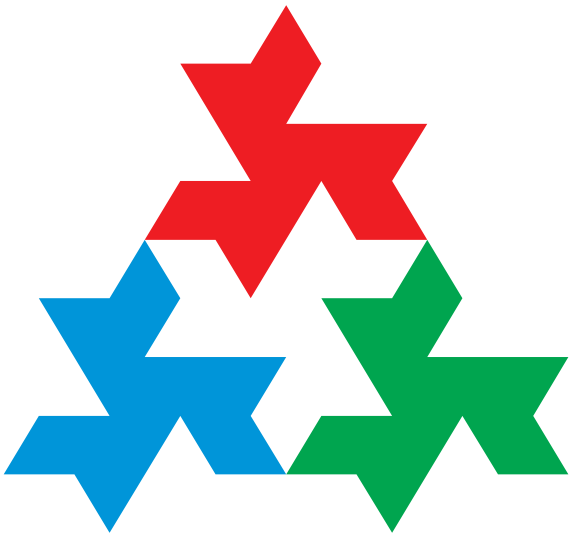
MINISTERSTWO EDUKACJI NARODOWEJ
UNIWERSYTET WROCŁAWSKI
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ



XVI OLIMPIADA INFORMATYCZNA

2008/2009

Olimpiada Informatyczna jest organizowana przy współudziale

WARSZAWA, 2009

MINISTERSTWO EDUKACJI NARODOWEJ
UNIWERSYTET WROCŁAWSKI
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

XVI OLIMPIADA INFORMATYCZNA
2008/2009

WARSZAWA, 2009

Autorzy tekstów:

dr Marek Biskup
prof. dr hab. Krzysztof Diks
Marian M. Kędzierski
dr Marcin Kubica
Tomasz Kulczyński
Jakub Łącki
Mirosław Michalski
Błażej Osiński
mgr Paweł Parys
mgr Jakub Radoszewski
prof. dr hab. Wojciech Rytter
Juliusz Sompolski
dr Andrzej Walat
mgr Bartosz Walczak
Michał Włodarczyk
dr Jakub Wojtaszczyk

Autorzy programów:

Adam Gawarkiewicz
Bartosz Górski
Tomasz Kulczyński
Jakub Łącki
Mirosław Michalski
Piotr Niedźwiedź
Błażej Osiński
mgr Paweł Parys
mgr Jakub Radoszewski
Juliusz Sompolski
Szymon Wrzyszcz

Opracowanie i redakcja:

Marcin Andrychowicz
dr Marcin Kubica
mgr Jakub Radoszewski

Tłumaczenie treści zadań:

dr Marcin Kubica
Tomasz Kulczyński
Piotr Niedźwiedź
dr Jakub Pawlewicz
mgr Jakub Radoszewski

Skład:

Marcin Andrychowicz

Pozycja dotowana przez Ministerstwo Edukacji Narodowej.

Druk książki został sfinansowany przez

ISBN 978–83–922946–6–5

Spis treści

Wstęp

Oddajemy do rąk Czytelników szesnasty tom sprawozdań z przebiegu Olimpiady Informatycznej, tym razem z roku szkolnego 2008/2009. Olimpiada Informatyczna jest jedną z najmłodszych olimpiad przedmiotowych, ale jednocześnie odnoszącą znaczące sukcesy na arenie międzynarodowej, z których największe to zwycięstwa indywidualne polskich uczniów w Międzynarodowej Olimpiadzie Informatycznej w latach 2006 (Filip Wolski) i 2007 (Tomasz Kulczyński). Sukcesy Olimpiady to wynik pracy i zaangażowania wielu osób — uczniów, nauczycieli, organizatorów Olimpiady — oraz instytucji — Ministerstwa Edukacji Narodowej, sponsorów oraz uczelni wyższych i instytucji oświatowych. W tym miejscu chciałbym słowa wdzięczności i uznania za 16 lat współpracy przy organizacji Olimpiady Informatycznej skierować do Uniwersytetu Wrocławskiego. Uniwersytet Wrocławski powołał Olimpiadę Informatyczną dzięki inicjatywie pana profesora Macieja Sysły, a w ostatnich latach aktywnie ją wspierał dzięki zaangażowaniu i życzliwości pana profesora Leszka Pacholskiego.

W roku szkolnym 2009/2010 głównym organizatorem Olimpiady została Fundacja Rozwoju Informatyki, która będzie w tym dziele współpracować ze wszystkimi dotychczas zaangażowanymi osobami i instytucjami.

Szesnasta Olimpiada Informatyczna stała na bardzo wysokim poziomie, o czym świadczą wyniki jej laureatów w konkursach międzynarodowych. W XV Bałtyckiej Olimpiadzie Informatycznej zwyciężył Jakub Pachocki, drugie miejsce zajęła Anna Piekarska, a trzecie Adrian Jaskółka. Cała trójka została nagrodzona złotymi medalami. Ponadto Michał Makarewicz zdobył srebrny medal, a Dawid Dąbrowski — brązowy medal.

W lipcu odbyła się XVI Olimpiada Informatyczna Krajów Europy Środkowej. Wszyscy nasi reprezentanci zdobyli medale: Jakub Pachocki — złoty medal, drugie miejsce w rankingu, Tomasz Kociumaka — srebrny medal, Adam Karczmarz — srebrny medal, Jarosław Błasiok — brązowy medal. Jeszcze lepiej reprezentacja Polski wypadła na XXI Międzynarodowej Olimpiadzie Informatycznej, która odbyła się w sierpniu w Bułgarii. Polska drużyna zdobyła trzecie miejsce w świecie, ex aequo z Tajwanem i USA, zdobywając 2 złote i 2 srebrne medale, po Chinach (3 złote i 1 srebrny medal) oraz Korei (3 złote i 1 brązowy medal). Indywidualnie siódme miejsce na świecie, a drugie wśród zawodników z Europy, zdobył Tomek Kociumaka. Wszyscy pozostali nasi reprezentanci również zdobyli medale: złoty medal zdobył Jarek Błasiok (24. miejsce), a srebrne medale — Adam Karczmarz (37. miejsce) i Jakub Pachocki (49. miejsce).

Tegoroczne wyniki potwierdzają, że młodzi polscy informatycy należą do czołówki światowej i kontynuują sukcesy swoich poprzedników. Ta książeczka może być pomocna kolejnym pokoleniom młodych informatyków. Znajdują się w niej dokładne opisy rozwiązań zadań olimpijskich. Każdego, kto chciałby poćwiczyć rozwiązywanie takich zadań, zachęcam do zajrzenia na stronę `main.edu.pl`.

Wszystkim Czytelnikom życzę dużo przyjemności z czytania tych materiałów, a przyszłym olimpijczykom — sukcesów.

Krzysztof Diks

Sprawozdanie z przebiegu XVI Olimpiady Informatycznej w roku szkolnym 2008/2009

Olimpiada Informatyczna została powołana 10 grudnia 1993 roku przez Instytut Informatyki Uniwersytetu Wrocławskiego zgodnie z zarządzeniem nr 28 Ministra Edukacji Narodowej z dnia 14 września 1992 roku. Olimpiada działa zgodnie z Rozporządzeniem Ministra Edukacji Narodowej i Sportu z dnia 29 stycznia 2002 roku w sprawie organizacji oraz sposobu przeprowadzenia konkursów, turniejów i olimpiad (Dz. U. 02.13.125). Organizatorem XVI Olimpiady Informatycznej jest Uniwersytet Wrocławski.

ORGANIZACJA ZAWODÓW

Olimpiada Informatyczna jest trójstopniowa. Rozwiązaniem każdego zadania zawodów I, II i III stopnia jest program napisany w jednym z ustalonych przez Komitet Główny Olimpiady języków programowania lub plik z wynikami. Zawody I stopnia miały charakter otwartego konkursu dla uczniów wszystkich typów szkół młodzieżowych.

14 października 2008 r. rozesłano do 3716 szkół i zespołów szkół młodzieżowych ponadgimnazjalnych plakaty informujące o rozpoczęciu XVI Olimpiady oraz promujące sukcesy młodych polskich informatyków. Zawody I stopnia rozpoczęły się 20 października 2008 r. Ostatecznym terminem nadsyłania prac konkursowych był 17 listopada 2008 r.

Zawody II i III stopnia były dwudniowymi sesjami stacjonarnymi, poprzedzonymi jednodniowymi sesjami próbnymi. Zawody II stopnia odbyły się w ośmiu okręgach: Katowicach, Krakowie, Poznaniu, Rzeszowie, Sopocie, Toruniu, Warszawie i Wrocławiu w dniach 10–12.02.2009 r., natomiast zawody III stopnia odbyły się w ośrodku firmy Combidata Poland S.A. w Sopocie, w dniach 31.03–04.04.2009 r.

Uroczystość zakończenia XVI Olimpiady Informatycznej odbyła się 04.04.2009 r. w siedzibie firm Combidata Poland i Asseco Poland w Gdyni przy ul. Podolskiej 21.

SKŁAD OSOBOWY KOMITETÓW OLIMPIADY INFORMATYCZNEJ

Komitet Główny

przewodniczący:
prof. dr hab. Krzysztof Diks (Uniwersytet Warszawski)

zastępcy przewodniczącego:
dr Przemysława Kanarek (Uniwersytet Wrocławski)
prof. dr hab. Paweł Idziak (Uniwersytet Jagielloński)

sekretarz naukowy:
dr Marcin Kubica (Uniwersytet Warszawski)

kierownik Jury:
mgr Jakub Radoszewski (Uniwersytet Warszawski)

kierownik techniczny:
Szymon Acedański (Uniwersytet Warszawski)

kierownik organizacyjny:
Tadeusz Kuran

członkowie:
dr Piotr Chrząstowski–Wachtel (Uniwersytet Warszawski)
prof. dr hab. inż. Zbigniew Czech (Politechnika Śląska)
dr hab. inż. Piotr Formanowicz (Politechnika Poznańska)
dr Barbara Klunder (Uniwersytet Mikołaja Kopernika w Toruniu)
mgr Anna Beata Kwiatkowska (Gimnazjum i Liceum Akademickie w Toruniu)
prof. dr hab. Krzysztof Loryś (Uniwersytet Wrocławski)
prof. dr hab. Jan Madey (Uniwersytet Warszawski)
prof. dr hab. Wojciech Rytter (Uniwersytet Warszawski)
dr hab. Krzysztof Stencel, prof. UW (Uniwersytet Warszawski)
prof. dr hab. Maciej Sysło (Uniwersytet Wrocławski)
dr Maciej Ślusarek (Uniwersytet Jagielloński)
mgr Krzysztof J. Święcicki (Partnerstwo dla Przyszłości)
mgr Tomasz Waleń (Uniwersytet Warszawski)
dr hab. inż. Stanisław Waligórski, prof. UW (Uniwersytet Warszawski)

sekretarz Komitetu Głównego:
Monika Kozłowska–Zając (OEIiZK)

Komitet Główny ma siedzibę w Ośrodku Edukacji Informatycznej i Zastosowań Komputerów w Warszawie przy ul. Nowogrodzkiej 73.
Komitet Główny odbył 4 posiedzenia.

Komitety okręgowe

Komitet Okręgowy w Warszawie
przewodniczący:
dr Łukasz Kowalik (Uniwersytet Warszawski)
sekretarz:
Monika Kozłowska–Zając (OEIiZK)
członkowie:
dr Marcin Kubica (Uniwersytet Warszawski)

Komitet Okręgowy ma siedzibę w Ośrodku Edukacji Informatycznej i Zastosowań Komputerów w Warszawie przy ul. Nowogrodzkiej 73.

Komitet Okręgowy we Wrocławiu
przewodniczący:
prof. dr hab. Krzysztof Loryś (Uniwersytet Wrocławski)
zastępca przewodniczącego:
dr Helena Krupicka (Uniwersytet Wrocławski)

sekretarz:
inż. Maria Woźniak (Uniwersytet Wrocławski)
członkowie:
dr Tomasz Jurdziński (Uniwersytet Wrocławski)
dr Przemysława Kanarek (Uniwersytet Wrocławski)
dr Witold Karczewski (Uniwersytet Wrocławski)

Siedzibą Komitetu Okręgowego jest Instytut Informatyki Uniwersytetu Wrocławskiego, ul. Joliot–Curie 15.

Komitet Okręgowy w Toruniu:
przewodniczący:
prof. dr hab. Adam Ochmański (Uniwersytet Mikołaja Kopernika w Toruniu)
zastępca przewodniczącego:
dr Barbara Klunder (Uniwersytet Mikołaja Kopernika w Toruniu)
sekretarz:
mgr Rafał Kluszczyński (Uniwersytet Mikołaja Kopernika w Toruniu)
członkowie:
mgr Anna Beata Kwiatkowska (Gimnazjum i Liceum Akademickie w Toruniu)
mgr Robert Mroczkowski (Uniwersytet Mikołaja Kopernika w Toruniu)
mgr Radosław Rudnicki (Uniwersytet Mikołaja Kopernika w Toruniu)

Siedzibą Komitetu Okręgowego jest Wydział Matematyki i Informatyki Uniwersytetu Mikołaja Kopernika w Toruniu, ul. Chopina 12/18.

Górnośląski Komitet Okręgowy
przewodniczący:
prof. dr hab. Zbigniew Czech (Politechnika Śląska w Gliwicach)
zastępca przewodniczącego:
mgr inż. Jacek Widuch (Politechnika Śląska w Gliwicach)
sekretarz:
mgr inż. Tomasz Wesołowski (Politechnika Śląska w Gliwicach)
członkowie:
mgr inż. Przemysław Kudłacik (Politechnika Śląska w Gliwicach)
mgr inż. Krzysztof Simiński (Politechnika Śląska w Gliwicach)
mgr inż. Tomasz Wojdyła (Politechnika Śląska w Gliwicach)

Siedzibą Górnośląskiego Komitetu Okręgowego jest Politechnika Śląska w Gliwicach, ul. Akademicka 16.

Komitet Okręgowy w Krakowie
przewodniczący:
prof. dr hab. Paweł Idziak (Uniwersytet Jagielloński)
zastępca przewodniczącego:
dr Maciej Ślusarek (Uniwersytet Jagielloński)
sekretarz:
mgr Monika Gilert (Uniwersytet Jagielloński)
członkowie:
mgr Henryk Białek (emerytowany pracownik Małopolskiego Kuratorium Oświaty)
mgr Grzegorz Gutowski (Uniwersytet Jagielloński)
Marek Wróbel (student Uniwersytetu Jagiellońskiego)

Siedzibą Komitetu Okręgowego w Krakowie jest Katedra Algorytmiki Uniwersytetu Jagiellońskiego, ul. Gronostajowa 3.

Komitet Okręgowy w Rzeszowie
przewodniczący:
prof. dr hab. inż. Stanisław Paszczyński (Wyższa Szkoła Informatyki i Zarządzania w Rzeszowie)
zastępca przewodniczącego:
dr Marek Jaszuk (Wyższa Szkoła Informatyki i Zarządzania w Rzeszowie)
sekretarz:
mgr inż. Grzegorz Owsiany (Wyższa Szkoła Informatyki i Zarządzania w Rzeszowie)
członkowie:
mgr inż. Piotr Błajdo (Wyższa Szkoła Informatyki i Zarządzania w Rzeszowie)
Maksymilian Knap (Wyższa Szkoła Informatyki i Zarządzania w Rzeszowie)
mgr Czesław Wal (Wyższa Szkoła Informatyki i Zarządzania w Rzeszowie)
mgr inż. Dominik Wojtaszek (Wyższa Szkoła Informatyki i Zarządzania w Rzeszowie)

Siedzibą Komitetu Okręgowego w Rzeszowie jest Wyższa Szkoła Informatyki i Zarządzania, ul. Sucharskiego 2.

Komitet Okręgowy w Poznaniu:
przewodniczący:
dr hab. inż. Małgorzata Sterna (Politechnika Poznańska)
zastępca przewodniczącego:
dr Jacek Marciniak (Uniwersytet Adama Mickiewicza w Poznaniu)
sekretarz:
mgr inż. Przemysław Wesołek (Politechnika Poznańska)
członkowie:
Hanna Ćwiek (Politechnika Poznańska)
mgr inż. Piotr Gawron (Politechnika Poznańska)
dr Maciej Machowiak (Politechnika Poznańska)
dr inż. Maciej Miłostan (Politechnika Poznańska)
Michał Połetek (Politechnika Poznańska)
mgr inż. Szymon Wąsik (Politechnika Poznańska)

Siedzibą Komitetu Okręgowego jest Instytut Informatyki Politechniki Poznańskiej, ul. Piotrowo 2.
Strona internetowa Komitetu Okręgowego: `http://www.cs.put.poznan.pl/oi/`.

Pomorski Komitet Okręgowy:
przewodniczący:
prof. dr hab. inż. Marek Kubale (Politechnika Gdańska)
zastępca przewodniczącego:
dr hab. Andrzej Szepietowski (Uniwersytet Gdański)
sekretarz:
mgr inż. Tomasz Dobrowolski (Politechnika Gdańska)
członkowie:
dr inż. Dariusz Dereniowski (Politechnika Gdańska)
dr inż. Adrian Kosowski (Politechnika Gdańska)
dr inż. Michał Małafiejski (Politechnika Gdańska)

mgr inż. Ryszard Szubartowski (III Liceum Ogólnokształcące im. Marynarki Wojennej w Gdyni)
dr Paweł Żyliński (Uniwersytet Gdański)

Siedzibą Komitetu Okręgowego jest Politechnika Gdańska, Wydział Elektroniki, Telekomunikacji i Informatyki, ul. Gabriela Narutowicza 11/12, Gdańsk Wrzeszcz.

