formula above by trying all the possible values $c$ for each $d$ (which would lead to $O(NM^2)$ time complexity). Instead, for each value of $a$ we try each value of $c$, and we add $V_{a-1,c}$ to respective $V_{a,d}$ (namely, for $d = (10c + l_a) \bmod M$).

Now we have to combine the two solutions. Note that the key in the second solution was to classify the subnumbers of $l_1 \ldots l_a$ with respect to their remainders modulo $M$. How about simply applying this to our palindrome counting algorithm? Let $P_{a,b,d}$ be the number of subpalindromes of $l_a \ldots l_b$ congruent to $d$, and $Q_{a,b,d}$ be the number of those that include $l_a$.

This will not be enough, though. Consider again the subpalindrome starting with $l_a$ and ending with $l_b$, whose heart ranges from $l_{a'}$ to $l_{b'}$. Assume that the heart divided by $M$ leaves a remainder $c$. What is the big palindrome congruent to? We do not have enough information, since the answer is $(l_a(10^{i+1} + 1) + 10c) \bmod M$, where $i$ is the length of the heart, which we do not know.



Thus, we need to classify our palindromes again, with respect to their lengths: let $P_{a,b,d,i}$ be the number of subpalindromes of $l_a \ldots l_b$ of length $i$ and congruent to $d$. We classify $Q$-type palindromes in a similar way. This leads to an algorithm of complexity $O(N^3M)$. Unfortunately, this complexity is too high...

How can we fix this? We need to use the fact that $M$ is equal to 666. What are the remainders of dividing $10^{i+1} + 1$ by 666? For $i = 0, 1, 2$ we get 11, 101, and 335, respectively. However, for $i = 3$ we get 11 again, for $i = 4$ we get 101, and so on. Therefore, it is enough to consider a coarser classification: let $P_{a,b,d,i}$, for $i = 0, 1, 2$, be the number of subpalindromes of $l_a \ldots l_b$ congruent to $d$, whose length is congruent to $i$ modulo 3.

This leads to an algorithm of complexity $O(N^3)$, and the result is obtained by summing $P_{1,N,0,i}$ over all three values of $i$. Be aware that the problem contains a small but nasty trap: if the total number of palindromes is divisible by 666, we should answer 666 instead of 0.

We should also be careful with space usage. We cannot just memorize all the values of $P_{a,b,d,i}$. However, to calculate $P_{a,b,d,i}$, where $b - a = L$, we only need to know values for which $b - a \in \{L - 2, L - 1, L\}$. Thus, it is enough to remember only three arrays $P_{a,d,i}$ corresponding to three consecutive values of $b - a$, and similarly only three arrays $Q_{a,d,i}$. We encourage the reader to figure out the technical details.

# / Questions

**Contest:** Algorithmic Engagements 2008
**Task author:** Eryk Kopczyński
**Memory:** 32 MB
http://main.edu.pl/en/archive/pa/2008/pyt

One day, king Bytour invented a game for his sons. He described the game to his counsellor, sorcerer Bytelean, in the following way:

*I ordered my three sons (numbered with integers 1, 2, 3) to stand in a row and I placed a gold or a silver crown upon the head of each of them. Son number 1 could then see the crowns of sons 2 and 3, and son number 2 could see the third son's crown. Each of the sons knew that there were at most two gold crowns in total. After that, I asked son number 1 if he knew his crown's color. He answered that he did not know. After that, I asked son number 2 if he knew his crown's color. He also answered that he did not know.*

At this moment, Bytelean interrupted Bytour and told him that he already knew what crown prince 3 wore. Bytour asked him how he knew that. He answered as follows:

*If prince 1 saw two gold crowns, then he would know that he had received a silver one (because there are at most two gold crowns). He answered* NO, *so this situation could not have taken place. Now, if prince 2 saw a gold crown on the third prince's head, then he would know that he had received a silver one—in the opposite case, prince 1 would have answered* YES. *But he did not know that. Therefore, it is a certainty that prince 3 had a silver crown.*

Your task is to implement a (quite) general simulator of such situations. Facts about which the king can ask the princes and the sorcerer (in the preceding situation it was the crown's type) are encoded as a sequence of variables. Some of them are in some way related to the previous variables; for the remaining ones, only the ranges of their values are known. A more detailed description of the problem that is to be solved is contained in the *Input* section.

## Task

Write a program that:

- reads a description of a situation from the input,
- computes the answers that the sorcerer should give,
- writes the result to the output.



## Input

The first line of the input contains three integers, $P$, $V$ and $A$, separated by single spaces. $P$ denotes the number of princes (numbered from 1 to $P$), $V$ denotes the number of variables (numbered from 1 to $V$), and $A$ denotes the number of actions. You can assume that the following inequalities hold: $1 \le P \le 10$, $1 \le V \le 600$, $1 \le A \le 600$.

The following $V$ lines describe variables $v_1, v_2, \ldots, v_V$. Each of these lines is of the form "$Z_i\ A_i\ B_i$" (this notation contains single spaces), where $Z_i$ is one of the characters =, +, −, ∗, /, %, > and $A_i$ and $B_i$ are integers. Depending on the value of $Z_i$ specified in the description of $v_i$, different kinds of facts about the variables are provided. The meanings of the possible $Z_i$ characters are as follows:

| | |
|---|---|
| $= A_i\ B_i$ | $v_i$ is an integer between $A_i$ and $B_i$ ($-1{,}000{,}000 \le A_i \le B_i \le 1{,}000{,}000$) |
| $+ A_i\ B_i$ | $v_i = v_{A_i} + v_{B_i}$ (in this and all subsequent cases, $1 \le A_i, B_i < i$) |
| $- A_i\ B_i$ | $v_i = v_{A_i} - v_{B_i}$ |
| $* A_i\ B_i$ | $v_i = v_{A_i} \cdot v_{B_i}$ |
| $/ A_i\ B_i$ | $v_i = v_{A_i} / v_{B_i}$ (integer part of the division) |
| $\% A_i\ B_i$ | $v_i = v_{A_i} \bmod v_{B_i}$ (the remainder) |
| $> A_i\ B_i$ | $v_i = 1$ if $v_{A_i} > v_{B_i}$; $v_i = 0$ otherwise. |

This information is provided at the beginning of the game to all princes and also to the sorcerer. The following $A$ lines describe the actions. The following actions can occur (the descriptions of actions contain single spaces):

- **S** *g n*
  The value of $v_n$ is revealed to son number $g$. The fact of the revelation of this value to the son is revealed to all princes and to the sorcerer (this does not mean that they are provided with the value itself).
- **T** *g n*
  The king asks son number $g$ whether he knows the value of $v_n$. He is answered YES, and passes this answer to the sorcerer. (The other sons will hear the answer only when action **A** is performed—consequently, a few of them can answer questions "simultaneously" and the answers of some of them will not be used by others.)
- **N** *g n*
  Just as in the previous action, except that the king receives the answer NO.
- **X** *g n*
  Just as in the previous action, except that the king does not pass the answer to the sorcerer, instead asking him to guess the answer. The sorcerer answers YES, NO or I DON'T KNOW (the answer I DON'T KNOW means that the sorcerer

is not sure whether the son knew the answer to the king's question). During this action the king *does not inform* the sorcerer about the actual answer of prince $g$. The king also *does not pass* the sorcerer's answer to the princes.

→ **A** 0 0

All sons are provided with the answers given by all the other sons to all of the king's previously asked questions (answers given during actions **T**, **N** and **X**). During this action the king also *does not inform* the sorcerer about the answers that he received in actions of type **X**.

→ **M** $w$ $n$

The king tells the sorcerer that variable $v_n$ has value $w$.

→ **Q** 0 $n$

The king asks the sorcerer what possible values—according to his current knowledge—$v_n$ may have. The king *does not pass* the sorcerer's answer to the princes.

It should be assumed that both the princes and the sorcerer perform perfect reasoning—meaning that, at every moment, they are able to deduce all the facts that are implied by the constraints revealed by the king at the beginning of the game and by what has happened in the game so far. Additionally, each of them knows that all of them perform perfect reasoning.

### Constraints

The following constraints hold: $1 \le P \le 10$, $1 \le V \le 600$, $1 \le A \le 600$. Additionally, the number of all possibilities (i.e. the product of values $B_i - A_i + 1$ over all variables of type "=") does not exceed 600. In every theoretically possible valuation of variables, the absolute value of each variable is not greater than 1,000,000, and, in the case of operations "/" and "%", $v_{A_i}$ is nonnegative and $v_{B_i}$ is positive. Variable $v_X$ may appear in the definition of $v_Y$ only if $X < Y$.

### Output

For every action of type **Q**, exactly one line should be written out, containing a sequence of all possible values of $v_n$, from the smallest to the largest, separated by single spaces. For every action of type **X**, one line containing TAK (Polish for *yes*), NIE (Polish for *no*) or NIE WIEM (Polish for *I don't know*) should be written out. The answers should be written to the output in the order of appearance of the actions (in the input) to which they apply; the order is independent of the types of these actions.



