Looking for a Challenge?

The Ultimate Problem Set from the University of Warsaw Programming Competitions

A preview—full version at www.lookingforachallengethebook.com

Looking for a Challenge?

The Ultimate Problem Set from the University of Warsaw Programming Competitions

A preview—get the full version at
www.lookingforachallengethebook.com

Authors of the texts in this preview
Marcin Andrychowicz, Marek Cygan, Tomasz Czajka, Paweł Parys, Jakub Pawlewicz

Authors of the texts in the full book
Szymon Acedański, Marcin Andrychowicz, Piotr Chrząstowski, Marek Cygan, Tomasz Czajka, Krzysztof Diks, Andrzej Gąsienica-Samek, Tomasz Idziaszek, Grzegorz Jakacki, Tomasz Kociumaka, Eryk Kopczyński, Marcin Kubica, Tomasz Kulczyński, Jakub Łącki, Krzysztof Onak, Jakub Pachocki, Paweł Parys, Jakub Pawlewicz, Marcin Pilipczuk, Michał Pilipczuk, Jakub Radoszewski, Wojciech Rytter, Krzysztof Stencel, Wojciech Śmietanka, Tomasz Waleń, Jakub Wojtaszczyk, Filip Wolski

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

Warsaw 2012

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 Lotos Poligrafia Sp. z o.o.

/ 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 self-development, self-discipline, hunger for knowledge, self-improvement, teamwork, honesty, ambition, eagerness to compete, striving for success. 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 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. During the other 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 centralized 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 solution 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 the scientific and private interests of the authors.