Jury Olimpiady Informatycznej

W pracach Jury, które nadzorował Krzysztof Diks, a którymi kierowali Szymon Acedański i Jakub Radoszewski, brali udział doktoranci i studenci Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego oraz Wydziału Informatyki i Zarządzania Politechniki Poznańskiej:

mgr Marek Cygan
Adam Gawarkiewicz
Konrad Gołuchowski
Bartosz Górski
mgr Tomasz Idziaszek
mgr Adam Iwanicki
Tomasz Kulczyński
Jakub Łącki
Marek Marczykowski
Mirosław Michalski
Jacek Migdał
Piotr Niedźwiedź
Błażej Osiński
mgr Paweł Parys
mgr Marcin Pilipczuk
Michał Pilipczuk
Juliusz Sompolski
Bartosz Szreder
Wojciech Śmietanka
Wojciech Tyczyński
Filip Wolski
Szymon Wrzyszcz

ZAWODY I STOPNIA

W zawodach I stopnia XVI Olimpiady Informatycznej wzięło udział 975 zawodników. Decyzją Komitetu Głównego zdyskwalifikowano 39 zawodników. Powodem dyskwalifikacji była niesamodzielność rozwiązań zadań konkursowych. Sklasyfikowano 936 zawodników.

Decyzją Komitetu Głównego do zawodów zostało dopuszczonych 44 uczniów gimnazjów oraz jeden uczeń szkoły podstawowej. Byli to uczniowie z następujących szkół:

• Gimnazjum nr 24 przy III LO	Gdynia	10 uczniów
• Gimnazjum nr 16 przy XIII LO	Szczecin	4
• Gimnazjum nr 58 im. Władysława IV	Warszawa	3

• Gimnazjum nr 2	Kraków	2
• Gimnazjum i Liceum Akademickie	Toruń	2
• Gimnazjum Dwujęzyczne nr 49	Wrocław	2
• Gimnazjum nr 10	Zielona Góra	2
• Społeczna Szkoła Podstawowa nr 3 Dębinka	Poznań	1 uczeń
• Publiczne Gimnazjum nr 2	Brzesko	1
• Gimnazjum nr 53	Bydgoszcz	1
• Gimnazjum Towarzystwa Salezjańskiego	Bydgoszcz	1
• Gimnazjum przy ZS nr 2	Hrubieszów	1
• Gimnazjum nr 1	Jaworzno	1
• Gimnazjum Integracyjne nr 4	Kielce	1
• Zespół Szkół	Koziegłowy	1
• VI Prywatne Gimnazjum Akademickie	Kraków	1
• Gimnazjum nr 24	Lublin	1
• Prywatne Gimnazjum im. I. J. Paderewskiego	Lublin	1
• Gimnazjum nr 4	Olsztyn	1
• Zespół Szkół	Rzerzęczyce	1
• Gimnazjum im. Jana Pawła II	Rzeszów	1
• Publiczne Gimnazjum nr 5	Siedlce	1
• Zespół Szkół Integracyjnych	Skierniewice	1
• Publiczne Gimnazjum nr 1 im. M. Kopernika	Starogard Gdański	1
• Gimnazjum	Trzebownisko	1
• Gimnazjum nr 13 im. Stanisława Staszica	Warszawa	1
• Gimnazjum nr 59 im. Tadeusza Reytana	Warszawa	1

Kolejność województw pod względem liczby uczestników była następująca:

małopolskie	158 zawodników	podlaskie	44
mazowieckie	122	zachodniopomorskie	37
pomorskie	112	lubelskie	32
śląskie	98	świętokrzyskie	26
dolnośląskie	83	łódzkie	23
kujawsko–pomorskie	79	opolskie	14
wielkopolskie	46	warmińsko–mazurskie	10
podkarpackie	44	lubuskie	8

W zawodach I stopnia najliczniej reprezentowane były szkoły:

V LO im. Augusta Witkowskiego	Kraków	96 uczniów
III LO im. Marynarki Wojennej RP	Gdynia	51
XIV LO im. Stanisława Staszica	Warszawa	41
XIV LO im. Polonii Belgijskiej	Wrocław	37
I LO im. Adama Mickiewicza	Białystok	27
VI LO im. J. i J. Śniadeckich	Bydgoszcz	25
XIII Liceum Ogólnokształcące	Szczecin	20
VIII LO im. Adama Mickiewicza	Poznań	19

VIII LO im. Marii Skłodowskiej	Katowice	18
I LO im. Tadeusza Kościuszki	Legnica	14
Liceum Akademickie	Toruń	14
V Liceum Ogólnokształcące	Bielsko–Biała	13
V LO im. Stefana Żeromskiego	Gdańsk	12
Gimnazjum nr 24 przy III LO	Gdynia	10
IV LO im. H. Sienkiewicza	Częstochowa	9
VI LO im. Jana Kochanowskiego	Radom	9
Zespół Szkół Ogólnokształcących nr 1	Bydgoszcz	8
IV LO im. Hanki Sawickiej	Kielce	8
VIII LO im. St. Wyspiańskiego	Kraków	8
I LO im. B. Nowodworskiego	Kraków	7
II LO im. H. J. Zamoyskiego	Lublin	7
Liceum Ogólnokształcące WSIiZ	Rzeszów	7
III LO im. Adama Mickiewicza	Tarnów	7
IV LO im. Tadeusza Kościuszki	Toruń	7
X Liceum Ogólnokształcące	Wrocław	7
I Liceum Ogólnokształcące	Bełchatów	6
II Liceum Ogólnokształcące	Opole	6
LO im. Mikołaja Kopernika	Ostrów Mazowiecka	6
XXVII LO im. T. Czackiego	Warszawa	6
Zespół Szkół Zawodowych	Brodnica	5
II LO im. Władysława Andersa	Chojnice	5
Ogólnokształcące Liceum Jezuitów	Gdynia	5
I LO im. Stefana Żeromskiego	Kielce	5
I LO im. Mikołaja Kopernika	Krosno	5
I LO im. Mikołaja Kopernika	Łódź	5
Zespół Szkół Elektronicznych	Rzeszów	5
VIII LO im. Władysława IV	Warszawa	5
VI Liceum Ogólnokształcące	Białystok	4
III LO im. Stefana Żeromskiego	Bielsko–Biała	4
I LO im. Juliusza Słowackiego	Chorzów	4
XIII LO im. Bohaterów Westerplatte	Kraków	4
I LO im. T. Kościuszki	Łomża	4
XII Liceum Ogólnokształcące	Łódź	4
II LO im. M. Kopernika	Mielec	4
III LO im. Św. Jana Kantego	Poznań	4
Zespół Szkół Elektronicznych	Radom	4
I LO im. B. Krzywoustego	Słupsk	4
Liceum Ogólnokształcące im. KEN	Stalowa Wola	4
Gimnazjum nr 16 przy XIII LO	Szczecin	4
II Liceum Ogólnokształcące	Chełm	3
II LO im. Władysława Pniewskiego	Gdańsk	3
Zespół Szkół Łączności	Gdańsk	3
V Liceum Ogólnokształcące	Gliwice	3
II LO im. Jana III Sobieskiego	Grudziądz	3

II LO im. M. Kopernika	Kędzierzyn–Koźle	3
II LO im. K. K. Baczyńskiego	Konin	3
I LO im. Stefana Żeromskiego	Lębork	3
Zespół Szkół Mechanicznych i Ogólnokształcących nr 5	Łomża	3
II LO im. J. Chreptowicza	Ostrowiec Świętokrzyski	3
LO im. St. Małachowskiego	Płock	3
Zespół Szkół nr 1 im. Jana Pawła II	Przysucha	3
II LO im. A. Frycza–Modrzewskiego	Rybnik	3
I Liceum Ogólnokształcące	Suwałki	3
I LO im. I. J. Paderewskiego	Wałbrzych	3
L LO im. Ruy Barbosy	Warszawa	3
Gimnazjum nr 58 im. Władysława IV	Warszawa	3
Technikum Elektroniczne nr 1	Warszawa	3
Akademickie LO przy PJWSTK	Warszawa	3
XXXVII LO im. J. Dąbrowskiego	Warszawa	3
II LO im. ks. Józefa Tischnera	Wodzisław Śląski	3
Zespół Szkół Mechaniczno–Elektrycznych	Żywiec	3

Najliczniej reprezentowane były miasta:

Kraków	124 zawodników	Mielec	6
Warszawa	80	Słupsk	6
Gdynia	71	Chorzów	5
Wrocław	52	Rybnik	5
Bydgoszcz	38	Bochnia	4
Białystok	32	Chełm	4
Szczecin	31	Grudziądz	4
Poznań	30	Inowrocław	4
Toruń	24	Konin	4
Gdańsk	21	Nowy Sącz	4
Katowice	20	Ostrów Mazowiecka	4
Bielsko–Biała	19	Płock	4
Kielce	17	Przysucha	4
Lublin	17	Stalowa Wola	4
Rzeszów	16	Wodzisław Śląski	4
Częstochowa	15	Zielona Góra	4
Legnica	15	Kędzierzyn–Koźle	3
Radom	15	Lębork	3
Gliwice	9	Ostrowiec Świętokrzyski	3
Łódź	9	Piła	3
Tarnów	9	Siedlce	3
Brodnica	7	Starachowice	3
Łomża	7	Suwałki	3
Opole	7	Wadowice	3
Bełchatów	6	Wałbrzych	3
Chojnice	6	Żywiec	3
Krosno	6		

Zawodnicy uczęszczali do następujących klas:

do klasy 6 szkoły podstawowej	1
do klasy I gimnazjum	3
do klasy II gimnazjum	7
do klasy III gimnazjum	34
do klasy I szkoły średniej	209
do klasy II szkoły średniej	356
do klasy III szkoły średniej	310
do klasy IV szkoły średniej	16

W zawodach I stopnia zawodnicy mieli do rozwiązania pięć zadań: „Gaśnice", „Kamyki", „Przyspieszenie algorytmu", „Słonie" i „Straż pożarna".

Poniższe tabele przedstawiają liczbę zawodników, którzy uzyskali określone liczby punktów za poszczególne zadania, w zestawieniu ilościowym i procentowym:

- **GAS** — Gaśnice

	GAS	
	liczba zawodników	czyli
100 pkt.	26	2,78%
75–99 pkt.	103	11,00%
50–74 pkt.	36	3,85%
1–49 pkt.	127	13,57%
0 pkt.	91	9,72%
brak rozwiązania	553	59,08%

- **KAM** — Kamyki

	KAM	
	liczba zawodników	czyli
100 pkt.	379	40,49%
75–99 pkt.	1	0,11%
50–74 pkt.	2	0,21%
1–49 pkt.	48	5,13%
0 pkt.	256	27,35%
brak rozwiązania	250	26,71%

- **PRZ** — Przyspieszenie algorytmu

	PRZ	
	liczba zawodników	czyli
100 pkt.	15	1,60%
75–99 pkt.	2	0,21%
50–74 pkt.	33	3,54%
1–49 pkt.	195	20,83%
0 pkt.	154	16,45%
brak rozwiązania	537	57,37%

- **SLO** — Słonie

	SLO	
	liczba zawodników	czyli
100 pkt.	419	44,77%
75–99 pkt.	31	3,31%
50–74 pkt.	48	5,13%
1–49 pkt.	139	14,85%
0 pkt.	104	11,11%
brak rozwiązania	195	20,83%

- **STR** — Straż pożarna

	STR	
	liczba zawodników	czyli
100 pkt.	36	3,85%
75–99 pkt.	8	0,85%
50–74 pkt.	7	0,75%
1–49 pkt.	671	71,69%
0 pkt.	103	11,00%
brak rozwiązania	111	11,86%

W sumie za wszystkie 5 zadań konkursowych:

SUMA	liczba zawodników	czyli
500 pkt.	4	0.43%
375–499 pkt.	31	3,31%
250–374 pkt.	151	16,13%
125–249 pkt.	271	28,95%
1–124 pkt.	384	41,03%
0 pkt.	95	10,15%

Wszyscy zawodnicy otrzymali informacje o uzyskanych przez siebie wynikach. Na stronie internetowej Olimpiady udostępnione były testy, na podstawie których oceniano prace.

ZAWODY II STOPNIA

Do zawodów II stopnia, które odbyły się w dniach 10–12 lutego 2009 r., zakwalifikowano 441 zawodników, którzy osiągnęli w zawodach I stopnia wynik nie mniejszy niż 138 pkt.

Zawodnicy zostali przydzieleni do następujących okręgów:

- w Gliwicach — 36 zawodników z następujących województw:
  - śląskie (36)
- w Krakowie — 105 zawodników z następujących województw:
  - małopolskie (97)
  - świętokrzyskie (8)
- w Poznaniu — 37 zawodników z następujących województw:
  - lubuskie (3)
  - wielkopolskie (20)
  - zachodniopomorskie (14)
- w Rzeszowie — 23 zawodników z następujących województw:
  - lubelskie (8)
  - podkarpackie (13)
  - świętokrzyskie (2)
- w Toruniu — 35 zawodników z następujących województw:
  - kujawsko–pomorskie (35)

- w Warszawie — 87 zawodników z następujących województw:
  - łódzkie (8)
  - mazowieckie (43)
  - podlaskie (33)
  - świętokrzyskie (2)
  - warmińsko–mazurskie (1)
- we Wrocławiu — 53 zawodników z następujących województw:
  - dolnośląskie (49)
  - lubuskie (1)
  - opolskie (3)
- w Sopocie — 65 zawodników z następujących województw:
  - pomorskie (65)

16 zawodników nie stawiło się na zawody. W zawodach II stopnia uczestniczyło 425 zawodników.

W zawodach II stopnia najliczniej reprezentowane były szkoły:

V LO im. Augusta Witkowskiego	Kraków	76 uczniów
III LO im. Marynarki Wojennej RP	Gdynia	42
XIV LO im. Polonii Belgijskiej	Wrocław	31
I LO im. Adama Mickiewicza	Białystok	22
XIV LO im. Stanisława Staszica	Warszawa	21
VI LO im. J. i J. Śniadeckich	Bydgoszcz	16
XIII Liceum Ogólnokształcące	Szczecin	12
VIII LO im. Adama Mickiewicza	Poznań	11
VIII LO im. Marii Skłodowskiej	Katowice	10
V LO im. Stefana Żeromskiego	Gdańsk	7
V Liceum Ogólnokształcące	Bielsko–Biała	6
ZS Ogólnokształcących nr 1	Bydgoszcz	6
Gimnazjum i Liceum Akademickie	Toruń	6
I LO im. B. Nowodworskiego	Kraków	5
I LO im. Tadeusza Kościuszki	Legnica	5
VI Liceum Ogólnokształcące	Białystok	4
IV LO im. Hanki Sawickiej	Kielce	4
X Liceum Ogólnokształcące	Wrocław	4
II LO im. gen. Władysława Andersa	Chojnice	3
I LO im. Juliusza Słowackiego	Chorzów	3
V Liceum Ogólnokształcące	Gliwice	3
II LO im. Jana III Sobieskiego	Kraków	3
I LO im. Mikołaja Kopernika	Łódź	3
II Liceum Ogólnokształcące	Opole	3
II LO im. J. Chreptowicza	Ostrowiec Świętokrzyski	3
VI LO im. Jana Kochanowskiego	Radom	3
LO WSIiZ	Rzeszów	3

I LO im. B. Krzywoustego	Słupsk	3
LO im. KEN	Stalowa Wola	3
VIII LO im. Władysława IV	Warszawa	3
XXVII LO im. Tadeusza Czackiego	Warszawa	3
I Liceum Ogólnokształcące	Bełchatów	3

Najliczniej reprezentowane były miasta:

Kraków	87 zawodników	Chorzów	4
Gdynia	46	Gliwice	4
Wrocław	40	Radom	4
Warszawa	31	Bełchatów	3
Białystok	27	Chojnice	3
Bydgoszcz	25	Konin	3
Poznań	15	Łomża	3
Szczecin	14	Łódź	3
Katowice	12	Nowy Sącz	3
Gdańsk	8	Opole	3
Kielce	8	Ostrowiec Świętokrzyski	3
Toruń	8	Rzeszów	3
Bielsko–Biała	7	Słupsk	3
Legnica	5	Stalowa Wola	3
Lublin	5		

W dniu 10 lutego 2009 r. odbyła się sesja próbna, na której zawodnicy rozwiązywali nieliczące się do ogólnej klasyfikacji zadanie „Wyspy na trójkątnej sieci". W dniach konkursowych zawodnicy rozwiązywali zadania: „Konduktor", „Architekci", „Przechadzka Bajtusia" i „Łyżwy", każde oceniane maksymalnie po 100 punktów.

Poniższe tabele przedstawiają liczby zawodników II etapu, którzy uzyskali podane liczby punktów za poszczególne zadania, w zestawieniu ilościowym i procentowym:

- **WYS — próbne** — Wyspy na trójkątnej sieci

	WYS — próbne	
	liczba zawodników	czyli
100 pkt.	14	3,20%
75–99 pkt.	8	1,88%
50–74 pkt.	8	1,88%
1–49 pkt.	68	16,00%
0 pkt.	106	24,94%
brak rozwiązania	221	52,00%

- **KON** — Konduktor

	KON	
	liczba zawodników	czyli
100 pkt.	40	9,41%
75–99 pkt.	19	4,47%
50–74 pkt.	2	0,47%
1–49 pkt.	54	12,71%
0 pkt.	241	56,71%
brak rozwiązania	69	16,23%

- **ARC** — Architekci

	ARC	
	liczba zawodników	czyli
100 pkt.	61	14,35%
75–99 pkt.	14	3,29%
50–74 pkt.	9	2,12%
1–49 pkt.	78	18,35%
0 pkt.	224	52,71%
brak rozwiązania	39	9,18%

- **PRZ** — Przechadzka Bajtusia

	PRZ	
	liczba zawodników	czyli
100 pkt.	4	0,94%
75–99 pkt.	2	0,47%
50–74 pkt.	14	3,29%
1–49 pkt.	145	34,12%
0 pkt.	99	23,29%
brak rozwiązania	161	37,89%

- **LYZ** — Łyżwy

	LYZ	
	liczba zawodników	czyli
100 pkt.	7	1,65%
75–99 pkt.	0	0%
50–74 pkt.	2	0,47%
1–49 pkt.	120	28,24%
0 pkt.	223	52,47%
brak rozwiązania	73	17,17%

W sumie za wszystkie 4 zadania konkursowe rozkład wyników zawodników był następujący:

SUMA	liczba zawodników	czyli
400 pkt.	1	0,24%
300–399 pkt.	6	1,41%
200–299 pkt.	28	6,59%
100–199 pkt.	65	15,29%
1–99 pkt.	183	43,06%
0 pkt.	142	33,41%

Wszystkim zawodnikom przesłano informację o uzyskanych wynikach, a na stronie Olimpiady dostępne były testy, według których sprawdzano rozwiązania. Poinformowano też dyrekcje szkół o kwalifikacji uczniów do finałów XVI Olimpiady Informatycznej.