### Example

For the input data:

| | | | |
|---|---|---|---|
| 3 | 7 | 16 | 3 princes, 7 variables, 16 actions. |
| = | 0 | 1 | $v_1$: crown color of son 1 (0 = silver, 1 = gold). |
| = | 0 | 1 | $v_2$: crown color of son 2. |
| = | 0 | 1 | $v_3$: crown color of son 3. |
| + | 1 | 2 | $v_4 = v_1 + v_2$. |
| + | 3 | 4 | $v_5 = v_3 + v_4 = v_1 + v_2 + v_3$, which is, in other words, equal to the total number of gold crowns. |
| = | 3 | 3 | $v_6 = 3$. |
| > | 6 | 5 | $v_7$ determines whether the number of gold crowns is fewer than 3. |
| **S** | 1 | 7 | All sons know that there are fewer than 3 gold crowns. |
| **S** | 2 | 7 | |
| **S** | 3 | 7 | |
| **M** | 1 | 7 | The sorcerer also knows that. |
| **S** | 1 | 3 | Prince 1 can see the crown of prince 3. |
| **S** | 2 | 3 | Prince 2 can see the crown of prince 3. |
| **S** | 1 | 2 | Prince 1 can see the crown of prince 2. |
| **X** | 3 | 3 | Does (according to the sorcerer) son 3 know his crown's color? (no). |
| **X** | 1 | 1 | Does (according to the sorcerer) son 1 know his crown's color? (the sorcerer does not know that). |
| **N** | 1 | 1 | Son 1 does not know his crown's color. |
| **A** | 0 | 0 | All sons are informed that 1 and 3 do not know their crown's colors. |
| **N** | 2 | 2 | Son 2 answers that he still does not know his crown's color. |
| **X** | 3 | 3 | The sorcerer knows that son 3 still does not know his crown's color (he did not hear 2's answer). |
| **A** | 0 | 0 | |
| **X** | 3 | 3 | Now son 3 knows. |
| **Q** | 0 | 3 | The sorcerer also knows that prince 3 has crown of type 0. |

the correct result is:

NIE
NIE WIEM
NIE
TAK
0

# / Solution

There are several logic puzzles similar to the one given to the sorcerer by the king; they are known as *induction puzzles*. One of the most interesting and well known is the one in which the king puts gold and silver crowns on the heads of his $n$ sons (everyone knows that somebody has a gold crown, and every son sees the crowns on the heads of the other sons), and then on each day he asks all his sons whether each knows the color of the crown on his own head. After some days, it turns out that all the sons know their colors! This puzzle appears in many versions. One of them is about wives who kill their husbands after deducing that they are cheating on them (each wife initially knows whether each man other than her husband cheats, and that there is a husband who cheats in the town)—this is called *Josephine's Problem*. Another variant is about people on an island who do not know the color of their own eyes (this version has appeared in an online comic series, *xkcd*). There are also several other similar puzzles, such as:

- Two logicians both receive a nonnegative integer, and they know that their numbers differ by 1. We ask them alternately whether they know their numbers. After a number of questions, both of them know.
- "My neighbor claimed to have three daughters. When I asked him about their ages, he said that the product of their ages was 36. This was not enough information for me, so he told me that the sum of their ages was equal to the number of his house. Despite knowing this number, I still could not calculate their ages. Thus, he said that his oldest daughter had a birthday yesterday. This was sufficient for me to solve the puzzle." The reader can solve it too, even though he does not know the sum.
- There are children from four families playing in a backyard. The number of children in each family is different. Doctor Watson tells Sherlock Holmes that the total number of children is less than 18, and reveals to him the product of the numbers of children in each family. This is not enough for Sherlock to deduce the number of children in every family. He has to ask Watson about the number of children in the smallest family, and this provides sufficient information. The reader can now deduce the answer too.

**Solution**

The game can be briefly summarized as follows: one of several possible situations has happened, and we do not know which one. In this solution the possible situations will be called *worlds*. From the task statement it follows that there are $M \leq 600$ worlds.

We label the possible worlds with integers from 1 to $M$ and calculate the values of all variables for each world. Let $val[v][n]$ be the value of variable number $n$ in world $v$. There's nothing interesting here.

Now we come across the main difficulty in this problem: how should we represent the knowledge available to the sons and the sorcerer? I think that this is the biggest challenge, because if we solve it right, the implementation is quite straightforward.

Imagine an observer who is watching the game. He knows which worlds are possible and the value of each variable in each world. In addition, he can observe that some variable or an answer to a question is revealed to the sons, although he is not given the value or the answer. While the observer does not know the contents of the information given to the sons, he can still simulate what would happen in every possible world. Such an observer has the knowledge that is common to all the sons. He can also tell what each son would do in any particular world.

Note that the sorcerer is independent of the sons. The information that he receives does not influence the sons in any way; nor does the observer benefit from it. In particular, the observer can ignore actions **M** (a value is revealed to the sorcerer) and **Q** (the king asks the sorcerer). Moreover, actions **N**, **T** and **X** (a son answers a question) differ only in what is told to the sorcerer, which means that they are the same from the observer's point of view. This observation allows us to remove the sorcerer from the game for the time being and focus on modeling the world from the observer's point of view.

A son can get to know information of two kinds: the value of a variable (action **S**) and the answer of another son (action **T**, **N** or **X**, followed by an announcement **A**). Note that, from the observer's point of view, none of these actions restricts the set of possible worlds: the son we are considering would get consistent information, possibly different for each world. The observer

knows, however, what would happen in any particular world. If we assume that we are in world $w$ and a son gets to know $v_n$, then afterwards he will only consider the worlds in which the value of $v_n$ is the same as in $w$. A similar thing happens when a son gets to know another prince's answer to a question. We can model this with a relation $\sim_g$, which captures "indiscernibility" for son $g$. We say that the two worlds $v$ and $w$ indiscernible ($v \sim_g w$) if, in world number $v$, son $g$ thinks that $w$ is potentially possible. Note that $\sim_g$ is an equivalence relation.

Initially, $\sim_g$ is a full relation: it has only one equivalence class containing all the worlds—son $g$ knows nothing and thinks that all the worlds are possible, no matter which one is the actual world. However, actions of type **S** and **A** break the equivalence classes into smaller ones.

Let us consider how the sons reason—from this we will know how to update the relation $\sim_g$. For example, consider $n$ sons, each of whom has a silver or a gold crown and knows the types of crowns of other sons. The relation $\sim_g$ is as follows: for each son, there are $2^{n-1}$ equivalence classes, each consisting of two worlds differing in the type of crown on the head of son $g$.

Now, every son learns whether somebody wears a gold crown. As a result, the equivalence class *all other sons have a silver crown* breaks into two, because son $g$ is now able to deduce the type of the crown on his own head.

Then, each son is asked whether he knows the type of the crown on his head. A son answers *yes* only if he sees only silver crowns.

After **A** each son gets to know the answers of other sons. Imagine we are a prince $g$ who sees only one gold crown. Before we discover the answers, two worlds $v$ and $w$ are possible from our point of view. However, the answers allow us to deduce the type of the crown on our head: if we had a silver crown, the owner of the gold crown would say *yes*; otherwise he would say *no*. In other words, the answer given by the prince wearing the gold crown allows us to differentiate worlds $v$ and $w$. Thus, the new knowledge breaks down equivalence classes *we see one gold crown* into two, $v \not\sim_g w$. Equivalence classes with more gold crowns remain untouched.

Again, we ask each son about the color of his crown—now, the answer is *yes* if we see at most one gold crown. After action **A**, just as before, equivalence classes with two gold crowns in sight are split. And so on: after each iteration of questioning and broadcast, more and more equivalence classes are broken.



Let us try to generalize this. When some son $g$ discovers, say, an answer of some other son to a question, we iterate through all equivalence classes of $\sim_g$. For a class consisting of worlds $w_1,\ldots,w_k$ we check what answer the son would receive in each world. Then we split the equivalence class into two smaller ones, one consisting of worlds where the answer would be *yes* and the other of worlds with the answer *no*.

Now, how should we implement this? We can do it as follows. We have arrays of Boolean values $valid[g][v][w]$. We set $valid[g][v][w]$ true if $v \sim_g w$. This leads to a solution that runs in $O(MV + APM^2)$ time.

We can also do it more effectively: instead of an array, we enumerate the equivalence classes in $class[g][v]$. Thus, $v \sim_g w$ if and only if $class[g][v] = class[g][w]$. The simplest way is to assume that $class[g][v]$ is a representative of the given class, i.e. $class[g][class[g][v]]$ always equals $class[g][v]$.

How might we implement questions (**X** $g$ $n$)? First, let $R$ be the set of class representatives, i.e. all values of $v$ such that $class[g][v]$ equals $v$. For each representative $v \in R$ we examine the value of $val[v][n]$. We now want to find the equivalence classes in which the value of variable $n$ is unambiguous—in these worlds son $g$ knows the value of $n$. Let us denote this set by $W$. Formally speaking, it is the set of representatives $v \in R$, such that for all $w$ satisfying $v \sim_g w$ we have $val[v][n] = val[w][n]$. The set $W$ can easily be calculated in $O(M)$—we simply examine all worlds $w$ and check whether $val[w][n]$ is equal to $val[class[g][w]][n]$. In world $v$ the son answers *yes* if $class[g][v] \in W$; otherwise he answers *no*. We write down the answers in $answer[v][a]$, where $v$ is the world index and $a$ is the action index.