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 >

/ Fishes

Contest: Algorithmic Engagements 2009
Task author: Eryk Kopczyński
Solution description: Marcin Andrychowicz
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 ($3 \le w \le 1000$, $3 \le h \le 1000$), 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 ($2 \le n \le 1000$) 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 which connects the centers of the cells along its route. However, we do not know its speed. The fish can speed up or slow down in order always to 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
```

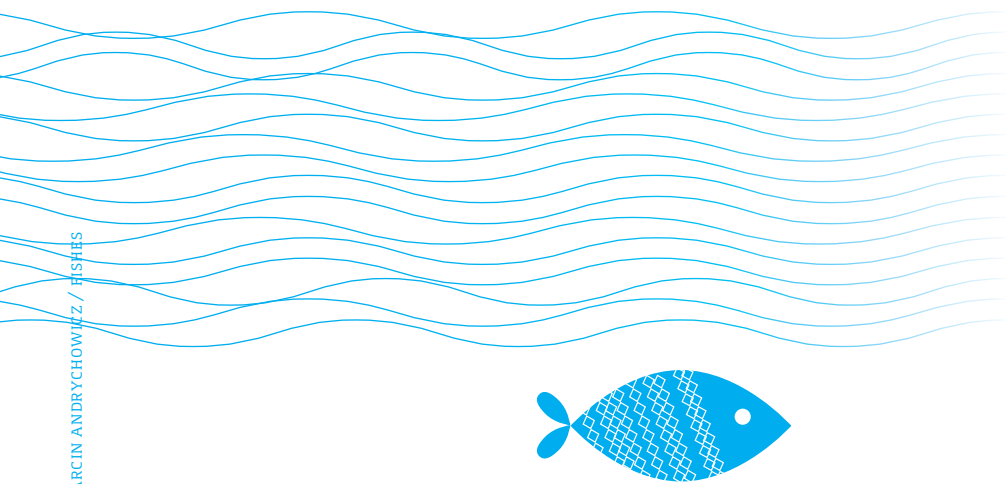
Explanation of the example: The first two routes could have been traveled by the same fish (even in two consecutive days). After a few days, the fish could have followed the third route. However, the last route must have been traveled by another fish. Unlike the first three routes, it goes around the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



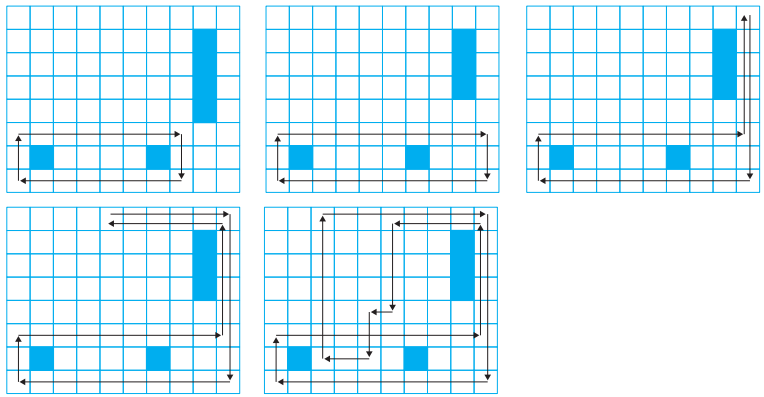


Figure 1: *An exemplary route that could be swum by one fish over consecutive days. The place where the fish wakes up is not marked, because it can be disregarded.*

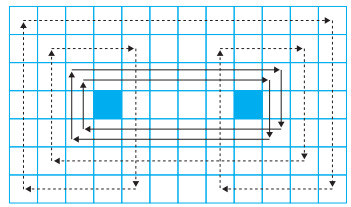


Figure 2: *Both routes circumnavigate both islands twice. Despite that, they are not equivalent.*

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.

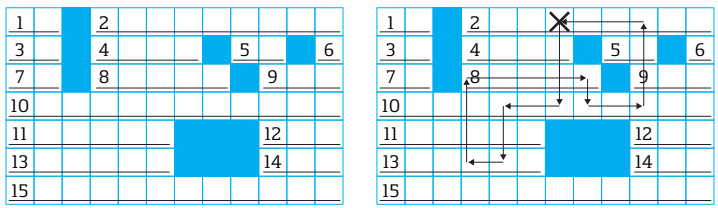


Figure 3: *Left: an archipelago with marked laser beams, which are additionally numbered with natural numbers. Right: an exemplary fish's route for which we find a sequence of intersections. A cell in which the fish wakes up is marked by a cross.*

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 which 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 the 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 working 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 which 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 $0 \leq r, t \leq 1$:

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

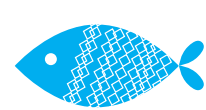
Two loops are called *homotopic* if there exists a homotopy between them.

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.



/ Barricades

Contest: Algorithmic Engagements 2007
Task author: Marek Turski
Solution description: Marek Cygan
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.

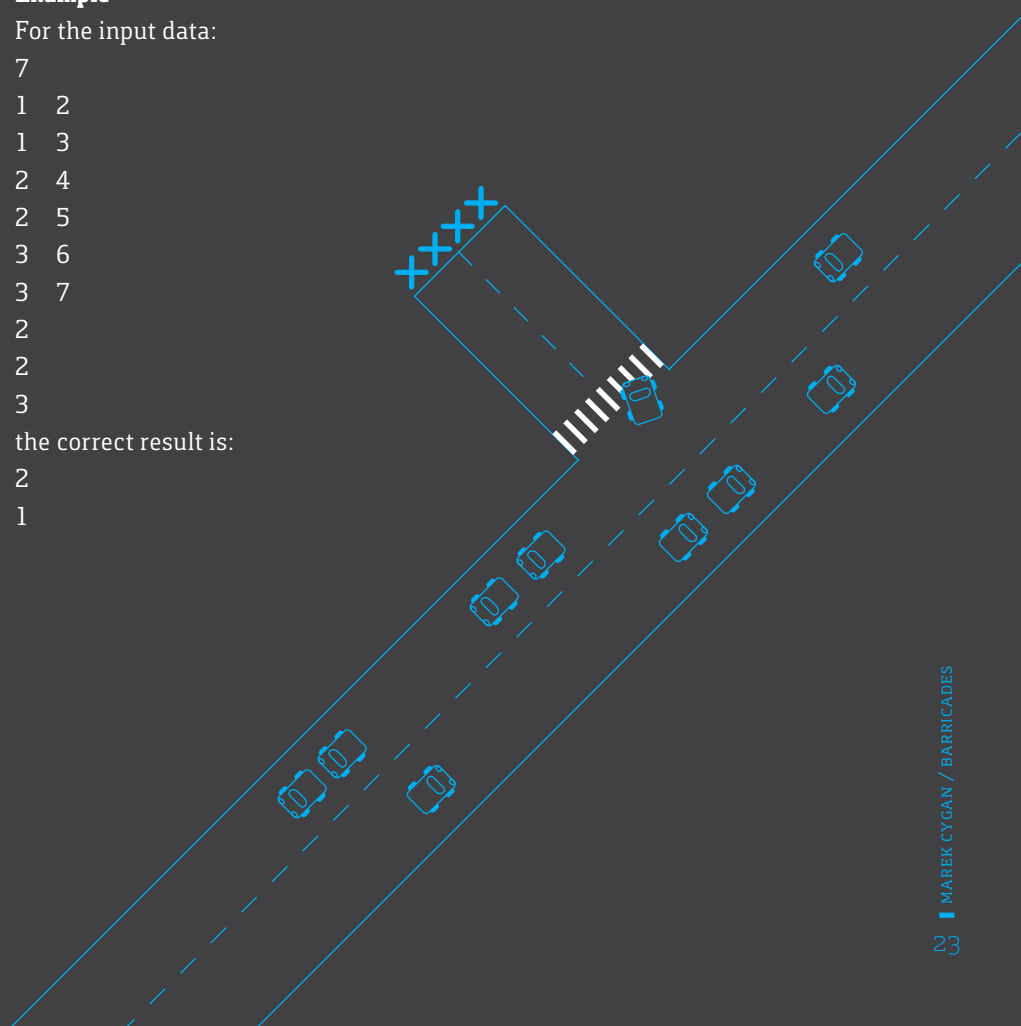
Example

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 $t[v][i] = t_k[v][i]$.

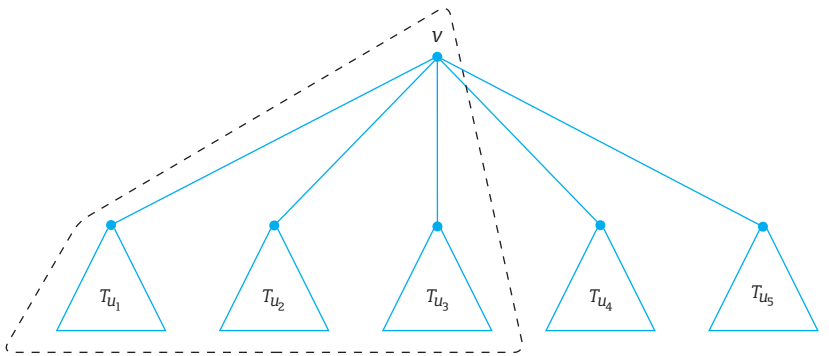


Figure 1: *The area surrounded by the dashed frame contains the vertices of the tree* T_v^3.

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:

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

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

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

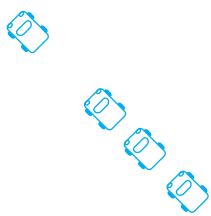
operations. From the inductive assumption, computing all the values in the table t for vertices below v takes $O(\sum_{j=1}^{k} a_j^2)$ time. Summing up, the number of operations needed to compute the values $t[u][\cdot]$ for all the vertices u belonging to T_v is

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

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.



/ Sweets

Contest: Algorithmic Engagements 2010
Task author: Jakub Łącki
Solution description: Tomasz Czajka
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 less 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 ($3 \le n \le 24$), specifying the number of boxes. The second line consists of n positive integers a_i ($1 \le a_i \le 1{,}000{,}000{,}000$), 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 2.8×10^{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 increasingly by A_1 − B_1
    sort L decreasingly 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 := ∞
    for aX ⊆ {a_1, ..., a_n}, |aX| ≤ n/3 do
        X := ∑aX
        (Y, Z) := Partition({a_1, ..., a_n} \ aX)
        r := min(r, max(X, Y, Z) − min(X, Y, Z))
    return r
```