ZAWODY III STOPNIA

Zawody III stopnia odbyły się w ośrodku firmy Combidata Poland S.A. w Sopocie w dniach od 31 marca do 4 kwietnia 2009 r.

Do zawodów III stopnia zakwalifikowano 82 najlepszych uczestników zawodów II stopnia, którzy uzyskali wynik nie mniejszy niż 124 pkt. Jeden zawodnik nie stawił się na zawody.

Zawodnicy uczęszczali do szkół w następujących województwach:

małopolskie	16 zawodników
śląskie	13
pomorskie	10
mazowieckie	9
dolnośląskie	7
kujawsko–pomorskie	7
podlaskie	6
łódzkie	3
zachodniopomorskie	3
wielkopolskie	3
lubuskie	2
opolskie	1 zawodnik
podkarpackie	1
świętokrzyskie	1

Niżej wymienione szkoły miały w finale więcej niż jednego zawodnika:

V LO im. Augusta Witkowskiego	Kraków	14 uczniów
III LO im. Marynarki Wojennej RP	Gdynia	9
I LO im. Adama Mickiewicza	Białystok	5
XIV LO im. Stanisława Staszica	Warszawa	5
VI LO im. J. i J. Śniadeckich	Bydgoszcz	4
VIII LO im. Marii Skłodowskiej	Katowice	4
XIV LO im. Polonii Belgijskiej	Wrocław	4
V Liceum Ogólnokształcące	Gliwice	3
XIII Liceum Ogólnokształcące	Szczecin	3
V Liceum Ogólnokształcące	Bielsko–Biała	2
VIII LO im. A. Mickiewicza	Poznań	2
Gimnazjum i Liceum Akademickie	Toruń	2

31 marca odbyła się sesja próbna, na której zawodnicy rozwiązywali nieliczące się do ogólnej klasyfikacji zadanie „Wiedźmak”. W dniach konkursowych zawodnicy rozwiązywali zadania: „Słowa”, „Tablice”, „Wyspa”, „Kod” i „Poszukiwania”, każde oceniane maksymalnie po 100 punktów.

Poniższe tabele przedstawiają liczby zawodników, którzy uzyskali podane liczby punktów za poszczególne zadania konkursowe, w zestawieniu ilościowym i procentowym:

- **WIE — próbne** — Wiedźmak

	WIE — próbne	
	liczba zawodników	czyli
100 pkt.	33	40,74%
75–99 pkt.	10	12,35%
50–74 pkt.	11	13,58%
1–49 pkt.	10	12,35%
0 pkt.	7	8,64%
brak rozwiązania	10	12,34%

- **SLO** — Słowa

	SLO	
	liczba zawodników	czyli
100 pkt.	0	0%
75–99 pkt.	1	1,23%
50–74 pkt.	0	0%
1–49 pkt.	15	18,52%
0 pkt.	48	59,26%
brak rozwiązania	17	20,99%

- **TAB** — Tablice

	TAB	
	liczba zawodników	czyli
100 pkt.	61	75,31%
75–99 pkt.	3	3,70%
50–74 pkt.	4	4,94%
1–49 pkt.	8	9,88%
0 pkt.	0	0%
brak rozwiązania	5	6,17%

- **WYS** — Wyspa

	WYS	
	liczba zawodników	czyli
100 pkt.	9	11,11%
75–99 pkt.	0	0%
50–74 pkt.	1	1,23%
1–49 pkt.	7	8,64%
0 pkt.	35	43,21%
brak rozwiązania	29	35,81%

- **KOD** — Kod

	KOD	
	liczba zawodników	czyli
100 pkt.	0	0%
75–99 pkt.	0	0%
50–74 pkt.	0	0%
1–49 pkt.	29	35,80%
0 pkt.	18	22,22%
brak rozwiązania	34	41,98%

- **POS** — Poszukiwania

	POS	
	liczba zawodników	czyli
100 pkt.	8	9,88%
75–99 pkt.	0	0%
50–74 pkt.	1	1,23%
1–49 pkt.	30	37,04%
0 pkt.	41	50,62%
brak rozwiązania	1	1,23%

W sumie za wszystkie 5 zadań konkursowych rozkład wyników zawodników był następujący:

SUMA	liczba zawodników	czyli
500 pkt.	0	0%
375–499 pkt.	0	0%
250–374 pkt.	7	8,64%
125–249 pkt.	28	34,57%
1–124 pkt.	44	54,32%
0 pkt.	2	2,47%

W dniu 4 kwietnia 2009 roku, w siedzibie firm Asseco Poland i Combidata Poland w Gdyni, ogłoszono wyniki finału XVI Olimpiady Informatycznej 2008/2009 i rozdano nagrody ufundowane przez: Asseco Poland SA, Wydawnictwa Naukowo–Techniczne, Ogólnopolską Fundację Edukacji Komputerowej, Olimpiadę Informatyczną, PWN i Wydawnictwo „Delta".

Poniżej zestawiono listę laureatów i wyróżnionych finalistów:

(1) **Tomasz Kociumaka**, 3 klasa, V Liceum Ogólnokształcące w Gliwicach, 345 pkt., laureat I miejsca
(2) **Jakub Pachocki**, 2 klasa, III LO im. Marynarki Wojennej RP w Gdyni, 333 pkt., laureat I miejsca
(3) **Jarosław Błasiok**, 3 klasa, VIII LO im. Marii Skłodowskiej w Katowicach, 300 pkt., laureat II miejsca
(4) **Adam Karczmarz**, 3 klasa, LO im. KEN w Stalowej Woli, 290 pkt., laureat II miejsca
(5) **Jakub Oćwieja**, 3 klasa, V Liceum Ogólnokształcące w Bielsku–Białej, 290 pkt., laureat II miejsca
(6) **Adrian Jaskółka**, 2 klasa, I LO im. Adama Mickiewicza w Białymstoku, 269 pkt., laureat II miejsca
(7) **Jakub Adamek**, 3 klasa, V LO im. Augusta Witkowskiego w Krakowie, 251 pkt., laureat II miejsca
(8) **Michał Makarewicz**, 2 klasa, I LO im. Adama Mickiewicza w Białymstoku, 240 pkt., laureat II miejsca
(9) **Wiktor Janas**, 3 klasa, XIV LO im. Polonii Belgijskiej we Wrocławiu, 220 pkt., laureat II miejsca
(10) **Tomasz Kłeczek**, 3 klasa, V LO im. Augusta Witkowskiego w Krakowie, 220 pkt., laureat II miejsca
(11) **Sławomir Fraś**, 3 klasa, I LO im. K. Miarki w Żorach, 200 pkt., laureat III miejsca
(12) **Paweł Wanat**, 3 klasa, V LO im. Augusta Witkowskiego w Krakowie, 200 pkt., laureat III miejsca
(13) **Anna Piekarska**, 2 klasa, XIV LO im. Polonii Belgijskiej we Wrocławiu, 199 pkt., laureatka III miejsca
(14) **Marcin Wrochna**, 3 klasa, XIV LO im. Stanisława Staszica w Warszawie, 195 pkt., laureat III miejsca
(15) **Michał Bejda**, 3 klasa, V LO im. Augusta Witkowskiego w Krakowie, 181 pkt., laureat III miejsca
(16) **Dawid Dąbrowski**, 2 klasa, III LO im. Marynarki Wojennej RP w Gdyni, 181 pkt., laureat III miejsca
(17) **Karol Pokorski**, 1 klasa, III LO im. Marynarki Wojennej RP w Gdyni, 179 pkt., laureat III miejsca
(18) **Adam Polak**, 3 klasa, V LO im. Augusta Witkowskiego w Krakowie, 168 pkt., laureat III miejsca
(19) **Grzegorz Prusak**, 2 klasa, II Liceum Ogólnokształcące w Poznaniu, 158 pkt., laureat III miejsca

(20) **Krzysztof Opolski**, 3 klasa, III LO im. Marii Dąbrowskiej w Płocku, 151 pkt., laureat III miejsca

(21) **Igor Adamski**, 2 klasa, V LO im. Augusta Witkowskiego w Krakowie, 150 pkt., laureat III miejsca

(22) **Artur Kraska**, 3 klasa, ZS Ogólnokształcących nr 1 w Bydgoszczy, 150 pkt., laureat III miejsca

(23) **Piotr Szefler**, 1 klasa, Gimnazjum i Liceum Akademickie w Toruniu, 149 pkt., laureat III miejsca

(24) **Robert Kozikowski**, 3 klasa, I LO im. Adama Mickiewicza w Białymstoku, 147 pkt., laureat III miejsca

(25) **Marcin Smulewicz**, 3 klasa gimnazjum, Zespół Szkół Integracyjnych w Skierniewicach, 142 pkt., finalista z wyróżnieniem

(26) **Grzegorz Guśpiel**, 2 klasa, V LO im. Augusta Witkowskiego w Krakowie, 141 pkt., finalista z wyróżnieniem

(27) **Krzysztof Król**, 2 klasa, XIV LO im. Polonii Belgijskiej we Wrocławiu, 141 pkt., finalista z wyróżnieniem

(28) **Daniel Malinowski**, 3 klasa, VIII LO im. Marii Skłodowskiej w Katowicach, 137 pkt., finalista z wyróżnieniem

(29) **Piotr Godlewski**, 3 klasa, VI LO im. Jana Kochanowskiego w Radomiu, 133 pkt., finalista z wyróżnieniem

(30) **Łukasz Jocz**, 1 klasa, I LO im. Adama Mickiewicza w Białymstoku, 131 pkt., finalista z wyróżnieniem

(31) **Janusz Kudełka**, 3 klasa, VI LO im. J. i J. Śniadeckich w Bydgoszczy, 131 pkt., finalista z wyróżnieniem

(32) **Jacek Szmigiel**, 3 klasa, V LO im. Augusta Witkowskiego w Krakowie, 131 pkt., finalista z wyróżnieniem

(33) **Adam Błaszkiewicz**, 3 klasa, XV Liceum Ogólnokształcące we Wrocławiu, 130 pkt., finalista z wyróżnieniem

(34) **Piotr Suwara**, 2 klasa, XIV LO im. Stanisława Staszica w Warszawie, 130 pkt., finalista z wyróżnieniem

(35) **Mateusz Litwin**, 3 klasa, III LO im. Marynarki Wojennej RP w Gdyni, 128 pkt., finalista z wyróżnieniem

Lista pozostałych finalistów w kolejności alfabetycznej:

- **Marcin Sebastian Baczyński**, 2 klasa, VIII LO im. Władysława IV w Warszawie
- **Konrad Baumgart**, 3 klasa, VI LO im. J. i J. Śniadeckich w Bydgoszczy
- **Arkadiusz Betkier**, 3 klasa, XIV LO im. Stanisława Staszica w Warszawie
- **Maciej Borsz**, 1 klasa, VI LO im. J. i J. Śniadeckich w Bydgoszczy
- **Marcin Cyran**, 3 klasa, I Liceum Ogólnokształcące w Bełchatowie
- **Radosław Dembkowski**, 2 klasa, III LO im. Marynarki Wojennej RP w Gdyni
- **Andrzej Dorobisz**, 2 klasa, V LO im. Augusta Witkowskiego w Krakowie
- **Bartłomiej Dudek**, 3 klasa gim., Gimnazjum Dwujęzyczne nr 49 we Wrocławiu

- **Szymon Gut**, 3 klasa, V Liceum Ogólnokształcące w Gliwicach
- **Michał Hordecki**, 2 klasa, VIII LO im. Adama Mickiewicza w Poznaniu
- **Adam Iwaniuk**, 3 klasa, VI Liceum Ogólnokształcące w Białymstoku
- **Wiktor Jakubiuk**, 3 klasa, Dulwich College w Londynie
- **Krzysztof Jan Kiewicz**, 3 klasa gimnazjum, Gimnazjum nr 58 im. Władysława IV w Warszawie
- **Karol Konaszyński**, 3 klasa, XIV LO im. Polonii Belgijskiej we Wrocławiu
- **Michał Kosnowski**, 1 klasa, Gimnazjum i Liceum Akademickie w Toruniu
- **Artur Kozak**, 1 klasa, III LO im. Marynarki Wojennej RP w Gdyni
- **Paweł Kubiak**, 1 klasa, I Liceum Ogólnokształcące w Katowicach
- **Adam Kunysz**, 3 klasa, XII LO im. Bolesława Chrobrego we Wrocławiu
- **Alan Kutniewski**, 2 klasa, XIII Liceum Ogólnokształcące w Szczecinie
- **Mateusz Kwapich**, 3 klasa, VIII LO im. Marii Skłodowskiej w Katowicach
- **Jan Kwaśniak**, 2 klasa, LO im. Św. Jadwigi Królowej w Kielcach
- **Anna Lewicka**, 2 klasa, Zespół Szkół SRKAK w Chorzowie
- **Paweł Lipski**, 1 klasa, LO „Filomata” w Gliwicach
- **Jan Milczek**, 1 klasa, III LO im. Marynarki Wojennej RP w Gdyni
- **Marcin Niestrój**, 3 klasa, Publiczne LO nr 2 w Opolu
- **Adam Obuchowicz**, 2 klasa, I Liceum Ogólnokształcące w Zielonej Górze
- **Tomasz Obuchowski**, 2 klasa, I LO im. Adama Mickiewicza w Białymstoku
- **Maciej Piekarz**, 2 klasa, V LO im. Augusta Witkowskiego w Krakowie
- **Dawid Pustułka**, 3 klasa, V LO im. Augusta Witkowskiego w Krakowie
- **Rafał Ruciński**, 3 klasa, XIII Liceum Ogólnokształcące w Szczecinie
- **Kamil Sałaś**, 2 klasa, II LO im. Jana III Sobieskiego w Krakowie
- **Michał Sapalski**, 3 klasa, V LO im. Augusta Witkowskiego w Krakowie
- **Wojciech Sirko**, 3 klasa, XIV LO im. Stanisława Staszica w Warszawie
- **Mateusz Skowron**, 3 klasa, V LO im. Augusta Witkowskiego w Krakowie
- **Konrad Strack**, 3 klasa, V Liceum Ogólnokształcące w Bielsku–Białej
- **Adam Ścibior**, 2 klasa, VIII LO im. A. Mickiewicza w Poznaniu
- **Adam Śpiewak**, 2 klasa, Ogólnokształcące Liceum Jezuitów w Gdyni
- **Michał Trybus**, 3 klasa, V Liceum Ogólnokształcące w Gliwicach
- **Karol Trzcionka**, 3 klasa, VIII LO im. Marii Skłodowskiej w Katowicach
- **Jakub Tyrcha**, 3 klasa, V LO im. Augusta Witkowskiego w Krakowie
- **Paweł Walczak**, 2 klasa, III LO im. Marynarki Wojennej RP w Gdyni
- **Martyna Wałaszewska**, 2 klasa, VI LO im. J. i J. Śniadeckich w Bydgoszczy
- **Bartłomiej Wiśniewski**, 2 klasa, III LO im. Marynarki Wojennej RP w Gdyni
- **Jakub Witaszek**, 3 klasa, XIV LO im. Stanisława Staszica w Warszawie
- **Jakub Woyke**, 2 klasa, XIII Liceum Ogólnokształcące w Szczecinie
- **Krzysztof Zając**, 3 klasa, I LO im. Mikołaja Kopernika w Łodzi
- **Michał Zając**, 3 klasa gimnazjum, Publiczne Gimnazjum nr 2 w Brzesku

Komitet Główny Olimpiady Informatycznej przyznał następujące nagrody rzeczowe:

(1) puchar ufundowany przez Olimpiadę Informatyczną przyznano zwycięzcy XVI Olimpiady — Tomaszowi Kociumace,
(2) złote, srebrne i brązowe medale ufundowane przez Olimpiadę Informatyczną przyznano odpowiednio laureatom I, II i III miejsca,
(3) notebooki Toshiba (3 szt.) ufundowane przez Asseco Poland SA przyznano laureatom I miejsca i jednemu laureatowi II miejsca,
(4) netbooki Acer (10 szt.) ufundowane przez Asseco Poland SA przyznano laureatom II miejsca i trzem laureatom III miejsca,
(5) aparaty fotograficzne (11 szt.) ufundowane przez Ogólnopolską Fundację Edukacji Komputerowej i Olimpiadę Informatyczną przyznano następnym laureatom III miejsca,
(6) odtwarzacze mp4 (11 szt.) ufundowane przez Ogólnopolską Fundację Edukacji Komputerowej przyznano wyróżnionym finalistom,
(7) odtwarzacze mp3 (47 szt.) ufundowane przez Ogólnopolską Fundację Edukacji Komputerowej przyznano wszystkim finalistom,
(8) książki ufundowane przez Wydawnictwa Naukowo-Techniczne i PWN przyznano wszystkim finalistom,
(9) roczną prenumeratę miesięcznika „Delta" przyznano wszystkim laureatom.