Actions **S** $g$ $n$ are handled similarly. For each world $v$ we define a pair $(class[g][v], val[v][n])$ and sort these pairs lexicographically in $O(M \log M)$. After that, we are able to break classes into smaller ones in such a way that, in each resulting class, the values $val[v][n]$ are equal. Actions **A** are executed likewise, except that we use $answer[v][a]$ for all values of $a$ corresponding to new answers, instead of $val[v][n]$.

Now we know how to simulate the sons' knowledge. It is quite straightforward to simulate the sorcerer's knowledge: we simply consider $S$, the set of *all* worlds that are consistent with the sorcerer's knowledge. Initially, $S = \{1,\ldots,M\}$, but the set may lose some of its members after each action of type **M**, **T** or **N**. The king's queries are also implemented quite easily: we simply check the answer for each element of $S$ and output all possible answers.

**Epilogue**

I think that the problem is interesting for two reasons. First, it is based on well-known puzzles. Although induction puzzles are very interesting, they appear only rarely in programming contests. Second, we have to implement a very simple programming language, which is also rather uncommon. The language is very simple, but even so, each of the puzzles we mentioned in the introduction can be implemented in it, as a special case of our problem (of course, we need to limit the numbers in the first puzzle). And indeed, all these puzzles appeared in the test data used to grade the contestants.

**MARCIN KUBICA**

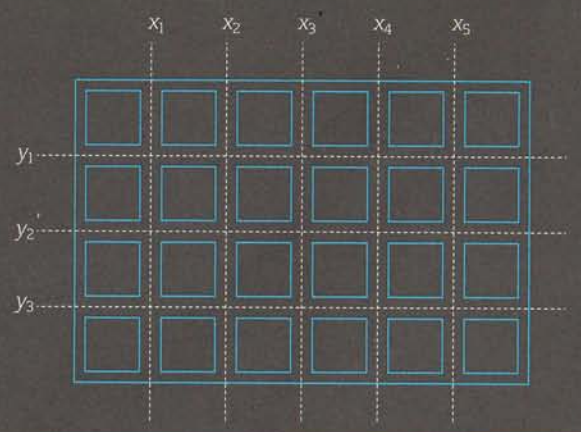
Marcin Kubica is an assistant professor at the Institute of Informatics, University of Warsaw. His research interests focus on combinatorial and text algorithms.

He is a scientific secretary of the Polish Olympiad in Informatics. In the past he has been involved in organizing numerous international programming contests such as the International Olympiad in Informatics (2005), Central European Olympiad in Informatics (2004, 2011) and Baltic Olympiad in Informatics (2001, 2008). Marcin Kubica is a former member of the International Scientific Committee of the International Olympiad in Informatics, from the years 2003-2006 and 2007-2010.

# / Chocolate

**Contest:** 10th Polish Olympiad in Informatics
**Task author:** Marcin Kubica
**Memory:** 32 MB
http://main.edu.pl/en/archive/oi/10/cze

We are given a bar of chocolate composed of $m \times n$ square pieces. The chocolate should be broken into single squares. Parts of the chocolate may be broken along vertical and horizontal lines, as indicated by the dashed lines in the picture. A single break of a part of the chocolate along a chosen vertical or horizontal line divides that part into two smaller pieces. Each break of a part of the chocolate is charged a cost, expressed by a positive integer. This cost does not depend on the size of the part that is being broken but only depends on the line the break goes along. Let us denote the costs of breaking along consecutive vertical lines with $x_1, x_2, \ldots, x_{m-1}$ and along horizontal lines with $y_1, y_2, \ldots, y_{n-1}$. The cost of breaking the whole bar into single squares is the sum of the successive breaks. The objective is to compute the minimum cost of breaking the whole bar of chocolate into single squares.




For example, if we break the chocolate presented in the picture first along the horizontal lines, and then break each resulting part along the vertical lines, then the cost of the whole operation will be $y_1 + y_2 + y_3 + 4 \cdot (x_1 + x_2 + x_3 + x_4 + x_5)$.



**Task**

Write a program that:

- reads the numbers $x_1, x_2, \ldots, x_{m-1}$ and $y_1, y_2, \ldots, y_{n-1}$,
- computes the minimum cost of breaking the whole chocolate bar into single squares,
- writes the result to the output.

**Input**

In the first line of the input there are two positive integers, $m$ and $n$ ($2 \le m, n \le 1000$), separated by a single space. In the successive $m-1$ lines there are numbers $x_1, x_2, \ldots, x_{m-1}$, one per line ($1 \le x_i \le 1000$). In the successive $n-1$ lines there are numbers $y_1, y_2, \ldots, y_{n-1}$, one per line ($1 \le y_i \le 1000$).

**Output**

In the first and only line of output your program should write one integer—the minimum cost of breaking the whole chocolate into single squares.

**Example**

For the input data:

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

the correct result is:

```
42
```

# / Solution

Oral examinations are exhausting, especially for the examiners. The majority of students who decide to take oral exams (to correct results from a prior written test) fall into two groups. The first group consists of those students who have to take the oral exam, or otherwise they would fail. The other group comprises very ambitious students: those who are determined to get the highest possible mark. Although both groups demand different kinds of questions, the examiner usually runs out of ideas for new, original questions quite quickly.

So it was, once, during an oral exam for *Introduction to Programming*. The exam was slowly coming to its end when another student arrived. He was not satisfied with *A*– and wanted to improve it to *A*. I decided to give him a task on dynamic programming, in which one had to calculate something for each rectangular fragment of a given rectangle. A very good example of a rectangle, and a delicious one too, is a chocolate bar. But what can one do with a chocolate bar? Of course it can be eaten, after dividing it into single pieces. The question of how many times one has to break the chocolate to divide it into single pieces is too easy—with each break, the number of pieces increases by one and the final number of pieces is known (of course, provided that no-one is eating them in the meantime!). The complexity of the problem had to be increased somehow. Breaking a piece of chocolate into smaller pieces can be associated with some cost, depending on the place where it is broken. A possible goal is to minimize the total cost. That is how the task *Chocolate* emerged.

**Solution**

When the student started describing a greedy solution to this problem, I switched on and started listening eagerly. Would he perhaps not deserve an *A* after all? The student had to prove the correctness of the greedy algorithm. This was not easy; cases and sub-cases kept multiplying all the time. Finally, we came to a common conclusion: that the correctness of the algorithm can be proven. Here is how the proof goes.

The basic observation is that it is better first to break the chocolate along the lines with the highest cost, so that the number of breaks along such lines is minimal. First, let us note that the total number of breaks does not depend on the way we break the chocolate into pieces, and always equals $n \cdot m - 1$. This fact is probably well known to all chocolate connoisseurs.



If the number of breaks does not depend on the order of breaking, then we should break the chocolate in such a way that the number of high-cost breaks is as small as possible. Intuitively, the chocolate should be broken along lines in order of their decreasing costs. The following lemma proves this intuition.

***Lemma 1.*** There exists an optimal solution in which the costs of consecutive breaks are non-increasing.

*Proof.* Let us assume that the lemma is false, and let

$$(p_1, p_2, \ldots, p_{n \cdot m-1})$$

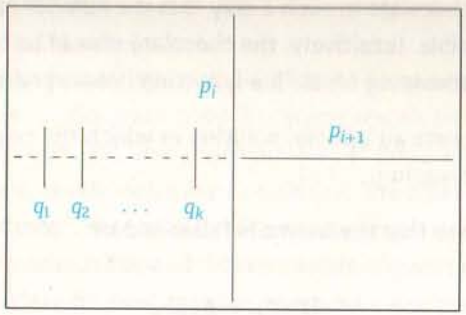
be an optimal sequence of breaks with a maximal number of inversions* of the breaks' costs. There exist two consecutive breaks, $p_i$ and $p_{i+1}$, such that the cost of $p_{i+1}$ is larger than the cost of $p_i$. If there are several such indices $i$, then let us choose the largest one. The costs of breaks $p_{i+1}, p_{i+2}, \ldots, p_{n \cdot m-1}$ form a non-increasing sequence. There are a few possible cases:

1. Break $p_{i+1}$ is located outside a piece of chocolate broken by $p_i$. In this case we can swap breaks $p_i$ and $p_{i+1}$, preserving the optimality of the solution. However, the sequence of costs of breaks

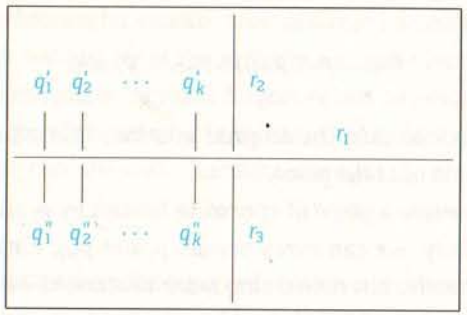
$$(p_1, \ldots, p_{i-1}, p_{i+1}, p_i, p_{i+2}, \ldots, p_{n \cdot m-1})$$

   has more inversions than the original solution. This is not possible; hence such a case could not take place.
2. Break $p_{i+1}$ lies inside a piece of chocolate broken by $p_i$ and both breaks are parallel. Similarly, we can swap breaks $p_i$ and $p_{i+1}$, obtaining an optimal sequence of breaks, but containing more inversions—a contradiction, so such a case is not possible either.
3. Break $p_{i+1}$ lies inside a piece of chocolate broken by $p_i$ and both breaks are orthogonal to each other. This situation is shown in the figure on the following page. Among the breaks to follow, there must be one or more breaks along a dashed line. The number of such breaks depends on the number of breaks $q_1, \ldots, q_k$ crossing the dashed line.