It remains 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

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

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

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

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

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

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

$$= O\left(n^{-1/2}e^{0}3^{n+1}2^{-(2n/3+1/2)}\right) = O\left(n^{-1/2}3^{n}2^{-2n/3}\right),$$

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

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 $A \geq D$, $D \geq B$ and minimizes the value of $A - B$. It is easy to verify that these correspond to $x_1 \geq x_2$, $y_1 \geq y_2$, 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 $x' \geq x$ and $x + y \leq x' + y'$. 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 $x' \leq x$, and at the same time it will also be the point with the largest $(x' + y')$.

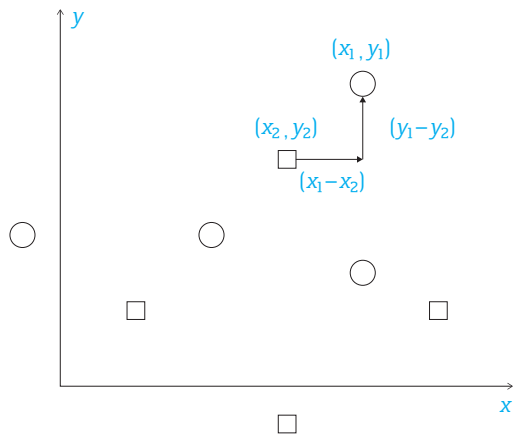
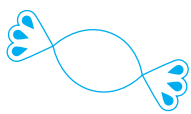