Ogłoszono komunikat o powołaniu reprezentacji Polski na:

- **Międzynarodową Olimpiadę Informatyczną IOI'2009**, która odbędzie się w Bułgarii, w miejscowości Plovdiv, w terminie 08–15 sierpnia 2009 r.:

  (1) Tomasz Kociumaka
  (2) Jakub Pachocki
  (3) Jarosław Błasiok
  (4) Adam Karczmarz lub Jakub Oćwieja[1]

  rezerwowi:

  (5) Adam Karczmarz lub Jakub Oćwieja
  (6) Adrian Jaskółka

- **Olimpiadę Informatyczną Krajów Europy Środkowej CEOI'2009**, która odbędzie się w Rumunii, w miejscowości Targu Mures, w terminie 18–22 lipca 2009 r.:

  (1) Tomasz Kociumaka
  (2) Jakub Pachocki
  (3) Jarosław Błasiok
  (4) Adam Kaczmarski
  (5) Jakub Oćwieja — pojedzie jako zawodnik dodatkowy, zdecydowały o tym wyniki II etapu

[1] Dwóch zawodników zdobyło taką samą liczbę punktów w finale XVI Olimpiady Informatycznej. Decyzją Komitetu Głównego obydwaj zawodnicy wezmą udział w Olimpiadzie Informatycznej Krajów Europy Środkowej i wyniki tej olimpiady zdecydują, kto wejdzie w skład reprezentacji na Międzynarodową Olimpiadę Informatyczną.

rezerwowi:

(6) Adrian Jaskółka
(7) Jakub Adamek

- **Bałtycką Olimpiadę Informatyczną BOI'2009**, która odbędzie się w Szwecji, w Sztokholmie w terminie 18–22 kwietnia 2009 r.:

  (1) Jakub Pachocki
  (2) Adrian Jaskółka
  (3) Michał Makarewicz
  (4) Anna Piekarska
  (5) Dawid Dąbrowski
  (6) Karol Pokorski

  rezerwowi:

  (7) Grzegorz Prusak
  (8) Igor Adamski

- **Obóz czesko–polsko–słowacki**, który odbędzie się w Warszawie w terminie 22–27 czerwca 2009 r.:
  - reprezentacja na IOI i CEOI wraz z rezerwowymi.
- **Obóz im. A. Kreczmara**, który odbędzie się w Krynicy Górskiej, w terminie 19–31 lipca 2009 r.:
  - reprezentanci na międzynarodowe zawody informatyczne, zawodnicy rezerwowi oraz wszyscy laureaci i finaliści, którzy uczęszczają do klas niższych niż maturalna.

Sekretariat wystawił łącznie 24 zaświadczenia o uzyskaniu tytułu laureata, 11 zaświadczeń o uzyskaniu tytułu wyróżnionego finalisty oraz 48 zaświadczeń o uzyskaniu tytułu finalisty XVI Olimpiady Informatycznej.

Komitet Główny wyróżnił, za wkład pracy w przygotowanie finalistów Olimpiady Informatycznej, wszystkich podanych przez zawodników opiekunów naukowych:

- Mariusz Blank (Alcatel – Lucent, Bydgoszcz)
  - Artur Kraska — laureat III miejsca
- Jarosław Błasiok (zawodnik, Gostyń)
  - Tomasz Kociumaka — laureat I miejsca
  - Daniel Malinowski — finalista z wyróżnieniem
  - Mateusz Kwapich — finalista
- Krzysztof Błasiok (Gostyń)
  - Jarosław Błasiok — laureat II miejsca
- Florian Brom (V Liceum Ogólnokształcące, Gliwice)
  - Szymon Gut — finalista
  - Michał Trybus — finalista
- Irena Brzozowska (III Liceum Ogólnokształcące im. M. Dąbrowskiej, Płock)

  - Krzysztof Opolski — laureat III miejsca
- Ireneusz Bujnowski (I Liceum Ogólnokształcące im. A. Mickiewicza, Białystok)
  - Adrian Jaskółka — laureat II miejsca
  - Michał Makarewicz — laureat II miejsca
  - Robert Kozikowski — laureat III miejsca
  - Łukasz Jocz — finalista z wyróżnieniem
  - Adam Iwaniuk — finalista
  - Tomasz Obuchowski — finalista
- Czesław Drozdowski (XIII Liceum Ogólnokształcące, Szczecin)
  - Alan Kutniewski — finalista
  - Rafał Ruciński — finalista
  - Jakub Woyke — finalista
- Jarosław Drzeżdżon (Ogólnokształcące Liceum Jezuitów, Gdynia)
  - Adam Śpiewak — finalista
- Łukasz Dudek (student Uniwersytetu Jagiellońskiego, Kraków)
  - Kamil Sałaś — finalista
- Lech Duraj (Uniwersytet Jagielloński, Kraków)
  - Igor Adamski — laureat III miejsca
  - Michał Bejda — laureat III miejsca
  - Adam Polak — laureat III miejsca
  - Grzegorz Guśpiel — finalista z wyróżnieniem
  - Jacek Szmigiel — finalista z wyróżnieniem
  - Andrzej Dorobisz — finalista
  - Michał Sapalski — finalista
  - Maciej Piekarz — finalista
- Andrzej Dyrek (V Liceum Ogólnokształcące im. A. Witkowskiego, Kraków)
  - Tomasz Kłeczek — laureat II miejsca
  - Jakub Adamek — laureat II miejsca
  - Paweł Wanat — laureat III miejsca
  - Michał Bejda — laureat III miejsca
  - Adam Polak — laureat III miejsca
  - Igor Adamski — laureat III miejsca
  - Jacek Szmigiel — finalista z wyróżnieniem
  - Grzegorz Guśpiel — finalista z wyróżnieniem
  - Andrzej Dorobisz — finalista
  - Michał Sapalski — finalista
  - Mateusz Skowron — finalista
  - Maciej Piekarz — finalista
  - Dawid Pustułka — finalista
  - Jakub Tyrcha — finalista
- Alina Gościniak (VIII Liceum Ogólnokształcące, Poznań)
  - Michał Hordecki — finalista
- Adam Herman (I Liceum Ogólnokształcące im. K. Miarki, Żory)

  - Sławomir Fraś — laureat III miejsca
- Witold Jarnicki (Uniwersytet Jagielloński, Kraków)
  - Kamil Sałaś — finalista
- Tomasz Kociumaka (zawodnik)
  - Daniel Malinowski — finalista z wyróżnieniem
- Ewa Kowalska–Zając (Łódź)
  - Krzysztof Zając — finalista
- Wojciech Kolarz (Zespół Szkół Stowarzyszenia Rodzin Katolickich Archidiecezji Katowickiej, Chorzów)
  - Anna Lewicka — finalistka
- Sławomir Krzywicki (I Liceum Ogólnokształcące, Zielona Góra)
  - Wiktor Jakubiuk — finalista
- Roman Kula (Zespół Szkół Ogólnokształcących nr 1, Katowice)
  - Paweł Kubiak — finalista
- Władysław Kwaśnicki (student Instytutu Informatyki Uniwersytetu Wrocławskiego)
  - Wiktor Janas — laureat II miejsca
- Barbara Lipska (Politechnika Śląska, Gliwice)
  - Paweł Lipski — finalista
- Ryszard Lisoń (II Liceum Ogólnokształcące im. M. Konopnickiej, Opole)
  - Marcin Niestrój — finalista
- Krzysztof Magiera (student Akademii Górniczo–Hutniczej, Kraków)
  - Michał Zając — finalista
- Mirosław Mortka (VI Liceum Ogólnokształcące im. J. Kochanowskiego, Radom)
  - Piotr Godlewski — finalista z wyróżnieniem
- Łucja Mularska (Liceum Ogólnokształcące, Bełchatów)
  - Marcin Cyran — finalista
- Rafał Nowak (Instytut Informatyki Uniwersytetu Wrocławskiego)
  - Anna Piekarska — laureatka III miejsca
  - Krzysztof Król — finalista z wyróżnieniem
  - Bartłomiej Dudek — finalista
  - Karol Konaszyński — finalista
- Andrzej Obuchowicz (Uniwersytet Zielonogórski, Zielona Góra)
  - Adam Obuchowicz — finalista
- Małgorzata Piekarska (VI Liceum Ogólnokształcące im. J. i J. Śniadeckich, Bydgoszcz)
  - Janusz Kudełka — finalista z wyróżnieniem
  - Maciej Borsz — finalista
  - Konrad Baumgart — finalista
  - Martyna Wałaszewska — finalistka
- Włodzimierz Raczek (V Liceum Ogólnokształcące, Bielsko–Biała)
  - Jakub Oćwieja — laureat II miejsca

  - Konrad Strack — finalista
- Jadwiga Roguska (Gimnazjum i Liceum Akademickie, Toruń)
  - Piotr Szefler — laureat III miejsca
- Bartosz Szreder (student Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Marcin Sebastian Baczyński — finalista
  - Krzysztof Kiewicz — finalista
- Ryszard Szubartowski (III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia)
  - Jakub Pachocki — laureat I miejsca
  - Dawid Dąbrowski — laureat III miejsca
  - Karol Pokorski — laureat III miejsca
  - Mateusz Litwin — finalista z wyróżnieniem
  - Radosław Dembkowski — finalista
  - Artur Kozak — finalista
  - Paweł Walczak — finalista
  - Bartłomiej Wiśniewski — finalista
- Joanna Śmigielska (XIV Liceum Ogólnokształcące im. S. Staszica, Warszawa)
  - Marcin Wrochna — laureat III miejsca
  - Piotr Suwara — finalista z wyróżnieniem
  - Arkadiusz Betkier — finalista
  - Wojciech Sirko — finalista
  - Jakub Witaszek — finalista
- Henryk Trzcionka (Mikołów)
  - Karol Trzcionka — finalista
- Paweł Walter (Google Polska, Kraków)
  - Jakub Adamek — laureat II miejsca
  - Michał Bejda — laureat III miejsca
  - Adam Polak — laureat III miejsca
  - Paweł Wanat — laureat III miejsca
  - Jacek Szmigiel — finalista z wyróżnieniem
  - Dawid Pustułka — finalista
  - Michał Sapalski — finalista
  - Jakub Tyrcha — finalista
- Filip Wolski (student Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Radosław Dembkowski — finalista
  - Artur Kozak — finalista

Zgodnie z decyzją Komitetu Głównego z dn. 3 kwietnia 2009 r., opiekunowie naukowi laureatów i finalistów, będący nauczycielami szkół, otrzymają nagrody pieniężne.

Podobnie jak w ubiegłych latach w przygotowaniu jest publikacja zawierająca pełną informację o XVI Olimpiadzie Informatycznej, zadania konkursowe oraz wzorcowe rozwiązania. W publikacji tej znajdą się także zadania z międzynarodowych zawodów informatycznych.

Programy wzorcowe w postaci elektronicznej i testy użyte do sprawdzania rozwiązań zawodników będą dostępne na stronie Olimpiady Informatycznej: `www.oi.edu.pl`.

Warszawa, 26 czerwca 2009 roku

Regulamin Olimpiady Informatycznej

§1 WSTĘP

Olimpiada Informatyczna, zwana dalej Olimpiadą, jest olimpiadą przedmiotową powołaną przez Instytut Informatyki Uniwersytetu Wrocławskiego, w dniu 10 grudnia 1993 roku. Olimpiada działa zgodnie z Rozporządzeniem Ministra Edukacji Narodowej i Sportu z 29 stycznia 2002 roku w sprawie organizacji oraz sposobu przeprowadzenia konkursów, turniejów i olimpiad (Dz. U. 02.13.125). Organizatorem Olimpiady Informatycznej jest Uniwersytet Wrocławski. W organizacji Olimpiady Uniwersytet Wrocławski współdziała ze środowiskami akademickimi, zawodowymi i oświatowymi działającymi w sprawach edukacji informatycznej.

§2 CELE OLIMPIADY I SPOSOBY ICH OSIĄGANIA

(1) Stworzenie motywacji dla zainteresowania nauczycieli i uczniów informatyką.

(2) Rozszerzanie współdziałania nauczycieli akademickich z nauczycielami szkół w kształceniu młodzieży uzdolnionej.

(3) Stymulowanie aktywności poznawczej młodzieży informatycznie uzdolnionej.

(4) Kształtowanie umiejętności samodzielnego zdobywania i rozszerzania wiedzy informatycznej.

(5) Stwarzanie młodzieży możliwości szlachetnego współzawodnictwa w rozwijaniu swoich uzdolnień, a nauczycielom — warunków twórczej pracy z młodzieżą.

(6) Wyłanianie reprezentacji Rzeczypospolitej Polskiej na Międzynarodową Olimpiadę Informatyczną i inne międzynarodowe zawody informatyczne.

(7) Cele Olimpiady są osiągane poprzez:

- organizację olimpiady przedmiotowej z informatyki dla uczniów szkół ponadgimnazjalnych;
- powołanie i organizację ogólnopolskiego konkursu informatycznego dla gimnazjalistów — Olimpiady Informatycznej Gimnazjalistów;
- organizowanie corocznych obozów naukowych dla wyróżniających się uczestników olimpiad;
- organizowanie warsztatów treningowych dla nauczycieli zainteresowanych przygotowywaniem uczniów do udziału w olimpiadach;
- przygotowywanie i publikowanie materiałów edukacyjnych dla uczniów zainteresowanych udziałem w olimpiadach i ich nauczycieli.

§3 ORGANIZACJA OLIMPIADY

(1) Olimpiadę przeprowadza Komitet Główny Olimpiady Informatycznej, zwany dalej Komitetem.

(2) Olimpiada jest trójstopniowa.

(3) W Olimpiadzie mogą brać indywidualnie udział uczniowie wszystkich typów szkół ponadgimnazjalnych i szkół średnich dla młodzieży, dających możliwość uzyskania matury.

(4) W Olimpiadzie mogą również uczestniczyć — za zgodą Komitetu Głównego — uczniowie szkół podstawowych, gimnazjów, zasadniczych szkół zawodowych i szkół zasadniczych.

(5) Zawody I stopnia mają charakter otwarty i polegają na samodzielnym rozwiązywaniu przez uczestnika zadań ustalonych dla tych zawodów oraz przekazaniu rozwiązań w podanym terminie; miejsce i sposób przekazania określone są w „Zasadach organizacji zawodów", zwanych dalej Zasadami.

(6) Zawody II stopnia są organizowane przez komitety okręgowe Olimpiady lub instytucje upoważnione przez Komitet Główny.

(7) Zawody II i III stopnia polegają na samodzielnym rozwiązywaniu zadań. Zawody te odbywają się w ciągu dwóch sesji, przeprowadzanych w różnych dniach, w warunkach kontrolowanej samodzielności. Zawody poprzedzone są sesją próbną, której rezultaty nie liczą się do wyników zawodów.

(8) Liczbę uczestników kwalifikowanych do zawodów II i III stopnia ustala Komitet Główny i podaje ją w Zasadach.

(9) Komitet Główny kwalifikuje do zawodów II i III stopnia odpowiednią liczbę uczestników, których rozwiązania zadań stopnia niższego zostaną ocenione najwyżej. Zawodnicy zakwalifikowani do zawodów III stopnia otrzymują tytuł finalisty Olimpiady Informatycznej.

(10) Rozwiązaniem zadania zawodów I, II i III stopnia są, zgodnie z treścią zadania, dane lub program. Program powinien być napisany w języku programowania i środowisku wybranym z listy języków i środowisk ustalanej przez Komitet Główny i ogłaszanej w Zasadach.

(11) Rozwiązania są oceniane automatycznie. Jeśli rozwiązaniem zadania jest program, to jest on uruchamiany na testach z przygotowanego zestawu.
Podstawą oceny jest zgodność sprawdzanego programu z podaną w treści zadania specyfikacją, poprawność wygenerowanego przez program wyniku oraz czas działania tego programu. Jeśli rozwiązaniem zadania jest plik z danymi, wówczas ocenia się poprawność danych i na tej podstawie przyznaje punkty.

(12) Komitet Główny zastrzega sobie prawo do opublikowania rozwiązań zawodników, którzy zostali zakwalifikowani do następnego etapu, zostali wyróżnieni lub otrzymali tytuł laureata.

(13) Rozwiązania zespołowe, niesamodzielne, niezgodne z „Zasadami organizacji zawodów" lub takie, co do których nie można ustalić autorstwa, nie będą oceniane.

(14) Każdy zawodnik jest zobowiązany do zachowania w tajemnicy swoich rozwiązań w czasie trwania zawodów.

(15) W szczególnie rażących wypadkach łamania Regulaminu i Zasad Komitet Główny może zdyskwalifikować zawodnika.

(16) Komitet Główny przyjął następujący tryb opracowywania zadań olimpijskich:

   (a) Autor zgłasza propozycję zadania, które powinno być oryginalne i nieznane, do sekretarza naukowego Olimpiady.

   (b) Zgłoszona propozycja zadania jest opiniowana, redagowana, opracowywana wraz z zestawem testów, a opracowania podlegają niezależnej weryfikacji. Zadanie, które uzyska negatywną opinię, może zostać odrzucone lub skierowane do ponownego opracowania.

   (c) Wyboru zestawu zadań na zawody dokonuje Komitet Główny, spośród zadań, które zostały opracowane i uzyskały pozytywną opinię.

   (d) Wszyscy uczestniczący w procesie przygotowania zadania są zobowiązani do zachowania tajemnicy do czasu jego wykorzystania w zawodach lub ostatecznego odrzucenia.

§4 KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

(1) Komitet Główny jest odpowiedzialny za poziom merytoryczny i organizację zawodów. Komitet składa corocznie organizatorowi sprawozdanie z przeprowadzonych zawodów.

(2) W skład Komitetu wchodzą nauczyciele akademiccy, nauczyciele szkół ponadgimnazjalnych i ponadpodstawowych oraz pracownicy oświaty związani z kształceniem informatycznym.

(3) Komitet wybiera ze swego grona Prezydium na kadencję trzyletnią. Prezydium podejmuje decyzje w nagłych sprawach pomiędzy posiedzeniami Komitetu. W skład Prezydium wchodzą w szczególności: przewodniczący, dwóch wiceprzewodniczących, sekretarz naukowy, kierownik Jury, kierownik techniczny i kierownik organizacyjny.

(4) Komitet dokonuje zmian w swoim składzie za zgodą Organizatora.

(5) Komitet powołuje i rozwiązuje komitety okręgowe Olimpiady.

(6) Komitet:

   (a) opracowuje szczegółowe „Zasady organizacji zawodów", które są ogłaszane razem z treścią zadań zawodów I stopnia Olimpiady;

   (b) udziela wyjaśnień w sprawach dotyczących Olimpiady;

   (c) ustala listy laureatów i wyróżnionych uczestników oraz kolejność lokat;

   (d) przyznaje uprawnienia i nagrody rzeczowe wyróżniającym się uczestnikom Olimpiady;

   (e) ustala skład reprezentacji na Międzynarodową Olimpiadę Informatyczną i inne międzynarodowe zawody informatyczne.

(7) Decyzje Komitetu zapadają zwykłą większością głosów uprawnionych przy udziale przynajmniej połowy członków Komitetu. W przypadku równej liczby głosów decyduje głos przewodniczącego obrad.