* The number of inversions of a sequence $(x_1, x_2, \ldots, x_k)$ is the number of pairs $(i, j)$ such that $i < j$ and $x_i > x_j$.




Let us consider a slightly modified way of breaking the chocolate. Let us replace breaks $p_i$, $p_{i+1}$ and breaks along the dashed line with breaks $r_1$, $r_2$ and $r_3$, as shown in the figure below. Additionally, let us replace breaks $q_1,\ldots,q_k$ with breaks $q'_1, q''_1, \ldots, q'_k, q''_k$.




How much does the total cost of breaking change? Each of the breaks $p_i, q_1, \ldots, q_k$ is replaced by two breaks of the same cost, and $k+2$ breaks with the same cost as $p_{i+1}$ are replaced by one such break. Note that the costs of breaks $q_1,\ldots,q_k$ are not larger than the cost of $p_{i+1}$. Since the cost of $p_{i+1}$ is larger than the cost of $p_i$, we obtain a solution that's better than the optimal one. This is not possible, so such a case could not happen.

The assumption that the lemma is false leads to a contradiction. Hence, the lemma is true. □



We have shown that there exists an optimal solution in which the costs of consecutive breaks form a non-increasing sequence. But does it suffice to order the costs of the breaks in such a way in order to obtain an optimal solution? The following lemma shows that it does.

***Lemma 2.*** All ways of breaking the chocolate, such that the costs of consecutive breaks form a non-increasing sequence, have the same total cost.

*Proof.* The proof goes by induction over the size of the chocolate bar. If the chocolate bar consists of just one piece, then there exists only one possible (empty) way of breaking, so the lemma holds.

Let us assume that the chocolate bar is larger. Consider all the lines of breaking with maximal cost. We start by breaking along these lines. Let us assume that there are $k$ such vertical and $l$ such horizontal lines. They divide the chocolate bar into $(k+1)\cdot(l+1)$ smaller rectangular chocolate bars.

Let us treat these smaller bars as single pieces of chocolate for a moment. This corresponds to a $(k+1)\times(l+1)$ chocolate bar with all the breaking lines having the same cost. Hence, each way of breaking such a chocolate bar has the same total cost and requires $(k+1)\cdot(l+1)-1$ breaks.

Observe that each way of breaking the whole chocolate bar, with non-increasing costs of consecutive breaks, consists of breaks with maximal costs augmented with ways of breaking the remaining smaller chocolate bars. By inductive assumption, for each of the remaining smaller chocolate bars, each way of breaking it with non-increasing costs of consecutive breaks has the same total cost. Hence, each way of breaking the whole chocolate bar, with non-increasing costs of consecutive breaks, has the same total cost. □

The following theorem is a direct consequence of Lemmas 1 and 2:

***Theorem 3.*** Breaking the chocolate by breaking along the whole chocolate bar and along lines of non-increasing costs gives an optimal way of breaking the chocolate bar.

Based on this theorem, the task can be solved using a greedy approach. First, we sort the lines of breaking according to their costs, remembering which are vertical and which horizontal. Then we break the chocolate bar (across the whole bar) along lines of non-increasing costs. We do not have to track exactly the state of the chocolate bar. It is sufficient just to remember the number of vertical and horizontal lines along which the chocolate bar has been broken. The cost of breaking across the whole chocolate bar equals the cost of breaking

along the given line multiplied by the number of orthogonal breaks (plus one) done so far. Here is pseudocode for such a solution:

```
Algorithm Chocolate((x_1, ..., x_{m-1}), (y_1, ..., y_{n-1}))
    lines := ((x_1, true), ..., (x_{m-1}, true), (y_1, false), ..., (y_{n-1}, false))
    sort lines in non-increasing order, according to the first elements of the pairs
    cost := 0
    vertical := 1; horizontal := 1
    for (w, d) in lines do
        if d then
            cost := cost + horizontal · w
            vertical := vertical + 1
        else
            cost := cost + vertical · w
            horizontal := horizontal + 1
    return cost
```

The cost of sorting dominates the time complexity of this algorithm. In the task presented at the Olympiad, costs of breaking were integers between 1 and 1000. This allows us to apply counting sort and achieve $\Theta(n+m)$ time complexity. Without this restriction, the time complexity of the solution is $\Theta((n+m)\log(n+m))$.

The presented solution is much simpler that the proof of its correctness. Also, the greedy strategy is intuitive and straightforward. No wonder, then, that many contestants have applied such a strategy without proving its correctness. The task turned out to be relatively easy—71% of contestants received the maximum number of points, and proved themselves to be smarter than the author during the described oral exam! Moreover, the aforementioned student who had to justify the correctness of the greedy strategy did indeed deserve his improved final mark of *A*.

# / Fibonacci *Sums*

**Contest:** 12th Polish Olympiad in Informatics
**Task author:** Marcin Kubica
**Memory:** 32 MB
http://main.edu.pl/en/archive/oi/12/sum

The Fibonacci numbers form an integer sequence defined in the following way: $Fib_0 = 1$, $Fib_1 = 1$, $Fib_i = Fib_{i-2} + Fib_{i-1}$ (for $i \geq 2$). The first few numbers in this sequence are: 1, 1, 2, 3, 5, 8,...

The great computer scientist Byteasar is constructing an unusual computer in which numbers are represented in a *Fibonacci system*: i.e. a bit string $(b_1, b_2, \ldots, b_n)$ denotes the number $b_1 \cdot Fib_1 + b_2 \cdot Fib_2 + \cdots + b_n \cdot Fib_n$. (Note that we do not use $Fib_0$.) Unfortunately, such a representation is ambiguous: i.e. the same number can have different representations. The number 42, for instance, can be written as: (0, 0, 0, 0, 1, 0, 0, 1), (0, 0, 0, 0, 1, 1, 1, 0) or (1, 1, 0, 1, 0, 1, 1). For this very reason, Byteasar has limited himself to using only representations that satisfy the following conditions:

- if $n > 1$ then $b_n = 1$, i.e. the representation of a number does not contain leading zeros,
- if $b_i = 1$ then $b_{i+1} = 0$ (for $i = 1, \ldots, n-1$), i.e. the representation of a number does not contain two (or more) consecutive 1s.

The construction of the computer has proved more demanding than Byteasar supposed. He has difficulties in implementing addition. Help him!

## Task

Write a program that:

- reads from the input the representations of two positive integers,
- calculates and writes to the output the representation of their sum.

## Input

The input contains the Fibonacci representations (satisfying the aforementioned conditions) of two positive integers $x$ and $y$—one in the first line, the other in the second. Each of these representations is in the form of a sequence of nonnegative integers, separated by single spaces. The first number in the line specifies the length of the representation $n$ ($1 \leq n \leq 1{,}000{,}000$). It is followed by $n$ 0s and/or 1s.

**Output**

In the first and only line of the output your program should write the Fibonacci representation (satisfying the aforementioned conditions) of the sum $x + y$. The representation should be in the form of a sequence of nonnegative integers, separated by single spaces, as described in the *Input* section. The first number in the line specifies the length of the representation $n$. It is followed by $n$ 0s and/or 1s.