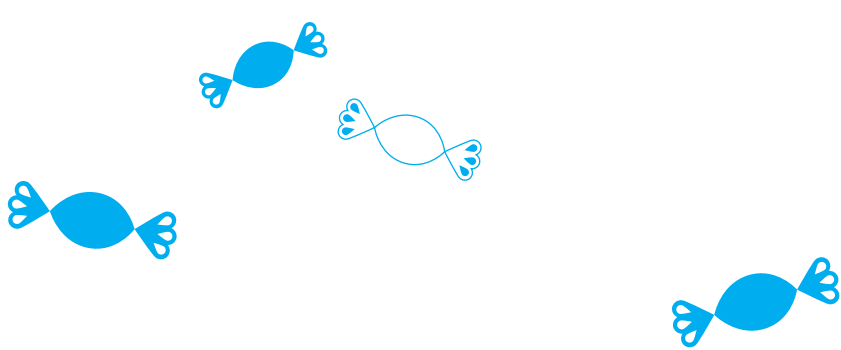
Figure 1: *The circles are elements of K', while the squares are elements of L'.*



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 the second coordinate y, then by the first coordinate 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

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



/ Cave

Contest: 11th Polish Olympiad in Informatics
Task author: Paweł Parys
Solution description: Paweł Parys
Memory: 16 MB
http://main.edu.pl/en/archive/oi/11/jas

There is a cave in Byteland. It consists of n chambers and corridors connecting them. The corridors are arranged in such a way that there is a unique path between each pair of chambers. In one of these chambers Hansel has hidden a treasure, but he won't say which chamber it is. Gretel wants to know where the treasure is, so she asks Hansel about various chambers. When she guesses right, Hansel tells her she is right, and when she guesses wrong, he tells her which way from this chamber leads to the treasure.

Task

Write a program that:

- reads from the input the description of the cave,
- finds the minimum number of questions Gretel has to ask, in the worst case, in order to learn in which chamber the treasure is hidden,
- writes the result to the output.

Input

In the first line of the input there is one positive integer n ($1 \le n \le 50{,}000$), specifying the number of chambers in the cave. The chambers are numbered from 1 to n. In the following $n - 1$ lines the corridors linking the chambers are described, one per line. Each of these lines contains a pair of distinct positive integers, a and b ($1 \le a, b \le n$), separated by a single space, indicating that there is a corridor between chambers a and b.

Output

Your program should write one integer to the output, stating the minimum number of questions Gretel has to ask in the worst case (i.e. we assume Gretel asks the questions in the best possible way, but the treasure is hidden in the chamber that requires the largest number of questions).

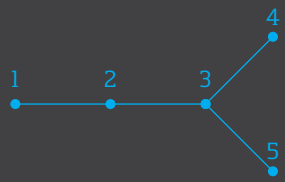
Example

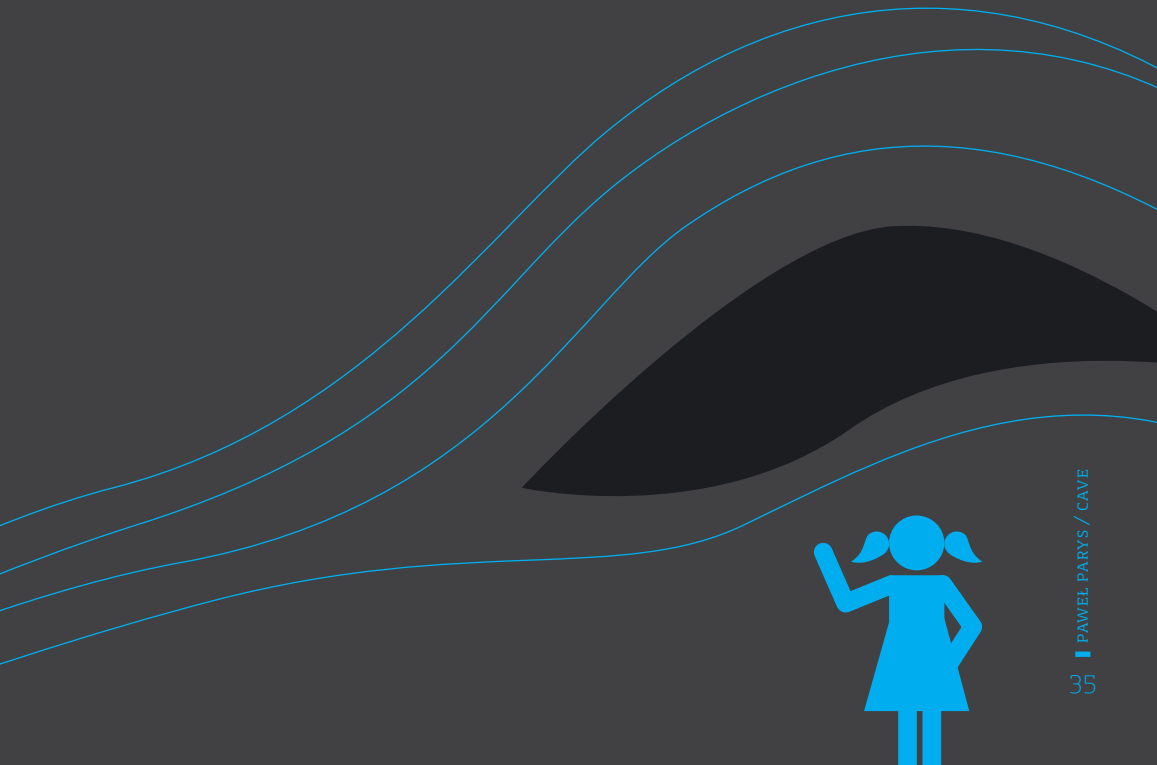
For the input data:

5
1 2
2 3
4 3
5 3

the correct result is:

2





/ Solution

This problem turned out to be one of the most difficult in the history of the Polish Olympiad in Informatics. The best candidate's solution received just 52 points (out of 100), and only a few contestants achieved more than 10 points. This was a great surprise to me (as the author of the problem). My feeling was that it would be a relatively standard problem; the line of reasoning described in the following text appeared to me completely natural—in fact, the only approach possible—and every one of its steps not too difficult. But the contestants, in the limited time available during the competition, proved unable to deal with this problem.

After the contest, I wrote a paper together with Krzysztof Onak based on this problem. The problem is very natural: a generalization of binary search into trees. We have added another variant, where the questions are asked in terms of edges instead of nodes, and a few other small extensions. Our paper was accepted by an American conference: IEEE Symposium on Foundations of Computer Science (FOCS).

Strategy functions

So, how should this problem be solved? Of course, the plan of the cave is a tree. But how can we describe a strategy for asking questions? We have to determine only the minimum number of questions to ask, but without considering a strategy for asking questions, we will probably not manage to do that. So let us think about how we might ask questions in our problem. When asking about some node of the cave, we split the cave into some number of parts (connected components), which arise when this node is removed. As our answer we get one of these parts, and in it we have to find the treasure. It makes no sense to ask any of the subsequent questions about anything outside this part, because the answer to such a question would not give us any new information; afterwards we will be asking questions only about nodes inside this part. Nevertheless, we have to consider all of these parts, because any answer is possible. So we can think about this as follows: In the tree, we remove one selected node. In the next step we again choose one node, in each of the resulting parts (connected components), and we remove these nodes. We repeat this until the parts contain only one node each. In the problem, our goal is to minimize the number of such steps.

Based on this observation, we will be describing a strategy using a function f which assigns a nonnegative integer to each tree node. This number will

tell us in which step, counting from the end, the node will be removed. Such a function has to satisfy the following condition:

for every two nodes u, v such that $f(u) = f(v)$,
on the path between u and v there is a node w such that $f(w) > f(u)$. (∗)

This condition corresponds to the requirement that if we remove two nodes in the same step, some of the nodes between them would have had to be removed earlier. Such a function will be called a *strategy function*. We see that every reasonable strategy of asking questions (i.e. such that we do not ask questions that would not provide us with any new information) can be described using a strategy function, where the number of questions asked (for the worst sequence of answers) is equal to the maximum value of the function. For example, in figure 1 we present an optimal strategy function for the tree from the problem statement, and the corresponding decision tree. But of course this correspondence also goes in the other direction: having a strategy function f, it is easy to give a strategy that requires no more than $\max_v f(v)$ questions. We just ask about the node v in which $f(v)$ is maximal. As an answer, we get some part of the tree. Again we ask about the node v in which $f(v)$ is maximal in this part of the tree, and so on. In the current part of the tree, we always have exactly one node with maximal $f(v)$; this follows from the condition (∗) above. Either we will hit the node with the treasure at some point or, at the end, we will get a tree consisting of one node and we will know that the treasure is in that node. To conclude, it is sufficient to find a strategy function f for which $\max_v f(v)$ is as small as possible.

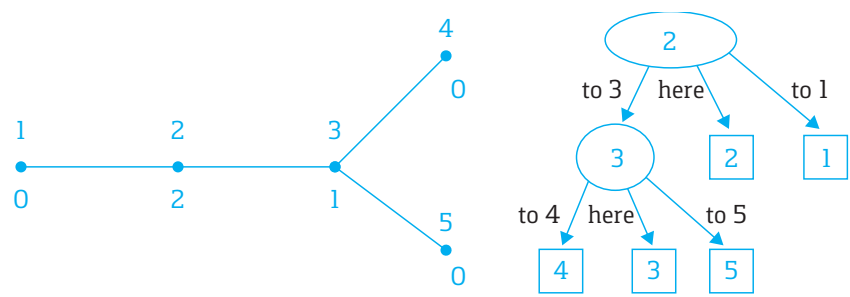


Figure 1: *Left: an optimal strategy function for the tree from the problem statement. Node numbers appear above the nodes; below them appear the values of the strategy function. Right: the corresponding decision tree. Inside the nodes appear Gretel's questions; on the edges, Hansel's answers.*