(8) Posiedzenia Komitetu, na których ustala się treści zadań Olimpiady, są tajne. Przewodniczący obrad może zarządzić tajność obrad także w innych uzasadnionych przypadkach.

(9) Do organizacji zawodów II stopnia w miejscowościach, których nie obejmuje żaden komitet okręgowy, Komitet powołuje komisję zawodów co najmniej miesiąc przed terminem rozpoczęcia zawodów.

(10) Decyzje Komitetu we wszystkich sprawach dotyczących przebiegu i wyników zawodów są ostateczne.

(11) Komitet dysponuje funduszem Olimpiady za pośrednictwem kierownika organizacyjnego Olimpiady.

(12) Komitet zatwierdza plan finansowy dla każdej edycji Olimpiady na pierwszym posiedzeniu Komitetu w nowym roku szkolnym.

(13) Komitet przyjmuje sprawozdanie finansowe z każdej edycji Olimpiady w ciągu czterech miesięcy od zakończenia danej edycji.

(14) Komitet ma siedzibę w Ośrodku Edukacji Informatycznej i Zastosowań Komputerów w Warszawie. Ośrodek wspiera Komitet we wszystkich działaniach organizacyjnych zgodnie z Deklaracją z 8 grudnia 1993 roku przekazaną Organizatorowi.

(15) Pracami Komitetu kieruje przewodniczący, a w jego zastępstwie lub z jego upoważnienia jeden z wiceprzewodniczących.

(16) Kierownik Jury w porozumieniu z Przewodniczącym powołuje i odwołuje członków Jury Olimpiady, które jest odpowiedzialne za opracowanie wzorcowych rozwiązań i sprawdzanie zadań.

(17) Kierownik techniczny odpowiada za stronę techniczną przeprowadzenia zawodów.

(18) Przewodniczący:

   (a) czuwa nad całokształtem prac Komitetu;

   (b) zwołuje posiedzenia Komitetu;

   (c) przewodniczy tym posiedzeniom;

   (d) reprezentuje Komitet na zewnątrz;

   (e) czuwa nad prawidłowością wydatków związanych z organizacją i przeprowadzeniem Olimpiady oraz zgodnością działalności Komitetu z przepisami.

(19) Komitet prowadzi archiwum akt Olimpiady, przechowując w nim między innymi:

   (a) zadania Olimpiady;

   (b) rozwiązania zadań Olimpiady przez okres 2 lat;

   (c) rejestr wydanych zaświadczeń i dyplomów laureatów;

   (d) listy laureatów i ich nauczycieli;

(e) dokumentację statystyczną i finansową.

(20) W jawnych posiedzeniach Komitetu mogą brać udział przedstawiciele organizacji wspierających, jako obserwatorzy z głosem doradczym.

§5 KOMITETY OKRĘGOWE

(1) Komitet okręgowy składa się z przewodniczącego, jego zastępcy, sekretarza i co najmniej dwóch członków.

(2) Zmiany w składzie komitetu okręgowego są dokonywane przez Komitet.

(3) Zadaniem komitetów okręgowych jest organizacja zawodów II stopnia oraz popularyzacja Olimpiady.

§6 PRZEBIEG OLIMPIADY

(1) Komitet rozsyła do szkół wymienionych w § 3.3 oraz kuratoriów oświaty i koordynatorów edukacji informatycznej informację o przebiegu danej edycji Olimpiady wraz z Zasadami.

(2) W czasie rozwiązywania zadań w zawodach II i III stopnia każdy uczestnik ma do swojej dyspozycji komputer.

(3) Rozwiązywanie zadań Olimpiady w zawodach II i III stopnia jest poprzedzone jednodniowymi sesjami próbnymi umożliwiającymi zapoznanie się uczestników z warunkami organizacyjnymi i technicznymi Olimpiady.

(4) Komitet zawiadamia uczestnika oraz dyrektora jego szkoły o zakwalifikowaniu do zawodów stopnia II i III, podając jednocześnie miejsce i termin zawodów.

(5) Uczniowie powołani do udziału w zawodach II i III stopnia są zwolnieni z zajęć szkolnych na czas niezbędny do udziału w zawodach, a także otrzymują bezpłatne zakwaterowanie i wyżywienie oraz zwrot kosztów przejazdu.

§7 UPRAWNIENIA I NAGRODY

(1) Laureaci i finaliści Olimpiady otrzymują z informatyki lub technologii informacyjnej celującą roczną (semestralną) ocenę klasyfikacyjną.

(2) Laureaci i finaliści Olimpiady są zwolnieni z egzaminu maturalnego z informatyki. Uprawnienie to przysługuje także wtedy, gdy przedmiot nie był objęty szkolnym planem nauczania danej szkoły.

(3) Uprawnienia określone w punktach 1. i 2. przysługują na zasadach określonych w rozporządzeniu MENiS z 7 września 2004 r. w sprawie warunków i sposobu oceniania, klasyfikowania i promowania uczniów i słuchaczy oraz przeprowadzania sprawdzianów i egzaminów w szkołach publicznych. (Dz. U. z 2004 r. Nr 199, poz. 2046, § 18 i § 56).

(4) Laureaci i finaliści Olimpiady mają ułatwiony lub wolny wstęp do tych szkół wyższych, których senaty podjęły uchwały w tej sprawie, zgodnie z przepisami ustawy z 12 września 1990 r. o szkolnictwie wyższym, na zasadach zawartych w tych uchwałach (Dz. U. z 1990 r. Nr 65 poz. 385, Art. 141).

(5) Zaświadczenia o uzyskanych uprawnieniach wydaje uczestnikom Komitet. Zaświadczenia podpisuje przewodniczący Komitetu. Komitet prowadzi rejestr wydanych zaświadczeń.

(6) Nauczyciel, którego praca przy przygotowaniu uczestnika Olimpiady zostanie oceniona przez Komitet jako wyróżniająca, otrzymuje nagrodę pieniężną wypłacaną z funduszu Olimpiady.

(7) Komitet przyznaje wyróżniającym się aktywnością członkom Komitetu i komitetów okręgowych nagrody pieniężne z funduszu Olimpiady.

(8) Osobom, które wniosły szczególnie duży wkład w rozwój Olimpiady Informatycznej, Komitet może przyznać honorowy tytuł „Zasłużony dla Olimpiady Informatycznej".

§8 FINANSOWANIE OLIMPIADY

Komitet będzie się ubiegał o pozyskanie środków finansowych z budżetu państwa, składając wniosek w tej sprawie do Ministra Edukacji Narodowej i przedstawiając przewidywany plan finansowy organizacji Olimpiady na dany rok. Komitet będzie także zabiegał o pozyskanie dotacji z innych organizacji wspierających.

§9 OLIMPIADA INFORMATYCZNA GIMNAZJALISTÓW

(1) Olimpiada Informatyczna Gimnazjalistów jest ogólnopolskim konkursem informatycznym przeprowadzanym za zgodą i pod nadzorem Komitetu Głównego Olimpiady Informatycznej.

(2) Olimpiada Informatyczna Gimnazjalistów jest przeprowadzana na podstawie Regulaminu Olimpiady Informatycznej Gimnazjalistów, zatwierdzonego przez Komitet Główny Olimpiady Informatycznej i stanowiącego załącznik do niniejszego regulaminu.

(3) Finansowanie Olimpiady Informatycznej Gimnazjalistów odbywa się ze środków pozyskiwanych wyłącznie na ten cel, bez uszczuplania środków przeznaczonych na finansowanie Olimpiady.

§10 PRZEPISY KOŃCOWE

(1) Koordynatorzy edukacji informatycznej i dyrektorzy szkół mają obowiązek dopilnowania, aby wszystkie wytyczne oraz informacje dotyczące Olimpiady zostały podane do wiadomości uczniów.

(2) Komitet zatwierdza sprawozdanie merytoryczne z przeprowadzonej edycji Olimpiady w ciągu 3 miesięcy po zakończeniu zawodów III stopnia i przedstawia je Organizatorowi i Ministerstwu Edukacji Narodowej.

(3) Niniejszy regulamin może być zmieniony przez Komitet tylko przed rozpoczęciem kolejnej edycji zawodów Olimpiady po zatwierdzeniu zmian przez Organizatora.

Warszawa, 19 września 2008 r.

Zasady organizacji zawodów w roku szkolnym 2008/2009

Podstawowym aktem prawnym dotyczącym Olimpiady jest Regulamin Olimpiady Informatycznej, którego pełny tekst znajduje się w kuratoriach. Poniższe zasady są uzupełnieniem tego Regulaminu, zawierającym szczegółowe postanowienia Komitetu Głównego Olimpiady Informatycznej o jej organizacji w roku szkolnym 2008/2009.

§1 WSTĘP

Olimpiada Informatyczna jest olimpiadą przedmiotową powołaną przez Instytut Informatyki Uniwersytetu Wrocławskiego 10 grudnia 1993 roku. Olimpiada działa zgodnie z Rozporządzeniem Ministra Edukacji Narodowej i Sportu z 29 stycznia 2002 roku w sprawie organizacji oraz sposobu przeprowadzenia konkursów, turniejów i olimpiad (Dz. U. 02.13.125). Organizatorem Olimpiady Informatycznej jest Uniwersytet Wrocławski. W organizacji Olimpiady Uniwersytet Wrocławski współdziała ze środowiskami akademickimi, zawodowymi i oświatowymi działającymi w sprawach edukacji informatycznej.

§2 ORGANIZACJA OLIMPIADY

(1) Olimpiadę przeprowadza Komitet Główny Olimpiady Informatycznej zwany dalej Komitetem Głównym.

(2) Olimpiada Informatyczna jest trójstopniowa.

(3) W Olimpiadzie Informatycznej mogą brać indywidualnie udział uczniowie wszystkich typów szkół ponadgimnazjalnych. W Olimpiadzie mogą również uczestniczyć — za zgodą Komitetu Głównego — uczniowie szkół podstawowych i gimnazjów.

(4) Rozwiązaniem każdego z zadań zawodów I, II i III stopnia jest program (napisany w jednym z następujących języków programowania: *Pascal*, *C*, *C++* lub *Java*) lub plik z danymi.

(5) Zawody I stopnia mają charakter otwarty i polegają na samodzielnym rozwiązywaniu zadań i nadesłaniu rozwiązań w podanym terminie i we wskazane miejsce.

(6) Zawody II i III stopnia polegają na rozwiązywaniu zadań w warunkach kontrolowanej samodzielności. Zawody te odbywają się w ciągu dwóch sesji, przeprowadzanych w różnych dniach.

(7) Do zawodów II stopnia zostanie zakwalifikowanych 400 uczestników, których rozwiązania zadań I stopnia zostaną ocenione najwyżej; do zawodów III stopnia — 70 uczestników, których rozwiązania zadań II stopnia zostaną ocenione najwyżej. Komitet Główny może zmienić podane liczby zakwalifikowanych uczestników co najwyżej o 20%.

(8) Podjęte przez Komitet Główny decyzje o zakwalifikowaniu uczestników do zawodów kolejnego stopnia, zajętych miejscach i przyznanych nagrodach oraz składzie polskiej reprezentacji na Międzynarodową Olimpiadę Informatyczną i inne międzynarodowe zawody informatyczne są ostateczne.

(9) Komitet Główny zastrzega sobie prawo do opublikowania rozwiązań zawodników, którzy zostali zakwalifikowani do następnego etapu, zostali wyróżnieni lub otrzymali tytuł laureata.

(10) Terminarz zawodów:
- zawody I stopnia — 20.10–17.11.2008 r.
  ogłoszenie wyników:
  - w witrynie Olimpiady — 12.12.2008 r.,
  - pocztą — 29.12.2008 r.
- zawody II stopnia — 10–12.02.2009 r.
  ogłoszenie wyników:
  - w witrynie Olimpiady — 20.02.2009 r.
  - pocztą — 05.03.2009 r.
- zawody III stopnia — 31.03–04.04.2009 r.

§3 ROZWIĄZANIA ZADAŃ

(1) Ocena rozwiązania zadania jest określana na podstawie wyników testowania programu i uwzględnia poprawność oraz efektywność metody rozwiązania użytej w programie.

(2) Rozwiązania zespołowe, niesamodzielne, niezgodne z „Zasadami organizacji zawodów" lub takie, co do których nie można ustalić autorstwa, nie będą oceniane.

(3) Każdy zawodnik jest zobowiązany do zachowania w tajemnicy swoich rozwiązań w czasie trwania zawodów.

(4) Rozwiązanie każdego zadania, które polega na napisaniu programu, składa się z (tylko jednego) pliku źródłowego; imię i nazwisko uczestnika muszą być podane w komentarzu na początku każdego programu.

(5) Nazwy plików z programami w postaci źródłowej muszą być takie jak podano w treści zadania. Nazwy tych plików muszą mieć następujące rozszerzenia zależne od użytego języka programowania:

Pascal	`pas`
C	`c`
C++	`cpp`
Java	`java`

(6) Programy w *C*/*C++* będą kompilowane w systemie Linux za pomocą kompilatora GCC/G++ v. 4.1.1. Programy w *Pascalu* będą kompilowane w systemie Linux za pomocą kompilatora FreePascal v. 2.2.2. Programy w *Javie* będą kompilowane w systemie Linux za pomocą kompilatora z Sun JDK 6 Update 2. Wybór polecenia kompilacji zależy od podanego rozszerzenia pliku w następujący sposób (np. dla zadania *abc*):

Dla `c`	`gcc -O2 -static abc.c -lm`
Dla `cpp`	`g++ -O2 -static abc.cpp -lm`
Dla `pas`	`ppc386 -O2 -XS -Xt abc.pas`
Dla `java`	`javac abc.java`

(7) Programy w Javie będą uruchamiane w systemie Linux za pomocą maszyny wirtualnej z Sun JDK 6 Update 2. Dla zadania *abc* klasa publiczna pliku źródłowego w Javie musi nosić nazwę `abc`. Uruchomieniu podlega treść publicznej funkcji statycznej `main(String[] args)` tej klasy.

(8) Program powinien odczytywać dane wejściowe ze standardowego wejścia i zapisywać dane wyjściowe na standardowe wyjście, chyba że dla danego zadania wyraźnie napisano inaczej.

(9) Należy przyjąć, że dane testowe są bezbłędne, zgodne z warunkami zadania i podaną specyfikacją wejścia.

§4 ZAWODY I STOPNIA

(1) Zawody I stopnia polegają na samodzielnym rozwiązywaniu podanych zadań (niekoniecznie wszystkich) i przesłaniu rozwiązań do Komitetu Głównego Olimpiady Informatycznej. Możliwe są tylko dwa sposoby przesyłania:

- poprzez System Internetowy Olimpiady o adresie: `http://sio.mimuw.edu.pl` do godziny 12:00 (w południe) dnia 17 listopada 2008 roku. Komitet Główny nie ponosi odpowiedzialności za brak możliwości przekazania rozwiązań przez Internet w sytuacji nadmiernego obciążenia lub awarii systemu. Odbiór przesyłki zostanie potwierdzony przez system zwrotnym listem elektronicznym (prosimy o zachowanie tego listu). Brak potwierdzenia może oznaczać, że rozwiązanie nie zostało poprawnie zarejestrowane. W tym przypadku zawodnik powinien przesłać swoje rozwiązanie przesyłką poleconą za pośrednictwem zwykłej poczty. Szczegóły dotyczące sposobu postępowania przy przekazywaniu zadań i związanej z tym rejestracji będą podane w witrynie.
- pocztą, przesyłką poleconą, na adres:

  Olimpiada Informatyczna
  Ośrodek Edukacji Informatycznej i Zastosowań Komputerów
  ul. Nowogrodzka 73
  02-006 Warszawa
  tel. (0-22) 626-83-90

  w nieprzekraczalnym terminie nadania do 17 listopada 2008 r. (decyduje data stempla pocztowego). Uczestnik ma obowiązek zachować dowód nadania przesyłki do czasu otrzymania wyników oceny. Nawet w przypadku wysyłania rozwiązań pocztą, każdy uczestnik musi założyć sobie konto w Systemie Internetowym Olimpiady. **Zarejestrowana nazwa użytkownika musi być zawarta w przesyłce.**

Rozwiązania dostarczane w inny sposób nie będą przyjmowane. W przypadku jednoczesnego zgłoszenia rozwiązania danego zadania przez Internet i listem poleconym, ocenie podlega jedynie rozwiązanie wysłane listem poleconym.

(2) Uczestnik korzystający z poczty zwykłej przysyła nośnik (dyskietkę lub CD-ROM) w standardzie dla komputerów PC zawierający:

- spis zawartości nośnika oraz nazwę użytkownika z Systemu Internetowego Olimpiady w pliku nazwanym `SPIS.TXT`,
- do każdego rozwiązanego zadania — program źródłowy lub plik z danymi.

Na nośniku nie powinno być żadnych podkatalogów.
W przypadku braku możliwości odczytania nośnika z rozwiązaniami, nieodczytane rozwiązania nie będą brane pod uwagę.

(3) Uczestnik korzystający z Systemu Internetowego Olimpiady postępuje zgodnie z instrukcjami umieszczonymi w witrynie systemu. W szczególności, warunkiem koniecznym do kwalifikacji zawodnika do dalszych etapów jest podanie w systemie wszystkich wymaganych danych osobowych.

(4) Każdy uczestnik powinien założyć w witrynie Systemu Internetowego Olimpiady dokładnie jedno konto. Zawodnicy korzystający z wielu kont mogą zostać zdyskwalifikowani.

(5) W Systemie Internetowym Olimpiady znajdują się *Odpowiedzi na pytania zawodników* dotyczące Olimpiady. Ponieważ *Odpowiedzi* mogą zawierać ważne informacje dotyczące toczących się zawodów, wszyscy zawodnicy proszeni są o regularne zapoznawanie się z ukazującymi się odpowiedziami. Dalsze pytania należy przysyłać poprzez System Internetowy Olimpiady. Komitet Główny może nie udzielić odpowiedzi na pytanie z ważnych przyczyn, m.in. gdy jest ono niejednoznaczne lub dotyczy sposobu rozwiązania zadania.