**Example**

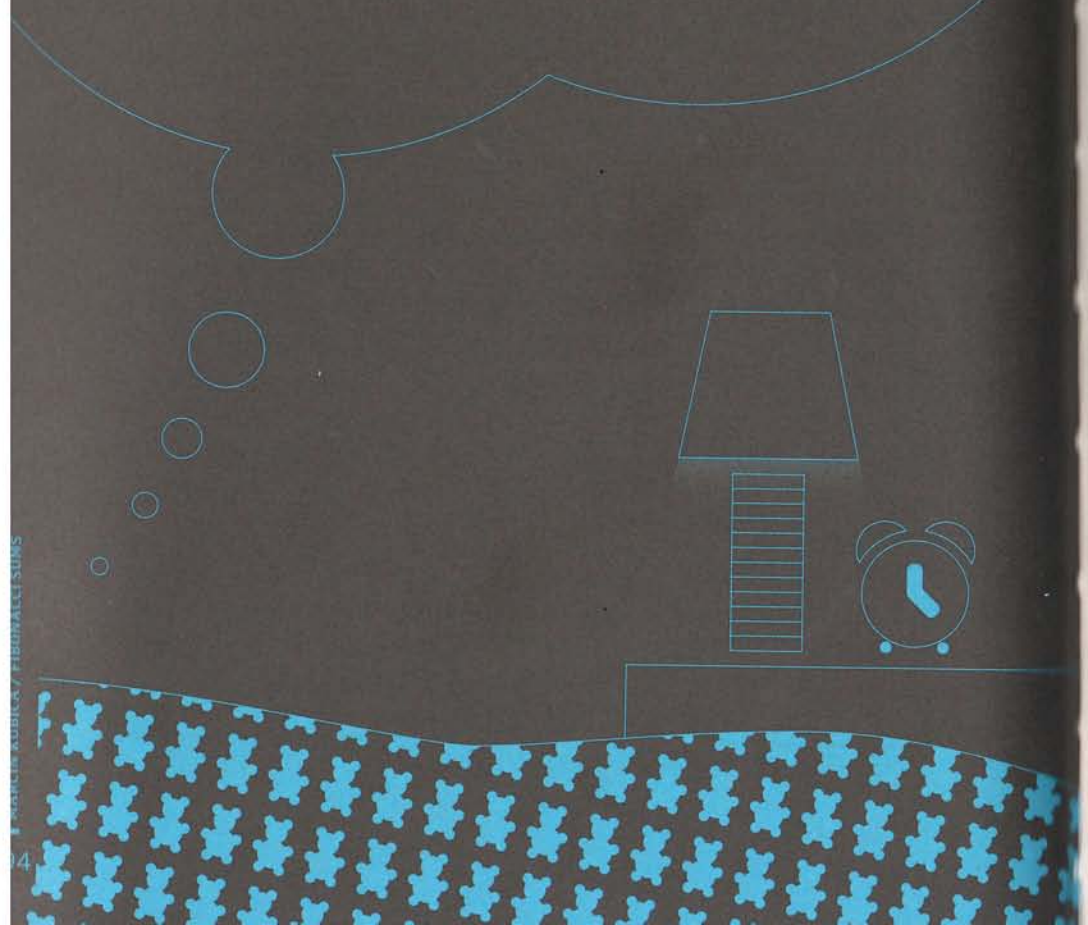
For the input data:

4 0 1 0 1

5 0 1 0 0 1

the correct result is:

6 1 0 1 0 0 1





# / Solution

When you work late at night and think up new things, they often do not seem the same in the morning. This is what happened one night before a written test on *Introduction to Programming*. Fighting against sleepiness, I had decided to contrive an easy task. What came to my mind was the addition of integers represented in Zeckendorf's system.

It turned out that this task surprised me in several ways. The first surprise was that no student managed to finish the test before the time ran out. The second surprise was that even the best students (including former champions in the International Olympiad in Informatics) complained about the difficulty of the task. But the biggest surprise came when the students inquired about the linear-time solution of this task, and I just realized that the problem was not as easy as it had seemed the night before. After a week or two, I thought I could solve the task. Unfortunately, the more inquisitive students disabused me. After two months Jacek Chrząszcz, with whom I gave an *Introduction to Programming* tutorial, helped me in finding a linear-time solution to this task. It was time to turn an initial failure into a success, and the task was used in the second stage of the 12th Polish Olympiad in Informatics.

But that was not the end of the surprises. During the preparation of this text, Tomasz Idziaszek noticed that the solution I presented could be simplified. This resulted in a simpler algorithm and a more elegant proof of its correctness than those presented in the *Blue Book*, containing analyses of tasks from the Olympiad*.

**Zeckendorf's system**

Before we see the linear-time solution of this task, let us have a look at Zeckendorf's system. It appeared in the task statement under the name of the *Fibonacci system*. This system is a mathematical curiosity and has no practical applications; the reason is that the implementation of arithmetical operations is more complicated and less efficient than in the case of the classical binary system.

* Unfortunately, the *Blue Book* is available only in Polish.

First, let us make sure that this system is an unambiguous and correct positional system; that is, each positive integer has exactly one representation in this system. Let us denote by $(b_k b_{k-1} \ldots b_1)_{Fib}$ the sum $\sum_{i=1}^{k} b_i \cdot Fib_i$. Let $x$ be any nonnegative integer. We can transform it into Zeckendorf's system in the following way:

1. If $x \leq 1$, then its representation consists of just one digit: $x_{Fib} = x$.
2. If $x > 1$, then let $Fib_k$ be the largest Fibonacci number not exceeding $x$. The representation of $x$ consists of $k$ digits and the most significant digit is $b_k = \lfloor x / Fib_k \rfloor$. The remaining digits $b_{k-1}, \ldots, b_1$ form a representation of $x - b_k \cdot Fib_k$, completed by some leading 0s if necessary.

Let us check that this algorithm is correct: that is that, for each positive integer, it gives a result satisfying the conditions just stated.

***Lemma 1.*** For any positive integer $x$, the above algorithm produces a sequence of $k$ digits 0 and 1, such that:

(a) $(b_k b_{k-1} \ldots b_1)_{Fib} = x$,
(b) $b_k = 1$,
(c) if $b_i = 1$, then $b_{i+1} = 0$ (for $i = 1, \ldots, k-1$).

*Proof.* Clearly, the representation produced by the algorithm satisfies condition (a). Proof of the other two conditions goes by induction over $x$. For $x \leq 2$ one can easily check that the lemma holds.

Assume that $x > 2$ and that the lemma holds for values smaller than $x$. The largest Fibonacci number not exceeding $x$ is $Fib_k$, so:

$$Fib_k \leq x < Fib_{k+1} = Fib_{k-1} + Fib_k,$$

and:

$$x - Fib_k < Fib_{k-1}.$$

Hence $b_k \leq 1$ (since $Fib_{k-1} < Fib_k$). From the construction of the algorithm, we know that $b_k \geq 1$. Thus we obtain condition (b). The above inequalities also imply that $b_{k-1} = 0$ and the representation of $x - Fib_k$ consists of at most $k-2$ digits. By the induction assumption we find that the sequence of digits $b_k, \ldots, b_1$ satisfies condition (c). Moreover, all the generated digits are 0s and 1s. □

Representations satisfying the conditions of Lemma 1 are called *correct*. Now we will show that representations in Zeckendorf's system are unambiguous.

***Lemma 2 (Zeckendorf).*** For each positive integer $x$, there exists exactly one correct representation of $x$ in Zeckendorf's system.

*Proof.* Let $x$ be any positive integer such that $x < Fib_{k+1}$. Its representation can have at most $k$ digits (that is, $b_i = 0$ for $i > k$). Let us count the number of correct representations of length at most $k$, denoted as $f_k$. Since $1_{Fib}$ is the only correct one-digit representation, we have $f_0 = 0$ and $f_1 = 1$.

To count the number of correct representations of length at most $k$ (for $k \geq 2$), we should add the number of correct representations of length $k$ to the number of correct representations of length at most $k-1$. If a representation $b_k, \ldots, b_1$ starts with $b_k = 1$ then, by condition (c), $b_{k-1} = 0$ and $b_{k-2}, \ldots, b_1$ can be any correct representation of length at most $k-2$ extended with an appropriate number of leading 0s or a sequence of 0s. Hence, we obtain:

$$f_k = f_{k-1} + f_{k-2} + 1.$$

It can be proven by simple induction that:

$$f_k = Fib_{k+1} - 1.$$

Indeed, for $k \leq 1$ it can be easily checked. If the above equation is true for all indices smaller than $k$, then:

$$f_k = f_{k-1} + f_{k-2} + 1 = Fib_k - 1 + Fib_{k-1} - 1 + 1 = Fib_k + Fib_{k-1} - 1 = Fib_{k+1} - 1.$$

Since every positive integer smaller than $Fib_{k+1}$ (and there are $Fib_{k+1} - 1$ such integers) has a correct representation of length at most $k$ (and there are also $Fib_{k+1} - 1$ such representations), each such integer has an unambiguous representation. □

**Simple addition algorithm**

The simplest way to add numbers in Zeckendorf's system is to add Fibonacci numbers corresponding to 1s in the representation of one number to representation of the other number. To add $Fib_i$ to representation of $x$, first we increase the $i$-th digit by 1. As a result we can obtain:

- a correct representation,
- a representation containing two or three neighboring 1s,
- a representation containing 2 surrounded by zeroes.

If the resulting representation is not correct, we apply the following transformations.

**Elimination of a single 2.** If the result contains a digit 2, we can eliminate it by applying the following identities, starting from the most significant digits:

$$x0200y_{Fib} = x1001y_{Fib} \tag{1}$$

$$x0201y_{Fib} = x1002y_{Fib} \tag{2}$$

$$x020_{Fib} = x101_{Fib} \tag{3}$$

$$x02_{Fib} = x10_{Fib} \tag{4}$$

Note that in the identity (2), digit 2 does not disappear, but is shifted to the right. However, applying the above identities from most to least significant digits, it is eliminated in one of the following steps. It is worth noting that a newly created 2 is also surrounded by 0s (or is the last digit of the representation), since 1 at this position before transformation was also surrounded by 0s (or was the last digit, respectively). The whole process requires linear time. Finally, we obtain a representation that does not contain any 2s, but may contain some neighboring 1s.