Idea of the solution

How should we find the best strategy function? A naïve approach is to begin by placing the greatest value of the function (the first question), and then consecutively smaller and smaller values. We see that the greatest value has to be assigned approximately in the middle of the cave. But it is not clear in which sense it should be the middle. We see also that it is better to choose a node which has more neighbors, since then we will divide the tree into a greater number of smaller parts. We can start in a node from which the distance to all other nodes is minimal; or in such a way that the emerging parts have the smallest number of nodes. But it is easy to ascertain that any such easy approach will not succeed. The problem is that our "size" of a tree should not be equal to the number of nodes or a diameter, but to the required number of questions; the first question should be asked in such a way that the "sizes" of the resulting parts, defined in this way, are minimal. We could start from smaller parts of the tree and, based on their results, analyze the larger ones; but the total number of parts of the tree might be very big, so we cannot process all of them in a reasonable time.

Another possible approach is to process the tree starting from the leaves and going into its center. In order to simplify the argumentation, let us choose any node of the tree as the root. This is done simply to fix the order in which we will traverse the nodes of the tree while calculating the solution: we will be considering the tree starting from the leaves and going towards the root; of course the choice of the root will not have any impact on the result. It turns out that the optimal strategy function can be obtained through a greedy approach: to every node, going from leaves to root, we assign the smallest possible strategy function value that does not conflict with the values already assigned in the descendants of that node.

It remains to justify that this approach gives us the correct result: that the maximal value of the strategy function created in this way is the smallest possible. We want to say: if, in the subtrees below a given node v, there are the best possible strategy functions, and to the node v we assign the smallest possible non-conflicting value, we will obtain the best possible strategy function for the whole subtree rooted in v.

Visibility sequences

It is easy to see that, in such an approach, a best possible strategy function is not only one whose maximal value is the smallest possible. A situation where the maximal value is at the root of a subtree is much better than a situation

where this value is somewhere deeper. In the first case, above the subtree we can assign arbitrary values different from this maximal one, what is not true in the second case. We ascertain that all values "visible" from the root of a subtree are important. The value $f(v)$ of a given node v is called *visible* in a given subtree if all nodes w on the path from v to the root (excluding v, but including the root) satisfy $f(w) < f(v)$. In other words, any greater value does not cover the value $f(v)$ when we are looking from the subtree's root. The *visibility sequence* of a given subtree is the sequence of all values visible in it, in descending order. Notice that every value visible in a subtree can be visible in only one node (otherwise the condition $(*)$ would not be satisfied for the two nodes in which this value is visible). We see also that the arrangement of the other values is unimportant: if the visibility sequence remains the same, the values in the subtree can be rearranged arbitrarily, and it will not cause any trouble from the point of view of the rest of the tree.

We want to prove that, when we greedily assign the smallest possible values of the strategy function in order from leaves to the root, in each subtree we will obtain the best possible visibility sequence. So which visibility sequence is the best possible (i.e. allowing the greatest capabilities in the rest of the tree)? First of all, the first (largest) value of the sequence counts, as this is the maximum of the strategy function in the subtree. But, as we have already said, the remaining values of the sequence also count. It turns out that visibility sequences have to be compared lexicographically: a lexicographically smaller sequence is a better one. Indeed, let us consider a tree T with v as its root, and its subtree T_w with w as its root (where w is a child of v). Assume that for some strategy function f the visibility sequence in T_w is F_w, and in T it is F_v. Say that in the subtree T_w someone managed to define a strategy function g giving a visibility sequence G_w, lexicographically smaller than F_w. How can we extend g to the whole tree, so that we will obtain a function whose visibility sequence is lexicographically smaller that F_v or equal to F_v? It is very easy. Let a be the largest number that appears in F_w but does not appear in G_w (all greater numbers appear in none of them, or in both). If $f(v) > a$, on the rest of the tree we can leave the values of f. All values of g visible in the subtree were visible also when there were f (values greater than a), or they will be covered by $f(v)$ (values smaller than a), so the function defined in this way will be a strategy function. Its visibility sequence in T is F_v, as for f. On the other hand, if $f(v) < a$, we assign value a to the root, and on the rest of the tree we leave the values of f. It is easy to check that the function defined in this way is a strategy function. Its visibility sequence in T

begins in the same way as F_v, but numbers smaller than a stop being visible. So in this case we also obtain a visibility sequence which is lexicographically smaller than F_v, or equal to F_v, whereas the case $f(v) = a$ is impossible, as condition (∗) would not be satisfied for v and the node of T_w in which a is visible.