(6) Przez witrynę Systemu Internetowego Olimpiady udostępniane są narzędzia do sprawdzania rozwiązań pod względem formalnym. Szczegóły dotyczące sposobu postępowania są dokładnie podane w witrynie.

(7) Od 01.12.2008 r. poprzez System Internetowy Olimpiady każdy zawodnik będzie mógł zapoznać się ze wstępną oceną swojej pracy.

(8) Do 05.12.2008 r. (włącznie) poprzez System Internetowy Olimpiady każdy zawodnik będzie mógł zgłaszać uwagi do wstępnej oceny swoich rozwiązań. Reklamacji nie podlega jednak dobór testów, limitów czasowych, kompilatorów i sposobu oceny.

(9) Reklamacje złożone po 05.12.2008 r. nie będą rozpatrywane.

§5 ZAWODY II I III STOPNIA

(1) Zawody II i III stopnia Olimpiady Informatycznej polegają na samodzielnym rozwiązywaniu zadań w ciągu dwóch pięciogodzinnych sesji odbywających się w różnych dniach.

(2) Rozwiązywanie zadań konkursowych poprzedzone jest trzygodzinną sesją próbną umożliwiającą uczestnikom zapoznanie się z warunkami organizacyjnymi i technicznymi Olimpiady. Wyniki sesji próbnej nie są liczone do klasyfikacji.

(3) W czasie rozwiązywania zadań konkursowych każdy uczestnik ma do swojej dyspozycji komputer z systemem Linux. Zawodnikom wolno korzystać wyłącznie ze sprzętu i oprogramowania dostarczonego przez organizatora. Stanowiska są przydzielane losowo.

(4) Komisja Regulaminowa powołana przez Komitet Główny czuwa nad prawidłowością przebiegu zawodów i pilnuje przestrzegania Regulaminu Olimpiady i Zasad Organizacji Zawodów.

(5) Zawody II i III stopnia są przeprowadzane za pomocą Serwisu Internetowego Olimpiady zwanego dalej SIO.

(6) Na sprawdzenie kompletności oprogramowania i poprawności konfiguracji sprzętu jest przeznaczone 45 minut przed rozpoczęciem sesji próbnej. W tym czasie wszystkie zauważone braki powinny zostać usunięte. Jeżeli nie wszystko uda się poprawić w tym czasie, rozpoczęcie sesji próbnej w tej sali może się opóźnić.

(7) W przypadku stwierdzenia awarii sprzętu w czasie zawodów, termin zakończenia pracy przez uczestnika zostaje odpowiednio przedłużony.

(8) W czasie trwania zawodów nie można korzystać z żadnych książek ani innych pomocy takich jak: dyski, kalkulatory, notatki itp. Nie wolno mieć na stanowisku komputerowym telefonu komórkowego ani innych własnych urządzeń elektronicznych.

(9) W ciągu pierwszej godziny każdej sesji nie wolno opuszczać przydzielonej sali zawodów. Zawodnicy spóźnieni więcej niż godzinę nie będą w tym dniu dopuszczeni do zawodów.

(10) W ciągu pierwszej godziny każdej sesji uczestnik może poprzez SIO zadawać pytania, na które otrzymuje jedną z odpowiedzi: *tak*, *nie*, *niepoprawne pytanie*, *odpowiedź wynika z treści zadania* lub *bez odpowiedzi*. Pytania mogą dotyczyć jedynie treści zadań.

(11) W czasie przeznaczonym na rozwiązywanie zadań jakikolwiek inny sposób komunikowania się z członkami Jury co do treści i sposobów rozwiązywania zadań jest niedopuszczalny.

(12) Komunikowanie się z innymi uczestnikami Olimpiady (np. ustnie, telefonicznie lub poprzez sieć) w czasie przeznaczonym na rozwiązywanie zadań jest zabronione pod rygorem dyskwalifikacji.

(13) Każdy zawodnik ma prawo wydrukować wyniki swojej pracy w sposób podany przez organizatorów.

(14) Każdy zawodnik powinien umieścić ostateczne rozwiązania zadań w SIO. Po zgłoszeniu rozwiązania każdego z zadań SIO dokona wstępnego sprawdzenia i udostępni jego wyniki zawodnikowi. Wstępne sprawdzenie polega na uruchomieniu programu zawodnika na testach przykładowych (wyniki sprawdzenia tych testów nie są liczone do końcowej klasyfikacji). Te same testy przykładowe są używane do wstępnego sprawdzenia w trybie weryfikacji rozwiązań na komputerze zawodnika.

(15) Każde zadanie można zgłosić w SIO co najwyżej 10 razy. Spośród tych zgłoszeń ocenianie jest jedynie najpóźniejsze.

(16) Jeżeli zawodnik nie zgłosił swoich rozwiązań w SIO, powinien je pozostawić w katalogu wskazanym przez organizatorów i niezwłocznie po zakończeniu sesji a przed opuszczeniem sali zawodów wręczyć pisemne oświadczenie dyżurującemu w tej sali członkowi Komisji Regulaminowej. Oświadczenie to musi zawierać imię i nazwisko zawodnika oraz numer stanowiska. Złożenie takiego oświadczenia powoduje, że rozwiązanie złożone wcześniej w SIO nie będzie rozpatrywane.

(17) W sprawach spornych decyzje podejmuje Jury Odwoławcze, złożone z jurora niezaangażowanego w rozważaną kwestię i wyznaczonego członka Komitetu Głównego. Decyzje w sprawach o wielkiej wadze (np. dyskwalifikacji) Jury Odwoławcze podejmuje w porozumieniu z przewodniczącym Komitetu Głównego.

(18) Każdego dnia zawodów około 2 godziny po zakończeniu sesji zawodnicy otrzymają raporty oceny swoich prac na niepełnym zestawie testów. Od tego momentu przez godzinę będzie czas na reklamację tej oceny, a w szczególności na reklamację wyboru rozwiązania, które ma podlegać ocenie.

§6 UPRAWNIENIA I NAGRODY

(1) Laureaci i finaliści Olimpiady otrzymują z informatyki lub technologii informacyjnej celującą roczną (semestralną) ocenę klasyfikacyjną.

(2) Laureaci i finaliści Olimpiady są zwolnieni z egzaminu maturalnego z informatyki. Uprawnienie to przysługuje także wtedy, gdy przedmiot nie był objęty szkolnym planem nauczania danej szkoły.

(3) Uprawnienia określone w p. 1. i 2. przysługują na zasadach określonych w rozporządzeniu MENiS z 7 września 2004 r. w sprawie warunków i sposobu oceniania, klasyfikowania i promowania uczniów i słuchaczy oraz przeprowadzania sprawdzianów i egzaminów w szkołach publicznych. (Dz. U. z 2004 r. Nr 199, poz. 2046, § 18 i § 56) wraz z późniejszymi zmianami zawartymi w Rozporządzeniu MENiS z 14 czerwca 2005 r. (Dz. U. z 2005 r. Nr 108, poz. 905).

(4) Laureaci i finaliści Olimpiady mają ułatwiony lub wolny wstęp do tych szkół wyższych, których senaty podjęły uchwały w tej sprawie, zgodnie z przepisami ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym, na zasadach zawartych w tych uchwałach (Dz. U. z 2005 r. Nr 164 poz. 1365).

(5) Zaświadczenia o uzyskanych uprawnieniach wydaje uczestnikom Komitet Główny.

(6) Komitet Główny ustala skład reprezentacji Polski na XXI Międzynarodową Olimpiadę Informatyczną w 2009 roku na podstawie wyników zawodów III stopnia i regulaminu tej Olimpiady Międzynarodowej.

(7) Komitet Główny może przyznać nagrody nauczycielom lub opiekunom naukowym, którzy przygotowywali laureatów lub finalistów Olimpiady.

(8) Wyznaczeni przez Komitet Główny reprezentanci Polski na olimpiady międzynarodowe oraz finaliści, którzy nie są w ostatniej programowo klasie swojej szkoły, zostaną zaproszeni do nieodpłatnego udziału w X Obozie Naukowo–Treningowym im. Antoniego Kreczmara, który odbędzie się w czasie wakacji 2009 r.

(9) Komitet Główny może przyznawać finalistom i laureatom nagrody, a także stypendia ufundowane przez osoby prawne lub fizyczne.

§7 PRZEPISY KOŃCOWE

(1) Dyrektorzy szkół mają obowiązek dopilnowania, aby wszystkie informacje dotyczące Olimpiady zostały podane do wiadomości uczniów.

(2) Komitet Główny zawiadamia wszystkich uczestników zawodów I i II stopnia o ich wynikach. Uczestnicy zawodów I stopnia, którzy prześlą rozwiązania jedynie przez Internet, zostaną zawiadomieni pocztą elektroniczną, a poprzez witrynę Olimpiady będą mogli zapoznać się ze szczegółowym raportem ze sprawdzania ich rozwiązań.

(3) Każdy uczestnik, który zakwalifikował się do zawodów wyższego stopnia, oraz dyrektor jego szkoły otrzymują informację o miejscu i terminie następnego stopnia zawodów.

(4) Uczniowie zakwalifikowani do udziału w zawodach II i III stopnia są zwolnieni z zajęć szkolnych na czas niezbędny do udziału w zawodach; mają także zagwarantowane na czas tych zawodów bezpłatne zakwaterowanie, wyżywienie i zwrot kosztów przejazdu.

Witryna Olimpiady: `www.oi.edu.pl`

Zawody I stopnia

opracowania zadań

Paweł Parys
Treść zadania, Opracowanie

Piotr Niedźwiedź
Program

Dostępna pamięć: 64 MB.

OI, Etap I, 20.10–17.11.2008

Gaśnice

Bajtazar zbudował nowy pałac. Pałac ten składa się z n pokoi i $n-1$ łączących je korytarzy. Pokoje są ponumerowane od 1 do n. Do pałacu jest jedno wejście, prowadzące do pokoju nr 1. Każdy korytarz łączy dwa różne pokoje. Do każdego pokoju prowadzi od wejścia dokładnie jedna droga (bez zawracania). Inaczej mówiąc, pokoje i łączące je korytarze tworzą **drzewo** *— spójny graf acykliczny.*

Inspektor straży pożarnej, który odbierał pałac, domaga się umieszczenia w pałacu gaśnic. Określił on następujące wymagania:

- *Gaśnice mają być umieszczone w niektórych pokojach, przy czym w jednym pokoju może znaleźć się kilka gaśnic.*
- *Każdemu pokojowi trzeba przydzielić jedną konkretną gaśnicę.*
- *Każda z gaśnic może być przydzielona do gaszenia co najwyżej s różnych pokoi.*
- *Dotarcie z dowolnego pokoju do przypisanej mu gaśnicy może wymagać przejścia co najwyżej k korytarzy.*

Bajtazar, jak to zwykle bywa po zakończeniu budowy, ma bardzo mało pieniędzy. Zastanawia się więc, jaka minimalna liczba gaśnic wystarczy do spełnienia powyższych wymagań?

Wejście

W pierwszym wierszu standardowego wejścia znajdują się trzy liczby całkowite n, s i k pooddzielane pojedynczymi odstępami, $1 \leqslant n \leqslant 100\,000$, $1 \leqslant s \leqslant n$, $1 \leqslant k \leqslant 20$. Każdy z następnych $n-1$ wierszy zawiera po dwie liczby całkowite oddzielone pojedynczym odstępem. W wierszu $i+1$ znajdują się liczby $1 \leqslant x_i < y_i \leqslant n$ reprezentujące korytarz łączący pokoje nr x_i i y_i.

Wyjście

W pierwszym i jedynym wierszu standardowego wyjścia powinna zostać wypisana jedna liczba całkowita — minimalna liczba gaśnic, jakie należy zainstalować w pałacu.