**Elimination of neighboring 1s.** We can eliminate neighboring 1s by applying the following identities, starting from the least significant digits:

$$x0\underbrace{11\ldots11}_{2l\text{ digits}}y_{Fib} = x\underbrace{10\ldots10}_{2l\text{ digits}}0y_{Fib}$$

$$x0\underbrace{11\ldots11}_{2l+1\text{ digits}}y_{Fib} = x\underbrace{10\ldots10}_{2l\text{ digits}}01y_{Fib}$$

The whole transformation takes linear time.

So, the time-cost of addition of $Fib_i$ and a number represented in Zeckendorf's system is also linear. Hence, the addition of two integers has a quadratic overall time complexity.



**Efficient addition algorithm**

An efficient algorithm for addition in Zeckendorf's system is more complicated. We start by adding corresponding digits of the two summands. As a result, we obtain a representation that:

- may contain 0s, 1s and 2s,
- permits 1s to be placed next to each other,
- allows 2s to be surrounded only by 0s.

Again, such a representation can be transformed to a correct one in two steps. In the first step we eliminate all 2s in linear time. In the second step, we can eliminate neighboring 1s as presented in the previous section.

**Elimination of 2s.** First, let us transform the representation in such a way that it satisfies the following condition:

to the left of each 2 there is 0,
to the right of each 2 there is 1 or a sequence $02y_{Fib}$. (∗)

It turns out that this can be achieved by applying identities (1)-(4) and:

$$x012y_{Fib} = x101y_{Fib} \tag{5}$$

starting from the most significant digits.

To prove the correctness of such a transformation, first let us note that initially each 2 is surrounded by 0s, so the first part of (∗) is satisfied. The only identity that can possibly violate it is (1), but that would mean that $y$ starts with a 2 and it will be eliminated in the next step by (5).

Let us focus on the most significant 2, which does not satisfy the second part of condition (∗). It is followed by $00y_{Fib}$, $01y_{Fib}$, $0_{Fib}$ or is the least significant digit of the representation. Each of these corresponds to one of the identities (1)-(4). Note that identities (1), (3) and (4) remove 2s, and (2) shifts 2 to the right. In the latter case, digit 2 will either be transformed in the next step, or $y$ starts with 1 and this 2 does not violate condition (∗).

Additionally, none of the applied identities violates the second part of the condition (∗), because if $x$ ends with 2, then it can be followed by sequences of the form $02y'_{Fib}$ or $1y'_{Fib}$. The only case left to be considered is when we apply (1) and $y$ starts with 2. But then, this 2 will be eliminated in the very next step by applying identity (5).

Now we are ready to eliminate all the remaining 2s. Scanning the representation from the least significant to the most significant digits, we apply the following identity:

$$x021y_{Fib} = x110y_{Fib} \tag{6}$$

Condition (∗) guarantees that we can apply identity (6) to the least significant 2 in the representation. Moreover, this identity preserves condition (∗)—argumentation for the case where $x$ ends with 2 is the same as above.

So, finally, all the 2s are eliminated. What remains, in obtaining a correct representation, is to apply the elimination of neighboring 1s, described above. Hence, the algorithm for the addition of two integers in Zeckendorf's system works in linear time.

**TOMASZ KULCZYŃSKI**

Tomasz Kulczyński is currently doing his M.S. degree in computer science at the Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, and participates in all kinds of programming contests. He won the International Olympiad in Informatics in 2007 and the Baltic Olympiad in Informatics in 2006. He was awarded over ten medals of different colors in Polish and international olympiads for high school students, both in computer science and mathematics. He finished in top places in TopCoder competitions (he is a five-time finalist of these competitions) as well as Google Code Jam 2006 and Facebook Hacker Cup 2011 and 2012. Together with his team, Tomasz Kulczyński has won the Polish Collegiate Programming Contest three times and the ACM-ICPC Central European Regional Contest twice. He also represented the University of Warsaw in the ACM-ICPC World Finals in 2011 and 2012 (in a team with Jakub Pachocki and Wojciech Śmietanka).

Additionally, Tomasz Kulczyński is passionate about billiards, solves lots of puzzles, and enjoys running and playing soccer.

# / The Search

**Contest:** 16th Polish Olympiad in Informatics
**Task author:** Tomasz Kulczyński
**Memory:** 32 MB
http://main.edu.pl/en/archive/oi/16/pos

Young Bytie and Byteen are in deep trouble. The evil sorcerer Bitter, knowing they are head over heels in love, has kidnapped Byteen for ransom. What Bitter did not know is that, while Bytie is determined to bring Byteen back, he is not well off and thus cannot afford the ransom. Having few options, Bytie has challenged Bitter for Byteen's return. Bitter despises all fights, but burning ambition made him propose the following solution. If the boy guesses the floor on which the girl is imprisoned, the sorcerer will let her free.

There are many floors in the sorcerer's tower—they are numbered from 1 to $n$. Bytie can only obtain information by asking Bitter questions of the form: "Is Byteen above/below floor number $x$?" Of course, Bytie can choose the number $x$ arbitrarily, as well as either of the words "above" or "below". Bitter promises to answer such queries truthfully, but asks for a reward. The sorcerer charges $a$ bytean dollars for every question with the answer YES and $b$ bytean dollars for every question with the answer NO. He also refuses to answer questions of any form other than that mentioned.

Bytie wonders what questions he should ask. Obvously, Bytie would like to spend as little as possible. To be precise: if, at any point during the conversation, it is possible to deduce that, from then on—irrespective of Bitter's answers—Bytie can guess Byteen's location by spending no more than $K$ bytean dollars, then Bytie wants to spend not more than $K$ bytean dollars in the rest of the conversation. Help Bytie find the right solution!

## Communication

You are to implement a program that solves Bytie's problem using a supplied library (which simulates the evil sorcerer Bitter).

The library supplies three functions:

- `void inicjuj(int *n, int *a, int *b)` (Polish for *init*)—returns the number of floors $n$ and the costs $a$ and $b$. It should be called once, at the very beginning of the program's execution.
- `int pytaj(char c, int x)` (Polish for *ask*)—character $c$ denotes the type of question (W for above or N for below), while $x$ is the floor number. Returns the Boolean answer to the query (0 = false, 1 = true). Your program may call this function arbitrarily many times.
- `void odpowiedz(int w)` (Polish for *answer*)—with this function you should declare the floor on which Byteen is located. It should be called exactly once. The call will terminate your program's execution.

Your program cannot open any files; nor can it use input and output.

## Constraints

You can assume that $1 \le n \le 10^9$ and $1 \le a, b \le 10{,}000$.

## Exemplary execution of a program

| FUNCTION CALL | VALUES RETURNED AND EXPLANATION |
|---|---|
| `inicjuj(&n,&a,&b);` | From now on, $n = 5$, $a = 1$, $b = 2$. |
| `pytaj('W',3);` | The answer is 0. You ask if Byteen is above the third floor. You receive the answer NO. You pay 2 bytean dollars. |
| `pytaj('N',2);` | The answer is 0. You ask if Byteen is below the second floor. You receive the answer NO, for which you pay 2 more bytean dollars. |
| `pytaj('W',2);` | The answer is 1. You ask if Byteen is above the second floor. You receive the answer YES, for which you pay 1 bytean dollar. |
| `odpowiedz(3);` | You guess that Byteen is on the third floor. Your guess is correct. You spend 5 bytean dollars in total. |

The above interaction is correct but not optimal. Hence, the program would not receive any points for this test. In particular, for $n = 5$, $a = 1$, $b = 2$, a well-written program can ask questions in such a way that it never pays more than 4 bytean dollars.

# Solution

I decided to write about this particular task—which I invented—because of a very smart trick used in the model solution. The idea is to reverse the question from the task description. This task was created during the International Olympiad in Informatics in 2008, as Jakub Radoszewski and I coached the Polish team. Our long discussions during warm Egyptian nights resulted in a few tasks, one of which is *The Search*.

**Similarity to binary search**

The description of this task reminds us of binary search. This association is natural, as binary search is the optimal way to solve the task if the cost for each question is equal to 1. It is also optimal when $a = b$, with the cost of finding Byteen equal to $a \cdot \lceil \log n \rceil$. In the general case, binary search has a pessimistic cost of $\max(a, b) \cdot \lceil \log n \rceil$. We can clearly see that, for $a \neq b$ and large values of $n$, this is not optimal. One should make queries about a point depending on $a$ and $b$, rather than the center of the interval, to compensate for the difference between the two query costs. It is not easy to determine this point, which is why we will use dynamic programming for this purpose.

**First solution**

The first solution tries to solve the problem directly by computing, for each positive integer $k$, the minimal worst-case cost of finding Byteen in an interval consisting of $k$ floors. Clearly, this cost does not depend on where the floors are located within the whole tower but only on their number $k$. From now on we will denote this cost by $t[k]$.

We will compute the values $t[k]$ in order of increasing $k$ via dynamic programming. Obviously, $t[1] = 0$. If $k > 1$, we must pose at least one question to find Byteen. Assume that the first question asks if Byteen is located below the floor $p + 1$ among those available ($1 \leq p < k$). If Bitter's answer is positive, Bytie will spend $a$ for asking the question and will continue his search with an interval consisting of the first $p$ floors. In the opposite case, Bytie will spend $b$ and the new interval's length will be $k - p$. In both cases, we used all the information we could obtain from a single answer from Bitter to draw a conclusion about Byteen's location.