As a consequence of the above we see that, if we want to obtain the lexicographically smallest visibility sequence for the whole tree, it is enough to choose for each subtree (among the subtrees rooted in the children of the root) a strategy function that gives the lexicographically smallest visibility sequence for this subtree, and then to choose some value in the root. It is not difficult to see that the smaller value will be assigned to the root; the lexicographically smaller will be the visibility sequence for the whole tree. Of course, the same reasoning can be applied by induction for all smaller subtrees. This proves that our greedy approach is correct.

Algorithm

We have not yet said how to write a procedure that assigns to a node v the smallest possible value of strategy function, which does not conflict with the values already assigned in its descendants. Here again the visibility sequences are useful. As we have already said, for the whole tree rooted in a child of v, it is enough to know its visibility sequence. Let $F_1, F_2, \ldots, F_k$ be the visibility sequences for the subtrees rooted in the children of v. Condition (∗) for pairs containing nodes from one subtree is satisfied by assumption. We have to ensure that condition (∗) is satisfied for pairs of nodes from different subtrees. If some number a appears in two sequences F_i, F_j, we have to put in v something greater than a. Additionally, we have to ensure condition (∗) for pairs consisting of v and a node from one of the subtrees. Thus we cannot place in v a value which is already present in some sequence F_i. In other words: we look for the maximal number a appearing in two sequences F_i, F_j (assuming $a = -1$ if such a number does not exist); to node v we assign the smallest number greater than a which does not appear in any F_i. Moreover, in order to perform the same operation in the parent of v, we want to compute the visibility sequence for the whole subtree rooted in v: it contains $f(v)$ and those values from each F_i which are greater than $f(v)$.

The whole algorithm can be written using the pseudocode below.

```
Function Compute(v : node)
    {let w_1, w_2, ..., w_k—children of v}
    for i := 1 to k do
        F_i := Compute(w_i)
    a := (the greatest number appearing in two F_i
          or −1 if such a number does not exist)
    b := (the smallest number greater than a not appearing in any F_i)
    return {b} ∪ {c ∈ F_i : c > b, 1 ≤ i ≤ k}

Algorithm HowManyQuestions()
    choose arbitrary node v as the root
    output the greatest element of Compute(v)
```

Notice that in every tree of size n it is sufficient to ask $\lfloor \log n \rfloor$ questions. It is true because in every tree of size n there exists a node dividing it into parts from which each has at most $\frac{n}{2}$ nodes; in such a node we ask the first question, and we continue the same way. Thus the visibility sequences processed by the algorithm will have length at most $\lfloor \log n \rfloor$. It is easy to implement the above algorithm so that it works in time $O(n \log n)$. As a curiosity, let us add that the solution can also be implemented in linear time, representing the visibility sequences as integers.

/ Game of Tokens

Contest: Polish Olympiad in Informatics Training Camp 2010
Task author: Jakub Pawlewicz
Solution description: Jakub Pawlewicz
Memory: 32 MB
http://main.edu.pl/en/archive/ontak/2010/zet

The *Game of Tokens* is a two-player game that is played with n tokens, labeled with the integers from 0 to $n - 1$, on an $n \times n$ square grid filled with numbers, one per cell. The tokens and the grid are visible to both players.

The players take turns picking up tokens from the table, one per move, until all of the tokens are removed. Then the score for each player is computed as the sum of numbers in the cells captured by the player, where the cell in the i-th row and j-th column is captured if and only if the player picked up the tokens with labels i and j (or, if $i = j$, the single token i). The objective of the game is to maximize one's score advantage over the other player (and, if that is not possible, to minimize one's disadvantage). Your task is to design and implement a program that will play *Game of Tokens*.

Programming interface

Your solution should implement the following functions.

- void startgame(int *n*)—this function will be called exactly once, before either player's first move. The argument n indicates the number of tokens and the size of the grid ($1 \le n \le 40$).
- void opponentmove(int *token*)—this function is called when the opponent picks up a token, and should update your idea of the state of the game; the argument token is the label of the token ($0 \le token \le n - 1$) that the opponent chose.
- int play()—this function should return your move, i.e. the label of the token that your implementation decides to take in your current move.

The functions play and opponentmove will be called in turns, with the first call indicating who makes the first move. In your implementation you may assume that the opponent's moves are correct and, in particular, that no token is picked up more than once.

The function valueat will be provided; it allows the state of any grid cell to be discovered, and may be called at any point in the program as many times as needed.