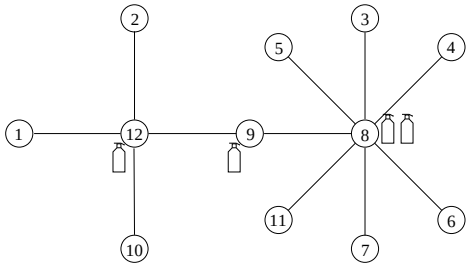
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
4
```



Rozwiązanie

W tym zadaniu, podobnie jak w wielu innych (choć bynajmniej nie we wszystkich!), dobre efekty przynosi postępowanie zachłanne, czyli optymalizowanie ustawienia gaśnic lokalnie, w każdym fragmencie drzewa z osobna. Co to konkretnie oznacza? Drzewo będziemy przetwarzać od liści do korzenia. Intuicyjnie, w każdym poddrzewie[1] chcemy postawić jak najmniej gaśnic, a te, które stawiamy, chcemy postawić jak najwyżej, żeby obejmowały jeszcze jak najwięcej wierzchołków w górę.

Drzewo opisujące pałac Bajtazara nie ma naturalnego korzenia. Jako korzeń możemy wybrać pokój numer 1 (do którego prowadzi wejście do pałacu), ale równie dobrze może to być dowolny inny wierzchołek. Ważne, aby jakiś korzeń wybrać i ustalić go raz na zawsze. Jeśli algorytm oblicza minimalną liczbę potrzebnych gaśnic, to z definicji jego wynik nie zależy od wyboru korzenia. Jednak samo rozmieszczenie gaśnic już będzie zależało od tego, który wierzchołek jest korzeniem.

Przejdźmy teraz do konkretów. Zanim opiszemy algorytm, zastanówmy się, jakie informacje na temat rozmieszczenia gaśnic w poddrzewie są potrzebne „na zewnątrz", w pozostałej części drzewa. Na pewno ważne jest, ile gaśnic zostało rozstawionych. Nie ma znaczenia, gdzie dokładnie one stoją i do których pokoi są przypisane. Potrzebujemy jednak wiedzieć, czy blisko korzenia poddrzewa znajdują się jakieś nie do końca wykorzystane gaśnice, na jakiej są one głębokości i do ilu jeszcze pokoi możemy je przypisać. Dobrze jest wyobrażać sobie, że gaśnica ma s „końcówek", z których prowadzimy do pokoi ścieżki długości co najwyżej k. Niezbędna jest też informacja, ile jeszcze pokoi w poddrzewie nie ma przypisanej gaśnicy i na jakiej są głębokości.

Rozmieszczenie gaśnic w danym poddrzewie będziemy zatem charakteryzować przez następujący układ liczb: $(g, x_0, x_1, \ldots, x_k, y_0, y_1, \ldots, y_k)$, przy czym:

[1] Pod pojęciem *poddrzewa* będziemy w tym opisie rozumieli dowolny wierzchołek drzewa wraz ze wszystkimi jego potomkami.

- g to liczba gaśnic rozmieszczonych w tym poddrzewie
- x_i to liczba wolnych końcówek gaśnic, które sięgają na i pokoi w górę ponad korzeń
- y_i to liczba pokoi na głębokości i (korzeń znajduje się na głębokości 0), które potrzebują przypisania gaśnicy.

Dla przykładu, jeśli ustawimy gaśnicę o 1 pod korzeniem (tzn. w dziecku korzenia) i nie przypiszemy do niej żadnego pokoju, to x_{k-1} będzie równe s, gdyż gaśnica ta sięga jeszcze o $k-1$ pokoi ponad korzeń; jeśli już przypisane jest do niej jakieś j pokoi, to x_{k-1} będzie równe $s-j$. Zauważmy w szczególności, że (jeśli rozstawienie jest poprawne) $y_k = 0$, ponieważ pokojom na głębokości k nie możemy przypisać gaśnicy spoza poddrzewa (zatem y_k jest w naszym ciągu tylko dla wygody).

Gdy przeprowadzamy obliczenia w jakimś wierzchołku, to rozstawienia dla wszystkich poddrzew odpowiadających dzieciom tego wierzchołka powinny być już wyznaczone (drzewo przetwarzamy od liści do korzenia). Algorytm musi działać tak, żeby w tym wierzchołku umieścić jak najmniej gaśnic, żeby pokoje, do których gaśnica nie jest przypisana, były położone jak najbliżej korzenia oraz żeby wolne końcówki gaśnic sięgały jak najwyżej. Po chwili zastanowienia dochodzimy do wniosku, że algorytm musi (z dokładnością do drobnych szczegółów) działać w następujący sposób:

1. Najpierw sumujemy charakteryzacje wszystkich poddrzew; robimy to po współrzędnych, tzn. nowe g to suma wszystkich g z poddrzew, nowe x_0 to suma wszystkich x_0 z poddrzew itd.

2. Przesuwamy ciągi (ponieważ korzeń poddrzewa jest teraz o 1 wyżej niż poprzednio):

   - za nowe $x_0, x_1, \ldots, x_{k-1}$ przyjmujemy $x_1, x_2, \ldots, x_k$
   - za $y_1, y_2, \ldots, y_k$ przyjmujemy $y_0, y_1, \ldots, y_{k-1}$
   - za x_k przyjmujemy 0 (w korzeniu nie stoi na razie żadna gaśnica, więc nie ma wolnych końcówek wystających o k w górę)
   - za y_0 przyjmujemy 1 (jest jeden pokój na głębokości 0, który potrzebuje przypisania końcówki gaśnicy, a mianowicie korzeń).

   Zapominamy o starym x_0 (te wolne końcówki i tak nie sięgają w górę, nawet do nowego korzenia) oraz o starym y_k (które i tak jest równe zeru).

3. W korzeniu aktualnie badanego poddrzewa stawiamy $c = \lceil y_k/s \rceil$ gaśnic (czyli tyle, ile jest bezwzględnie koniecznych), tzn. wykonujemy instrukcje:

$$g \mathrel{+}= c \quad \text{oraz} \quad x_k = s \cdot c.$$

4. Dla każdego $i = 0, 1, \ldots, k$ parujemy nieprzypisane pokoje na głębokości i z wolnymi końcówkami wystającymi o i w górę. Możemy to zrobić, gdyż jeśli końcówka wystaje o i ponad korzeń, to sięga też na głębokość i. Innymi słowy, dla $c = \min(x_i, y_i)$ wykonujemy:

$$x_i \mathrel{-}= c \quad \text{oraz} \quad y_i \mathrel{-}= c.$$

5. Dla każdego $i < k$ parujemy nieprzypisane pokoje na głębokości i z wolnymi końcówkami wystającymi o $i+1$ w górę. Zatem dla $c = \min(x_{i+1}, y_i)$ wykonujemy instrukcje:
$$x_{i+1} \mathrel{-}= c \quad \text{oraz} \quad y_i \mathrel{-}= c.$$
Ważne, że wykonujemy ten krok dopiero po kroku 4.

W ten sposób otrzymujemy wynikowy ciąg dla naszego poddrzewa, który przekazujemy w górę. Zauważmy, że będzie w nim zachodziło $y_k = 0$, tak jak chcieliśmy, gdyż po wykonaniu kroku 3 zachodzi $x_k \geqslant y_k$.

W korzeniu całego drzewa zamiast kroków 3-5 trzeba oczywiście postąpić nieco inaczej:

3'. Teraz przeglądamy kolejne $i \in \{0, \ldots, k\}$ (będą to głębokości pokoi, które nie mają przypisanej gaśnicy) i dla każdego i przeglądamy po kolei wszystkie $j \in \{i, \ldots, k\}$ (długości wystających końcówek gaśnic). Ważna jest tutaj kolejność — zaczynamy od najmniejszych j, czyli od końcówek wystających jak najmniej w górę; te wystające bardziej mogą być także przydatne dla kolejnych i. Dla każdej takiej pary łączymy jak najwięcej pokoi z gaśnicami, czyli wykonujemy instrukcje
$$y_i \mathrel{-}= c \quad \text{oraz} \quad x_j \mathrel{-}= c$$
dla $c = \min(y_i, x_j)$.

4'. Pozostałych końcówek nie da się wykorzystać, więc w korzeniu drzewa stawiamy nowe gaśnice. Potrzeba ich $c = \lceil (y_0 + \ldots + y_k)/s \rceil$; gaśnice te przypisujemy wszystkim nieobsłużonym pokojom, tzn.
$$g \mathrel{+}= c, \quad x_k \mathrel{+}= s \cdot c - (y_0 + \ldots + y_k), \quad y_0 = \ldots = y_k = 0.$$

Implementując wprost powyższe rozwiązanie, dostajemy złożoność $O(nk)$. Autor zadania oraz jurorzy Olimpiady nie znają żadnego rozwiązania mającego lepszą złożoność (aczkolwiek nie jest wykluczone, że takie istnieje).

Intuicyjnie jasne jest, że opisane powyżej postępowanie daje najmniejszą możliwą liczbę gaśnic. Spróbujmy to jednak udowodnić formalnie. Dowód jest indukcyjny — dla coraz to większych poddrzew p będziemy dowodzić równocześnie, że:

A) Każde poprawne rozmieszczenie gaśnic w drzewie zawiera w poddrzewie p przynajmniej tyle gaśnic, ile zwraca nasz algorytm.

B) Mając poprawne rozmieszczenie gaśnic w całym drzewie, możemy przeorganizować gaśnice w poddrzewie p tak, aby stały zgodnie z naszym algorytmem. W reszcie drzewa ustawienie powinno pozostać bez zmian. Jedynie jeśli w p jest więcej gaśnic niż obliczył to nasz algorytm, to zbyteczne gaśnice wynosimy o jeden wierzchołek ponad korzeń p. Zgodność ma dotyczyć także przypisań pokoi do końcówek.

   Żeby być precyzyjnym, należy wyobrażać sobie przypisania pokoi końcówkom jako ścieżki długości co najwyżej k, jednak niekoniecznie ścieżki proste. Ścieżka taka biegnie najpierw w górę, a potem w dół. Nasz algorytm pewne przypisania wykonuje w obrębie poddrzewa p. Ścieżki od pozostałych, nieprzypisanych pokoi i nieprzypisanych końcówek prowadzi w górę, ponad p. Gdzieś tam poza p mogą

one zostać ze sobą połączone, ale takie przypisanie traktujemy już jako przypisanie poza p. Tutaj chcemy, żeby przypisania wykonane przez nasz algorytm wewnątrz p były w rozważanym rozmieszczeniu wykonane w ten sam sposób. Natomiast jeśli jakieś pokoje lub końcówki wychodzą w naszym algorytmie poza p, to w otrzymanym rozmieszczeniu też mają wychodzić poza p.

Baza indukcji (liść drzewa) jest trywialna. Natomiast aby dowieść powyższe dla jakiegoś poddrzewa p, najpierw stosujemy założenie indukcyjne do poddrzew tego poddrzewa odpowiadających dzieciom korzenia i przestawiamy w nich gaśnice tak, aby stały zgodnie z naszym algorytmem.

Zauważmy, że w korzeniu p trzeba postawić co najmniej $\lceil y_k/s \rceil$ gaśnic (tyle ile stawia nasz algorytm), bo pokojom na głębokości k trzeba przypisać gaśnicę, która stoi dokładnie tutaj. To już dowodzi własności A.

Proponuję Ci, Drogi Czytelniku, w tym miejscu przerwać czytanie i samodzielnie udowodnić własność B. Dowód nie jest trudny, wymaga jednak analizy różnych możliwych przypadków i przekonania się, że w każdym z nich wszystko jest w porządku. Jeśli jednak wolałbyś dowód ten przeczytać, zamieszczamy go poniżej.

Zauważmy wpierw, że rozstawienie gaśnic w poddrzewie p jest prawie wszędzie takie, jak powinno być; jedynie w korzeniu p może być więcej gaśnic niż ustawiłby tam nasz algorytm. Podobnie z przypisaniami pokoi do gaśnic — interesują nas jedynie te, które przechodzą przez korzeń p, gdyż pozostałe są już poprawione z założenia indukcyjnego.

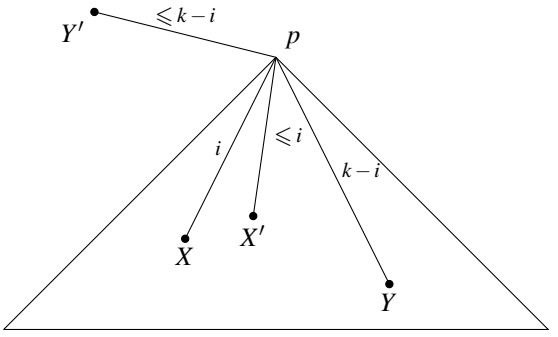
Poczyńmy na początku następujące spostrzeżenie: jeśli w rozważanym rozmieszczeniu istnieje połączenie pewnego pokoju na głębokości i z pewną końcówką wystającą na wysokość j, a nasz algorytm też wymaga połączenia pewnego pokoju na głębokości i z pewną końcówką o długości j, ale być może innych, to możemy połączenia tych pokoi i końcówek pozamieniać tak, aby prowadziły zgodnie z naszymi oczekiwaniami. To samo dotyczy pokoi lub końcówek na ustalonym poziomie niepołączonych z niczym wewnątrz p. Innymi słowy, nie liczy się, co dokładnie z czym jest połączone, liczy się tylko liczba połączeń danego rodzaju (rodzaj jest charakteryzowany przez głębokość łączonego pokoju i długość łączonej końcówki lub też brak jednej z tych dwóch rzeczy).

To pozwala już poradzić sobie z nadmiarowymi gaśnicami w korzeniu p. Przypuśćmy, że w korzeniu p jest c gaśnic. Początkowo pokoje na głębokości k mogą być podpięte do dowolnych z nich. Możemy jednak pozamieniać połączenia tak, żeby tylko $\lceil y_k/s \rceil$ gaśnic było połączonych z pokojami na głębokości k, pozostałe tylko z pokojami na mniejszej głębokości oraz z pokojami spoza p. Wówczas te $c - \lceil y_k/s \rceil$ nadmiarowych gaśnic możemy przesunąć o 1 w górę, uzyskując w korzeniu p tyle gaśnic, ile ustawiłby tam nasz algorytm.

Pozostaje naprawić przypisania. Bierzemy kolejno pokoje i staramy się je przypisać zgodnie z tym, jak zostałoby to wykonane w naszym algorytmie. Jeżeli natrafiamy na jakąś niezgodność, to musi wystąpić jeden z kilku przypadków:

- Mamy pokój, który zgodnie z algorytmem nie ma być przypisany niczemu w p, a teraz jest czemuś przypisany. W takim przypadku długość ścieżki od gaśnicy do tego pokoju jest mniejsza niż $k-1$, inaczej nasz algorytm wykonałby takie połączenie (jak już wspominaliśmy: niekoniecznie akurat tego wierzchołka z tą końcówką, ale między wierzchołkiem i gaśnicą na takich głębokościach). Zatem możemy nasze przypisanie wydłużyć o 2 tak, żeby wystawało poza p. To jest to, czego trzeba — nie mamy już przypisania wewnątrz p.

- Mamy pokój X na głębokości i, który zgodnie z algorytmem ma być przypisany końcówce Y wystającej o i w górę (czyli gaśnicy na głębokości $k-i$), a teraz jest przypisany pewnej innej końcówce Y' (przy czym X i Y znajdują się wewnątrz p, a Y' może znajdować się gdziekolwiek w drzewie). Jeśli końcówka Y nie jest niczemu przypisana, to nie ma problemu, po prostu przepinamy. Przypuśćmy, że końcówka Y jest połączona z pewnym pokojem X' (rys. 1). Istotne jest, że połączenia między X' i Y oraz między X i Y' przechodzą przez korzeń p, gdyż pokoje X i Y znajdują się w obrębie p, a w obrębie poddrzew p występują jedynie połączenia zgodne z wymaganiami naszego algorytmu, podczas gdy te takie nie są. Zamieniamy te dwa połączenia: łączymy X z Y oraz X' z Y'. Aby móc to zrobić, musimy mieć pewność, że nowe ścieżki są nie dłuższe niż k. Ścieżka z X do Y ma długość k. Odległość od korzenia p do Y' to co najwyżej $k-i$ (bo istniało połączenie z X do Y'), a odległość od korzenia p do X' to co najwyżej i (bo istniało połączenie z X' do Y) — a zatem rzeczywiście ścieżka z X' do Y' ma długość co najwyżej k.



Rys. 1: Zamiast łączyć X z Y' i X' z Y, można połączyć X z Y i X' z Y'.

- Mamy pokój X na głębokości i, który zgodnie z algorytmem ma być przypisany końcówce Y wystającej o $i+1$ w górę (czyli gaśnicy na głębokości $k-i-1$), a teraz jest przypisany pewnej innej końcówce Y'. Jeśli końcówka Y nie jest niczemu przypisana, to możemy po prostu przypisać X do Y. Przypuśćmy, że końcówka Y jest połączona z pewnym pokojem X'. Połączenia między X' i Y oraz między X i Y' przechodzą przez korzeń p, jak poprzednio. Wiemy, że odległość od korzenia p do Y' to co najwyżej $k-i$ (bo istniało połączenie z X do Y'), a odległość od korzenia p do X' to co najwyżej $i+1$ (bo istniało połączenie z X' do Y). Jeśli przynajmniej jedna z tych nierówności jest ostra, to możemy połączyć X z Y oraz X' z Y' i obie ścieżki będą długości co najwyżej k. Co jednak, jeśli w obu miejscach mamy równości? Zauważmy, że wtedy oba pokoje X' i Y' znajdują się poza p. Gdyby bowiem na przykład pokój X' był wewnątrz p, to nasz algorytm mógłby połączyć X' z Y ścieżką długości k, co ma wyższy priorytet (krok 4) niż łączenie X z Y ścieżką długości $k-1$ (krok 5). Podobnie, gdyby końcówka Y' była wewnątrz p, to nasz algorytm raczej połączyłby X z Y' ścieżką długości k niż X z Y ścieżką długości $k-1$. Skoro więc oba X' i Y' są poza p, to najkrótsza ścieżka między nimi nie przechodzi przez korzeń p, lecz jest co najmniej o 2 krótsza niż

  (odległość od korzenia p do X') + (odległość od korzenia p do Y'),

  czyli jest długości co najwyżej $k-1$. Zatem możemy dokonać zamiany połączeń.

W końcu dochodzimy do korzenia. Zauważmy, że nasz algorytm w korzeniu maksymalizuje liczbę połączeń istniejących końcówek z pokojami nieprzypisanymi jeszcze do żadnej gaśnicy — szerszą argumentację uzasadniającą ten fakt pozostawiamy już Czytelnikowi. Wynika z tego, że w każdym innym rozwiązaniu potrzeba w korzeniu co najwyżej tyle gaśnic, ile w naszym algorytmie. Ponadto, każdy inny sposób połączenia można sprowadzić w oczywisty sposób do uzyskanego przez nasz algorytm (po prostu zapominamy o starych połączeniach i łączymy tak, jak w naszym algorytmie; tym razem nie musimy się martwić o to, co dzieje się poza aktualnym poddrzewem, bo rozważamy już całe drzewo). Tym oto sposobem dobrnęliśmy do końca dowodu poprawności naszego rozwiązania.

Opisane powyżej rozwiązanie zostało zaimplementowane w plikach `gas.cpp`, `gas1.pas` oraz `gas2.java`.

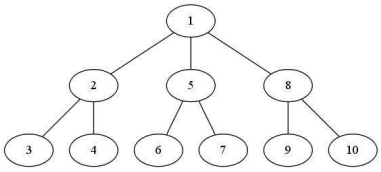
Testy

Rozwiązania zadania były sprawdzane na 10 zestawach danych testowych, w każdym zestawie zgrupowane były dwa lub trzy testy. Począwszy od piątej grupy, testy z końcówką *a* to testy wydajnościowe. Testy 10*b*, 10*c* sprawdzają użycie typu całkowitego 64-bitowego w rozwiązaniu wzorcowym. Wszystkie testy zostały wygenerowane losowo, jednak dla różnych parametrów charakteryzujących wygląd drzewa. W poniższej tabelce przedstawiamy statystyki poszczególnych testów. Kolumny n, s, k określają parametry z treści zadania, d_{max} to maksymalny stopień (liczba sąsiadów) wierzchołka w drzewie, *diam* to średnica drzewa (maksymalna odległość między dwoma wierzchołkami), natomiast g to potrzebna liczba gaśnic (wynik).

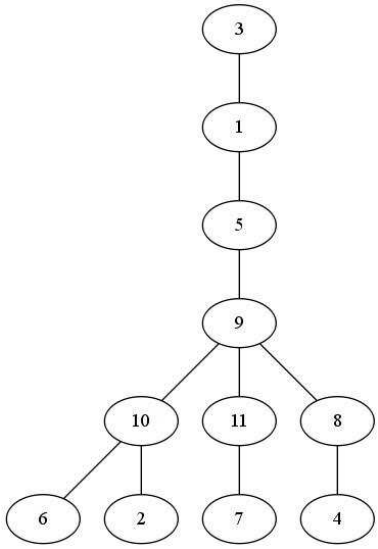
Nazwa	**n**	**s**	**k**	d_{max}	*diam*	g	**Opis**
gas1a.in	10	10	3	3	4	1	
gas1b.in	11	2	2	5	5	6	
gas2a.in	20	20	1	18	3	2	duże d_{max}
gas2b.in	25	3	5	4	9	9	
gas3a.in	159	10	3	7	8	16	
gas3b.in	400	200	20	11	15	2	duże s
gas4a.in	1 000	7	4	8	18	143	
gas4b.in	2 000	9	18	6	122	223	dość duży *diam*
gas5a.in	4 728	3	20	11	12	1 576	
gas5b.in	3 702	10	16	3	186	371	drzewo binarne
gas6a.in	8 004	7	19	21	13	1 144	
gas6b.in	20 000	126	7	23	147	1 206	
gas7a.in	44 192	17	20	101	14	2 600	
gas7b.in	50 000	10 000	2	9	435	10 196	$k = 2$, więc duże g
gas8a.in	47 231	31	20	21	10	1 524	