Assume that, in the rest of the conversation, Bytie will ask questions in a way that minimizes the total cost. Then, in the case of a positive answer to the first question, Bytie will spend $a + t[p]$ in total; otherwise, his total cost will be $b + t[k - p]$. The pessimistic cost is the maximum of the two values.

We should now realize that the questions:

- "Is Byteen located above floor $p$?"
- "Is Byteen located below floor $k - p + 1$?"

do not differ from each other, taking into consideration the lengths of intervals that we get after Bitter's answer (equal to $k - p$ and $p$) and the costs of the first question ($a$ and $b$ respectively). That is why it is sufficient to consider one of the types of questions. To compute $t[k]$ we only have to choose the value of $p$ such that the cost of finding Byteen with the first question, as stated above, is minimal. We have:

$$t[k] = \min_{1 \leq p < k} \{\max(a + t[p],\ b + t[k - p])\}.$$

We will also remember the value of $p$ that minimizes the cost and call it $p[k]$, so that:

$$t[k] = \max(a + t[p[k]],\ b + t[k - p[k]]).$$

This way we can find all the values of $t[k]$ and $p[k]$ for $k = 1, 2, \ldots, n$ in $O(n^2)$ time with space usage of $O(n)$. After this precomputation the process of asking questions is easy to implement: at each step, if we have an interval that consists of $k$ floors (with $k > 1$), the next question Bytie should ask is: "Is Byteen located below floor $p[k] + 1$ in this interval?" The total time of the conversation phase is $O(n)$.

**Model solution**

Let us reverse the problem. Instead of finding the optimal costs of locating Byteen in an interval of floors, for each cost $c$ let us find the maximum number of floors in an interval in which Byteen can be located at the cost of at most $c$. We will denote this number of floors by $Q[c]$.

First, we observe that $Q[0] = 1$. For simplicity we assume that for $c < 0$, we have $Q[c] = -\infty$. From now on we consider $c > 0$. Assume that we have already

limited the search to an interval of $Q[c]$ floors and want to ask the next question. The answer we get splits the interval into two shorter intervals. Now we need to find Byteen in one of them with a maximum cost of $c-a$ or $c-b$ respectively. Those two intervals should then have lengths of at most $Q[c-a]$ and $Q[c-b]$ respectively. This leads to the formula

$$Q[c] = \max(Q[c-a] + Q[c-b], 1)$$

from which $Q[c]$ can be computed via dynamic programming.

To ask the right question, we have to find the smallest cost $w$ such that $Q[w-1] < n \leq Q[w]$. This is the cost that we need to spend in a pessimistic case. Let us denote $n_0 = n$, $w_0 = w$. Each question will be asked so as to split the interval of length

$$n_i \leq Q[w_i] = Q[w_i - a] + Q[w_i - b]$$

into two intervals of length $Q[w_i - a]$ (which we will consider if the answer is positive) and $n_i - Q[w_i - a] \leq Q[w_i - b]$ (in the opposite case). Then we either spend $a$ and search in an interval of length $n_{i+1} = Q[w_i - a]$ or spend $b$ and search in an interval of length $n_{i+1} \leq Q[w_i - b]$. In both cases we will have a total cost of at most $w_i$.

However, we need to remember that, in the case of a negative answer, $w' = w_i - b$ is not necessarily the smallest $w'$ such that $Q[w'] \geq n_{i+1}$. It is possible that $w_{i+1}$ is actually smaller:

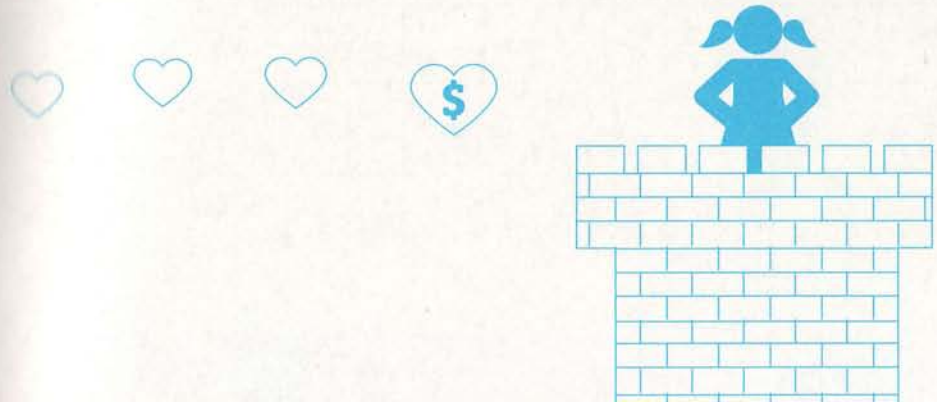
$$w_{i+1} = \min\{j : j \leq w' \wedge Q[j] \geq n_{i+1}\}.$$

The necessity to remember about this comes from the requirement to save as much as we can at each step.

This solution has time complexity $O(w)$. First, for each $i \leq w$ we have to compute $Q[i]$, in order of increasing $i$, to find $w$ itself. Then we have to go through the values once more while asking questions in order of decreasing $i$.

```
Algorithm Search
    init(n, a, b) {inicjuj}
    Q[0] := 1
    w := 0
    while Q[w] < n do
        w := w + 1
        if w < max(a, b) then
            Q[w] := 1
        else
            Q[w] := Q[w − a] + Q[w − b]
    down := 1; up := n
    while down < up do
        while Q[w − 1] ≥ up − down + 1 do
            w := w − 1
        if ask('N', down + Q[w − a]) then {pytaj}
            up := down + Q[w − a] − 1
            w := w − a
        else
            down := down + Q[w − a]
            w := w − b
    answer(down) {odpowiedz}
```

To estimate the complexity of this solution properly we still need to know how big $w$ can get. We know that $w$ is the cost of the optimal strategy, so it is not greater than the cost of any strategy and, in particular, not greater than the cost of binary search. We have already stated that this cost does not exceed $\max(a,b) \cdot \lceil \log n \rceil$. This estimate is not a bad one, because for $a = b$ we have exactly $w = \max(a,b) \cdot \lceil \log n \rceil$. The time and space complexity is therefore $O(\log n \cdot \max(a,b))$.



**Results**

The tables below show the values of $t$, $p$ and $Q$ from both solutions for $n = 10$ and several different pairs of $a$, $b$.

→ $a = 1$, $b = 2$:

| $i =$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $t[i] =$ | 0 | 0 | 2 | 3 | 4 | 4 | 5 | 5 | 5 | 6 | 6 |
| $p[i] =$ | - | - | 1 | 2 | 2 | 3 | 3 | 4 | 5 | 4 | 5 |

| $i =$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $Q[i] =$ | 1 | 1 | 2 | 3 | 5 | 8 | 13 |

→ $a = 3$, $b = 10$:

| $i =$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $t[i] =$ | 0 | 0 | 10 | 13 | 16 | 19 | 20 | 22 | 23 | 23 | 25 |
| $p[i] =$ | - | - | 1 | 2 | 3 | 4 | 4 | 5 | 5 | 6 | 7 |

| $i =$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $Q[i] =$ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 |
| $i =$ | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |
| $Q[i] =$ | 3 | 3 | 3 | 4 | 4 | 4 | 5 | 6 | 6 | 7 | 9 | 9 | 10 |

→ $a = 5$, $b = 7$:

| $i =$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $t[i] =$ | 0 | 0 | 7 | 12 | 14 | 17 | 19 | 19 | 21 | 22 | 24 |
| $p[i] =$ | - | - | 1 | 2 | 2 | 3 | 3 | 4 | 4 | 5 | 5 |

| $i =$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $Q[i] =$ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 3 |
| $i =$ | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | |
| $Q[i] =$ | 3 | 4 | 4 | 4 | 5 | 5 | 7 | 7 | 8 | 9 | 9 | 12 | |

→ $a = 6$, $b = 6$:

| $i =$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $t[i] =$ | 0 | 0 | 6 | 12 | 12 | 18 | 18 | 18 | 18 | 24 | 24 |
| $p[i] =$ | - | - | 1 | 1 | 2 | 1 | 2 | 3 | 4 | 1 | 2 |

| $i =$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $Q[i] =$ | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 4 |
| $i =$ | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | |
| $Q[i] =$ | 4 | 4 | 4 | 4 | 4 | 8 | 8 | 8 | 8 | 8 | 8 | 16 | |



# / Hashing

**Contest:** Polish Olympiad in Informatics Training Camp 2008
**Task author:** Jakub Pawlewicz
**Memory:** 32 MB
http://main.edu.pl/en/archive/ontak/2008/has

Participants in the POI Training Camp have put Byteasar in a difficult position. Instead of turning in solutions based on a beautiful and natural application of the Karp-Miller-Rosenberg algorithm, they submitted solutions based on hashing. In principle, the hashing-based approach is incorrect due to the possibility of hash function collisions. Byteasar has decided to construct test cases that will invalidate such submitted solutions. To this end, he has to find a pair of words with the same hash value.