→ int valueat(int *row*, int *col*)—this function returns the number placed in row *row* and column *col* of the grid ($0 \le row, col \le n - 1$). Numbers in the grid are nonnegative integers not larger than 10^6.

Your solution (functions) must neither produce any output nor terminate the execution of the program. The program will terminate automatically after the last move.

Scoring

Your submission will gain a point for each match played in an optimal way: that is, maximizing the value of your score minus the opponent's score. This means that points may be gained even if the match is lost, provided that your program made optimal moves against the opponent and thereby minimized the loss.

Example

For $n = 3$ and the grid

2 8 1
3 5 5
3 2 7

a possible sequence of function calls when running a program that plays the *Game of Tokens* is:

```
-> startgame(3)
    <- valueat(0,0) = 2
    <- valueat(0,1) = 8
    <- valueat(0,2) = 1
    <- valueat(1,0) = 3
    <- valueat(1,1) = 5
    <- valueat(1,2) = 5
    <- valueat(2,0) = 3
    <- valueat(2,1) = 2
    <- valueat(2,2) = 7
-> opponentmove(1)
-> play() = 2
-> opponentmove(0)
```

Explanation of the example: As required, startgame is called first. In this example implementation, it calls valueat for each cell to return the numbers in the grid. The first move is made by the opponent, who picks up token 1 (call to opponentmove(1)). The player responded by picking up token 2. In the last move, the opponent takes token 0 (the only remaining token) and the game terminates. Both players made optimal moves, and the game ended with the opponent scoring 11 more points than the player.

/ Solution

The task's constraint ($n \leq 40$) suggests that one can try some brute-force methods, such as a game-tree search or some variant of the meet-in-the-middle technique. However, it turns out that the token game can be reduced to another very simple game with an obvious strategy. To obtain that reduction, we take a closer look into the game result.

Some notations have to be introduced. We represent a rectangular board with numbers by a matrix:

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{bmatrix}$$

Let $T = \{0,1,\ldots,n-1\}$ denote the set of all tokens. Let T_i, for $i = 1,2$, denote tokens collected by the i-th player. The pair of sets T_1 and T_2 is a partition of the set T: $T_1 \cup T_2 = T$ and $T_1 \cap T_2 = \varnothing$. The advantage of the first player over the second player is determined by the formula

$$Res = \sum_{i,j \in T_1} a_{i,j} - \sum_{i,j \in T_2} a_{i,j}.$$

Let us try to express the value of *Res*—the game result—by other means. Let us investigate the influence of a player taking token number i. Firstly, the opponent certainly gets no points for numbers from the i-th row and i-th column. Moreover, the benefit from the field $a_{i,i}$ is doubled—besides denying the opponent the field's points, the player is also able to claim them.

Let us denote the sum of numbers of the i-th row of the matrix A as r_i and the sum of numbers of the i-th column as c_i:

$$r_i = \sum_{j \in T} a_{i,j}, \quad c_i = \sum_{j \in T} a_{j,i}.$$

We can try to use these symbols to express the game result. Let us count points according to the above reasoning. Assume that, for taking the i-th token,

the player gets $r_i + c_i$ points, because this action strips from the opponent the opportunity to get points for numbers from the i-th row and i-th column, and at the same time acquires the points for field $a_{i,i}$, which is counted twice in the sum $r_i + c_i$. This counting method gives the following total advantage for the first player.

$$Res' = \sum_{i \in T_1} (r_i + c_i) - \sum_{i \in T_2} (r_i + c_i).$$

How does this method compare to the previous one? Consider the element $a_{i,j}$. If both tokens i and j belong to the first player then $a_{i,j}$ is counted twice in the first sum. But if both tokes belong to the opponent then $a_{i,j}$ is subtracted twice because it is in the second sum. In the third case, if tokens i and j were split between players then $a_{i,j}$ is added in the first sum and subtracted in the second sum.

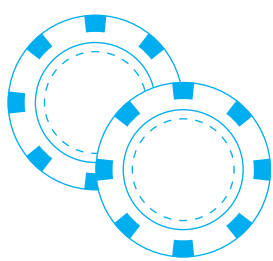
Therefore, we see that the advantage counted using the new method is twice the advantage computed with the regular rules. Thus, if we assign to token i the value

$$v_i = r_i + c_i,$$

then the doubled game result will be

$$2Res = \sum_{i \in T_1} v_i - \sum_{i \in T_2} v_i.$$

In this way, the token game reduces to a game in which each token has a fixed value v_i and the total number of points obtained by a player is the sum of possessed tokens. In such a game, a greedy strategy is optimal: always take the token with the largest value.



Contents

Contest tasks say a lot about the quality of a programming competition. 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 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.