A word is defined as a non-empty string consisting of at most 20 lowercase letters of the English alphabet. The value of the hash function $h$ on a word is a number from the set $\{0, 1, \ldots, n-1\}$. To a very good approximation, the probability that two different words $x$ and $y$ have the same hash value is $\frac{1}{n}$. Your task is the following: given a hash function with the above properties, demonstrate two different words $x$ and $y$ with the same hash value.

## Hash function

The hash function used in this problem is available through a C/C++ library and takes two arguments: $n$, indicating the range of values returned by hash(n,x), and $x$, indicating the input word (non-empty, at most 20 characters long).

This function has the following prototype:

```
unsigned long long hash(unsigned long long n, const char *x);
```

## Task

Write a program that:

→ reads in $n$ from the input,
→ finds two words $x$ and $y$ such that hash(n,x) = hash(n,y),
→ outputs the answer.

## Input

The input describes one input instance and consists of one line, containing a positive integer $n$.

**Output**

The output consists of two lines, each containing exactly one string of at most 20 lowercase English letters, such that hash returns the same value for both words.

**Example**

For the input data:

10000

the correct result is:

bgvhfqbfenb

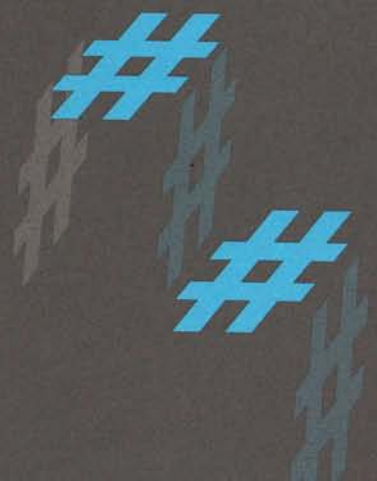
xeonduwaxi

**Tests**

Test cases that will be used to test your submission are made available in advance. In effect, your solution may consist of a program with hardcoded answers that are returned after the program reads and identifies the input $n$.

**Comments and remarks**

Note that the task *is not* to find the hash function or its values. Additionally, hash values are unspecified for values of $n$ that are not in the available test files and for strings that do not satisfy the specifications.



# / Solution

The task *Hashing* appeared in the Polish Olympiad in Informatics Training Camp in 2008. I had the pleasure of being a jury member at that camp, and this task drew my attention at once. The problem seems quite natural: it is well known that algorithms that use hashing are inaccurate, so why not try to find sample data for which such an algorithm fails?

The participants of the camp had many different ideas as to how to solve the problem. Some of them tried to decompile binary files of the library containing the *hash* method and use its implementation details to solve the inputs one by one. Of course, that was not the intended approach.

We will try to solve the task without any additional assumptions about how the hash function is implemented. First of all, it is easy to observe that it is sufficient to check any $n + 1$ different strings, because some two of them will have the same hash values (by the pigeonhole principle). This yields a simple solution with $O(n)$ complexity, which is not satisfactory—but in practice we can expect it to run much faster, i.e. it is likely to find the requested pair long before all the $n + 1$ strings have been checked. The only information we have about the hash function is a probabilistic fact, so it naturally comes to mind to choose subsequent strings randomly until a collision is found. The following pseudocode implements this simple approach.

```
S := ∅
while true do
    a := random_word()
    x := hash(n, a)
    if (x, b) ∈ S for some b ≠ a then
        return (a, b)
    S := S ∪ {(x, a)}
```

Denote by $w$ the number of different strings that we need to examine before we find the result. The above algorithm performs $O(w \log w)$ operations and uses $O(w)$ memory. We still have to think about how big $w$ can get.

**Lemma.** The probability of a collision within $k = \lceil\sqrt{n}\rceil + 1$ random strings is greater than $\frac{1}{3}$.

*Proof.* We will try to find an upper estimate for the probability of the lack of collisions. Certainly, the first string we draw does not create any collisions. The second string has a different hash value with the probability $\frac{n-1}{n}$. Similarly, the $i$-th string drawn does not create a collision with any previous string with probability $\frac{n+1-i}{n}$. The probability that there will be no collisions at all equals:

$$P = \frac{n}{n} \cdot \frac{n-1}{n} \cdots \frac{n-k+1}{n}.$$

Using a well known inequality $1 + x \le e^x$ we estimate that value as follows:

$$P = \left(1 - \frac{1}{n}\right)\left(1 - \frac{2}{n}\right) \cdots \left(1 - \frac{k-1}{n}\right) \le e^{-\frac{1}{n}} e^{-\frac{2}{n}} \cdots e^{-\frac{k-1}{n}} = e^{-\frac{k(k-1)}{2n}} \le e^{-\frac{1}{2}} < \frac{2}{3}. \qquad \square$$

From the lemma it follows with high probability that $w$ is of the order of $O(\sqrt{n})$. E.g. $w > 12 \cdot (\lceil\sqrt{n}\rceil + 1)$ with a probability of less than $1 - \left(\frac{2}{3}\right)^{12} < 0{,}01$. Therefore we can expect that the above solution will perform $O(\sqrt{n} \log n)$ operations with a memory usage of $O(\sqrt{n})$.

The expected running time of the solution seems satisfactory, but the space complexity of $O(\sqrt{n})$ for $n$ up to $10^{18}$ might be problematic. We will now try to reduce the space complexity. For this, we must avoid storing strings that we have already drawn. Let us start by selecting a new function $g(x)$, which maps integers from $H = \{0, 1, \ldots, n-1\}$ to different strings. This function can be chosen arbitrarily. It is important, though, for us to be able to compute its values quickly. Let $f$ be the composition of $g$ and *hash*: $f(x) = hash(n, g(x))$. We know that $f$ maps $H$ to $H$ and has the following property: $f(x) = f(y)$ with $x \ne y$ if and only if $g(x)$ and $g(y)$ are different strings having the same hash value. Now the task reduces to finding two distinct integers $x$ and $y$ such that $f(x) = f(y)$. Note that we can assume that, for each $x$, $f(x)$ is a uniformly random value from the set $H$, because *hash* has that property and $g$ was chosen independently of the function *hash*. Because of that, for every string $s_0$ and $x_0 = hash(n, s_0)$, the sequence

$$x_0, \quad x_1 = f(x_0), \quad x_2 = f(f(x_0)), \ \ldots, \ x_i = f^{(i)}(x_0), \ldots \qquad (1)$$

behaves as a random sequence of values from $H$. As before, we expect a collision among the first $O(\sqrt{n})$ elements of the sequence. Depending on the value of $x_0$, the sequence will have one of the forms depicted in figure 1.

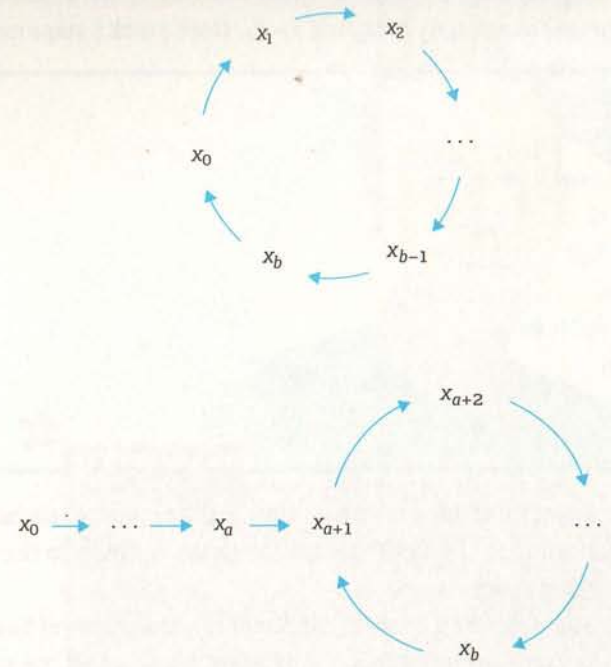



***Figure 1:*** *Possible forms of the sequence (1).*

In the first case, we should output $s_0$ and $g(x_b)$, and in the second case the output is $g(x_a)$ and $g(x_b)$. To avoid considering both cases, we can introduce $x_{-1} \notin H$ such that $g(x_{-1}) = s_0$ and $f(x_{-1}) = x_0$. Then in both cases the answer is $g(x_a)$, $g(x_b)$ (with $a = -1$ in the first case). From now on, we only have to find indexes $a$ and $b$. How can this be done without storing the sequence $(x_i)$?

Let $d$ be the length of the cycle, $d = b - a$. We first take two pointers, $p$ and $q$, and make them both point to $x_0$. We then take consecutive steps, and within each step $p$ goes one element forward and $q$ goes two elements forward (which is equivalent to the assignments $p := f(p)$ and $q := f(f(q))$). We keep performing such steps until $p$ and $q$ become equal. This will happen at some point, since, when both pointers are on the cycle, the distance between them increases by one (modulo $d$) with each step. Let $k$ be the total number of steps. We know that $q$ has made a full cycle several times more than $p$.