Nazwa	n	s	k	d_{max}	*diam*	*g*	Opis
gas8b.in	70000	40	13	4	434	2286	
gas9a.in	90000	131	18	31	21	688	
gas9b.in	100000	6000	20	6	139	541	
gas10a.in	68929	4	20	3	32	17233	małe *s*
gas10b.in	90001	90001	2	30000	6	30000	duże d_{max} i *g*
gas10c.in	99001	99001	2	33000	6	33000	duże d_{max} i *g*

Poniżej zamieszczamy ilustracje dwóch pierwszych testów.



Test *gas1a.in*.



Test *gas1b.in*.

Paweł Parys
Treść zadania, Opracowanie

Szymon Wrzyszcz
Program

Dostępna pamięć: 128 MB.

OI, Etap I, 20.10–17.11.2008

Kamyki

Jaś i Małgosia grają w „kamyki". Początkowo na stole ułożona jest pewna liczba kamyków, pogrupowanych w n kupek. Kupki znajdują się obok siebie w jednym rzędzie. Ustawienie kamieni spełnia dodatkowo własność, że na każdej kupce jest nie mniej kamieni niż na kupce sąsiadującej z nią po lewej (nie dotyczy to pierwszej kupki, na lewo od której nie ma już nic). Gracze na przemian usuwają z jednej, wybranej przez siebie w danym ruchu kupki dowolną dodatnią liczbę kamieni. Muszą to jednak robić tak, aby na tej kupce nie zostało mniej kamieni niż na kupce o jeden w lewo. Początkowa własność ustawienia zostaje więc zachowana. Gdy ktoś nie ma już możliwości wykonania ruchu (tzn. przed jego ruchem wszystkie kamienie są zdjęte ze stołu), to przegrywa. Jaś zawsze zaczyna.

Małgosia jest bardzo dobra w tę grę i jeśli tylko może, to gra tak, aby wygrać. Jaś prosi Cię o pomoc — chciałby wiedzieć, czy przy danym ustawieniu początkowym ma w ogóle szanse wygrać z Małgosią. Napisz program, który to określi.

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita u ($1 \leqslant u \leqslant 10$), oznaczająca liczbę ustawień początkowych gry do przeanalizowania. W kolejnych $2u$ wierszach znajdują się opisy tych ustawień; każdy z nich zajmuje dokładnie dwa wiersze.

W pierwszym wierszu każdego opisu znajduje się jedna liczba całkowita n, $1 \leqslant n \leqslant 1\,000$ — liczba kupek kamieni. W drugim wierszu opisu znajduje się n nieujemnych liczb całkowitych a_i, pooddzielanych pojedynczymi odstępami i reprezentujących liczby kamieni na poszczególnych kupkach, od lewej do prawej. Liczby te spełniają nierówności $a_1 \leqslant a_2 \leqslant \ldots \leqslant a_n$. Łączna liczba kamieni w żadnym opisie nie przekracza 10 000.

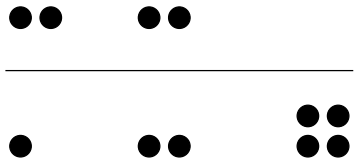
Wyjście

Na standardowe wyjście powinno zostać wypisanych u wierszy. W i-tym wierszu wyjścia (dla $1 \leqslant i \leqslant u$) powinno znajdować się słowo `TAK`*, jeśli Jaś może wygrać, zaczynając z i-tego na wejściu ustawienia początkowego, lub słowo* `NIE`*, jeśli Jaś zawsze przegra przy optymalnej grze Małgosi.*

Przykład

Dla danych wejściowych:

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



poprawnym wynikiem jest:
```
NIE
TAK
```

Rozwiązanie

Analiza problemu

W tym zadaniu, podobnie jak w wielu innych zadaniach o grach, należy określić, czy pierwszy gracz ma strategię wygrywającą, jeśli gra rozpoczyna się od danego ustawienia kamyków na kupkach. Takie ustawienia będziemy nazywali *pozycjami*; pozycja charakteryzowana jest przez liczby kamieni na poszczególnych kupkach.

Poczyńmy wpierw oczywiste spostrzeżenie: gracze są nierozróżnialni. Obojętnie czy z danego stanu rusza się Jaś, czy Małgosia, mają takie same możliwe ruchy do wykonania. Również warunek wygranej jest symetryczny: jeśli gracz nie ma do wyboru żadnego ruchu, to przegrywa, niezależnie czy jest to Jaś czy Małgosia. Ma więc sens mówienie o pozycjach wygrywających i przegrywających (dla aktualnego gracza). Pozycję nazywamy *wygrywającą*, gdy gracz rozpoczynający w niej grę ma strategię wygrywającą. W przeciwnym przypadku pozycję nazwiemy *przegrywającą*. Dla ścisłości dopowiedzmy jeszcze, że po każdym ruchu łączna liczba kamieni zmniejsza się co najmniej o jeden, więc gra kończy się zawsze po skończonej liczbie ruchów. Zatem pojęcie pozycji przegrywającej ma sens — nie tylko pierwszy gracz nie ma możliwości wygrania, zaczynając od niej, ale też drugi gracz może na pewno wygrać, czyli doprowadzić do przegranej pierwszego gracza. Zauważmy, że:

1. Pozycja końcowa (wszystkie kupki puste) jest przegrywająca.
2. Jeżeli z danej pozycji istnieje ruch prowadzący do pozycji przegrywającej, to pozycja ta jest wygrywająca (jeśli tak się ruszymy, to przeciwnik przegra).
3. Jeżeli z danej pozycji wszystkie ruchy prowadzą do pozycji wygrywających, to pozycja ta jest przegrywająca (cokolwiek zrobimy, przeciwnik wygra).

Warunek 1 wynika z warunku 3, ale został dodany dla zwiększenia czytelności.

Rozwiązanie wykładnicze

Powyższe rozważania prowadzą bezpośrednio do rozwiązania, w którym sprawdzamy, czy dana pozycja początkowa jest wygrywająca, korzystając wprost z powyższych warunków. Sprowadza się ono do prostej funkcji rekurencyjnej parametryzowanej pozycją. Poprawność rozwiązania jest oczywista. Po chwili zastanowienia widzimy, że czas działania jest wykładniczy względem rozmiaru pozycji, czyli łącznej liczby kamyków. Rozwiązanie takie, zaimplementowane w plikach `kams1.cpp`, `kams3.pas` i `kams4.java`, przechodziło tylko jeden test.

Powyższe rozwiązanie można trochę poprawić poprzez spamiętywanie wyników dla już rozpatrzonych pozycji. Jest to znacząca optymalizacja, gdyż wiele sekwencji ruchów może

prowadzić do tego samego układu. Jednak liczba pozycji osiągalnych z pozycji początkowej nadal jest wykładnicza, więc to rozwiązanie również ma szansę przejść jedynie małe testy (konkretnie przechodziło ono trzy pierwsze testy). Implementacje takiego rozwiązania można znaleźć w plikach `kams2.cpp` i `kams5.java`.

Rozwiązanie wzorcowe

Aby rozwiązać nasze zadanie w sensownej złożoności czasowej, potrzebujemy łatwiej obliczalnego kryterium pozwalającego określać, czy pozycja jest wygrywająca, czy przegrywająca. W przypadku zadań podobnych do tego często udaje się przypisać każdej pozycji p pewną (łatwą do obliczenia) liczbę $value(p)$ (nazywaną *wartością*) taką, że:

1. Wartość pozycji końcowej wynosi 0.
2. Jeśli wartość jakiejś pozycji jest niezerowa, to istnieje z niej ruch prowadzący do pozycji o wartości 0.
3. Jeśli wartość jakiejś pozycji p jest równa zeru, to każda pozycja, do której prowadzi ruch z p, ma wartość niezerową.

Przypuśćmy, że dla naszej gry udałoby się zdefiniować taką funkcję *value*. Porównajmy powyższe punkty z własnościami 1-3 pozycji wygrywających i przegrywających. Widzimy, że pozycje przegrywające to dokładnie te, których wartość wynosi 0. Formalnie możemy dowieść tego indukcyjnie po maksymalnej liczbie ruchów do końca gry z danej pozycji. Dostajemy więc bardzo proste rozwiązanie zadania: wystarczy obliczyć wartość pozycji początkowej; jeśli wychodzi 0, odpowiadamy „`NIE`”, w przeciwnym przypadku „`TAK`”.

Pozostaje zdefiniować funkcję *value*. Niech a_i oznacza liczbę kamieni na i-tej kupce. Dla ułatwienia zapisu przyjmijmy, że liczba kupek n jest parzysta (jeśli jest nieparzysta, to na początek możemy wstawić dodatkową kupkę zawierającą 0 kamieni i nic to nie zmieni w grze). Do definicji wartości często (w tym przypadku także) używa się operacji xor (oznaczenie: $\oplus$) na liczbach całkowitych, czyli sumy modulo 2 na bitach (cyfrach w zapisie dwójkowym) tych liczb. Innymi słowy, liczba $a \oplus b$ ma jedynki w zapisie dwójkowym na tych pozycjach, na których dokładnie jedna spośród liczb a, b ma jedynkę. Przykładowo:

$$12 \oplus 10 = [01100]_2 \oplus [01010]_2 = [00110]_2 = 6.$$

Zauważmy, że operacja xor jest przemienna ($a \oplus b = b \oplus a$), łączna ($(a \oplus b) \oplus c = a \oplus (b \oplus c)$) oraz odwrotna do samej siebie ($a \oplus b \oplus b = a$). Funkcję *value* określimy jako

$$(a_2 - a_1) \oplus (a_4 - a_3) \oplus \ldots \oplus (a_n - a_{n-1}). \tag{1}$$

Bierzemy więc xor-a wszystkich różnic liczb kamieni kolejnych par kupek o numerach parzystym i nieparzystym. Zauważmy, że w wyniku otrzymujemy liczbę całkowitą nieujemną.

Dlaczego taka funkcja spełnia wymagane własności? Własność 1 jest oczywista. Uzasadnienie własności 3 także nie jest trudne. Przypuśćmy, że wartość jakiejś pozycji wynosi 0. Po wykonaniu jednego ruchu zmniejszy się rozmiar dokładnie jednej kupki, zatem zmieni się dokładnie jedna różnica będąca argumentem naszego xor-a. Zauważmy, że jeśli zmieniamy dokładnie jeden argument xor-a z s na s', to wartość całego xor-a także na pewno się zmieni (konkretnie: z 0 na $s \oplus s' \neq 0$).

Aby udowodnić własność 2, przypuśćmy, że wartość w pewnej pozycji jest niezerowa i wynosi x. Niech i będzie numerem najbardziej znaczącego bitu x, który jest równy 1. Wówczas przynajmniej jeden spośród argumentów naszego xor-a ma i-ty bit równy 1. Oznaczmy pewien taki argument przez s (jest to $a_j - a_{j-1}$ dla pewnego parzystego j). Wówczas xor pozostałych argumentów jest równy $x \oplus s$. W liczbie tej i-ty bit jest równy 0, a wszystkie bardziej znaczące bity są równe tym w liczbie s, gdyż w x bity te są wszystkie wyzerowane. Stąd $x \oplus s < s$. Jeśli w najbliższym ruchu zmienimy s na $s' = x \oplus s$, to nowa wartość wyniesie:

$$x \oplus s \oplus s' = x \oplus s \oplus x \oplus s = 0.$$

Zauważmy, że taki ruch rzeczywiście zawsze istnieje. Jest to bowiem ruch, w którym zmniejszamy różnicę $s = a_j - a_{j-1}$. Różnicę tę można zmniejszyć o dowolną niezerową (i nie większą niż s) liczbę poprzez zdjęcie właśnie tylu kamieni ze stosu a_j. To kończy uzasadnienie faktu, że nasza definicja (1) wartości pozycji spełnia własności 1-3.

Pozostaje wciąż pytanie, w jaki sposób można dojść do powyższej definicji funkcji *value*? Trudno opisać to ściśle, ale spróbujemy przedstawić pewne intuicje. Zauważmy najpierw, że dowód powyższych własności opierał się praktycznie tylko na własnościach operacji xor, a w bardzo niewielkim stopniu na tym, co konkretnie xor-ujemy. Aby punkty 1 i 3 były spełnione, wystarczyłoby xor-ować cokolwiek, byleby każdy a_i występował w dokładnie jednym argumencie xor-a (na przykład można by wziąć xor wszystkich a_i) — wówczas zmiana a_i zmienia wartość. Aby zapewnić własność 2, wystarczy (jak już widzieliśmy), aby jakiś legalny ruch zmniejszał dowolny (tj. każdy możliwy) argument xor-a o dowolną wartość. Gdybyśmy po prostu xor-owali wszystkie a_i, to tak by nie było: a_i możemy zmniejszać tylko do a_{i-1}, a nie do zera. Tym co można zmniejszać do dowolnej nieujemnej wartości, są właśnie różnice. Wreszcie wspomniana własność występowania a_i w dokładnie jednym argumencie wymusza, aby brać tylko co drugą różnicę.

Rozwiązanie wzorcowe zostało zaimplementowane w plikach `kam.cpp`, `kam1.pas`, `kam2.java`. Jego złożoność czasowa jest liniowa względem n, więc może ono działać w rozsądnym czasie nawet dla liczby kupek rzędu 1 000 000; liczba kamieni na każdej kupce mogłaby bez problemu być ograniczona przez 10^9. W zadaniu obrano niższe limity na dane wyłącznie celem zmylenia zawodników. Autor zadania oraz jurorzy Olimpiady nie przypuszczają, aby istniało jakieś rozwiązanie (np. zachłanne lub dynamiczne), które działałoby tylko dla tak zmniejszonych limitów.

Gra „Staircase Nim”

Zadanie *Kamyki* ma rozwiązanie nieco podobne jak zadanie *Gra* z XI Olimpiady Informatycznej. W opisie rozwiązania tamtego zadania w [11] została zdefiniowana gra „Staircase Nim”. Występują w niej schody złożone z pewnej liczby stopni, które są numerowane kolejnymi liczbami naturalnymi, przy czym najniższy stopień ma numer 1. Na każdym stopniu znajduje się pewna liczba kamieni. W grze bierze udział dwóch graczy, którzy wykonują ruchy naprzemiennie. Ruch polega na przełożeniu dowolnej (dodatniej) liczby kamieni z dowolnie wybranego stopnia na stopień o numerze o jeden mniejszym (jeśli wybrano stopień o numerze 1, to kamienie są usuwane z dalszej gry). Gracz, który nie może wykonać ruchu (na żadnym stopniu nie ma już kamieni), przegrywa. Dana pozycja gry „Staircase Nim” jest przegrywająca wtedy i tylko wtedy, gdy xor liczby kamieni na

stopniach o nieparzystych numerach jest równy zeru (dowód tego faktu można znaleźć we wspomnianym już opracowaniu zadania *Gra* w książeczce XI OI).

Oznaczmy $r_i = a_i - a_{i-1}$ dla $1 \leqslant i \leqslant n$, przyjmując $a_0 = 0$. Mając dane wszystkie r_i, możemy łatwo odzyskać wyjściowy ciąg a_i, gdyż $a_i = \sum_{j=1}^{i} r_j$ dla każdego $i = 1, 2, \ldots, n$. Wynika stąd, że obie reprezentacje stanu gry są równoważne. Warunek $a_{i-1} \leqslant a_i$ przekłada się na $r_i \geqslant 0$. Wykonanie ruchu polega na wybraniu numeru kupki i oraz zdjęciu z niej pewnej liczby k kamieni. W wyniku takiej operacji a_i zmniejsza się o k. Przy reprezentacji układu ciągiem r, taki ruch odpowiada zmniejszeniu r_i o k i zwiększeniu r_{i+1} o k (o ile $i \leqslant n-1$), co można sobie wyobrazić jako przerzucenie k kamieni z i-tego na $(i+1)$-szy stopień. Zauważmy, że otrzymana gra jest równoważna grze „Staircase Nim" z „odwróconymi schodami" (tzn. najniższy stopień ma numer n, a najwyższy — numer 1). Tym samym rozwiązanie sprowadza się do wykonania xor-owania r_n, r_{n-2}, itd. Pozycja jest przegrywająca wtedy i tylko wtedy, gdy w ten sposób otrzymamy 0.

Testy

Zadanie było sprawdzane za pomocą 14 testów, z których każdy składa się z dziesięciu układów do rozwiązania. Wszystkie testy zostały wygenerowane (do jakiegoś stopnia) losowo. W każdym zestawie występuje co najmniej jeden test z odpowiedzią `TAK` i co najmniej jeden z odpowiedzią `NIE`. Liczby układów wygrywających i przegrywających w ramach jednego testu są podobne, mimo że losowy układ ma dużo większe prawdopodobieństwo bycia wygrywającym (fakt stwierdzony eksperymentalnie). Poniżej podajemy rozmiary poszczególnych testów, kolumna n_{max} to maksymalna (po wszystkich układach w teście) liczba kupek, a s_{max} to maksymalna łączna liczba kamieni.

Nazwa	n_{max}	s_{max}	Opis
kam1.in	5	10	bardzo małe dane
kam2.in	7	25	małe dane
kam3.in	12	47	małe dane
kam4.in	15	74	
kam5.in	17	121	
kam6.in	80	10000	dużo kamieni
kam7.in	153	155	
kam8.in	302	4582	
kam9.in	450	8812	
kam10.in	545	10000	
kam11.in	767	4364	
kam12.in	930	10000	
kam13.in	1000	7690	
kam14.in	1000	10000	maksymalne dane

Jakub Radoszewski, Wojciech Rytter — Treść zadania, Opracowanie
Adam Gawarkiewicz — Program

Dostępna pamięć: 32 MB. *OI, Etap I, 20.10–17.11.2008*

Przyspieszenie algorytmu

Bajtazar musi za karę obliczyć pewną paskudną i tajemniczą funkcję logiczną $\mathbf{F}(x,y)$*, która dla dwóch ciągów liczb naturalnych* $x = (x_1,\ldots,x_n)$*,* $y = (y_1,\ldots,y_m)$ *jest zdefiniowana w następujący sposób:*

boolean $\mathbf{F}(x,y)$
  if $\mathbf{W}(x) \neq \mathbf{W}(y)$ **then return** 0
  else if $|\mathbf{W}(x)| = |\mathbf{W}(y)| = 1$ **then return** 1
  else return $\mathbf{F}(\mathbf{p}(x),\mathbf{p}(y)) \wedge \mathbf{F}(\mathbf{s}(x),\mathbf{s}(y))$.

W powyższym zapisie:

- $\mathbf{W}(x)$ *oznacza zbiór złożony ze wszystkich liczb ciągu* x *(ignorujemy powtórzenia i kolejność liczb),*
- $\mathbf{p}(x)$ *jest najdłuższym prefiksem (początkowym fragmentem) ciągu* x*, dla którego* $\mathbf{W}(x) \neq \mathbf{W}(\mathbf{p}(x))$*,*
- $\mathbf{s}(x)$ *jest najdłuższym sufiksem (końcowym fragmentem) ciągu* x*, dla którego* $\mathbf{W}(x) \neq \mathbf{W}(\mathbf{s}(x))$*,*
- $\wedge$ *oznacza koniunkcję logiczną, 1 — prawdę, 0 — fałsz, a* $|Z|$ *— liczność zbioru* Z*.*

Na przykład dla ciągu $x = (2,3,7,2,7,4,7,2,4)$ *mamy:*

$$\mathbf{W}(x) = \{2,3,4,7\}, \quad \mathbf{p}(x) = (2,3,7,2,7), \quad \mathbf{s}(x) = (7,2,7,4,7,2,4).$$

Dla bardzo dużych danych program obliczający funkcję bezpośrednio z definicji jest zdecydowanie zbyt wolny. Twoim zadaniem jest jak największe przyspieszenie obliczania tej funkcji.

Napisz program, który wczyta ze standardowego wejścia kilka par ciągów (x,y) *i wypisze na standardowe wyjście wartości* $\mathbf{F}(x,y)$ *dla każdej pary wczytanych ciągów.*

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą k $(1 \leqslant k \leqslant 13)$*, oznaczającą liczbę par ciągów do przeanalizowania. Kolejne* $3k$ *wierszy zawiera opisy przypadków testowych. Pierwszy wiersz każdego opisu zawiera dwie liczby całkowite* n *oraz* m $(1 \leqslant n,m \leqslant 100\,000)$*, oddzielone pojedynczym odstępem i oznaczające długości pierwszego i drugiego ciągu. Drugi wiersz zawiera* n *liczb całkowitych* x_i $(1 \leqslant x_i \leqslant 100)$*, pooddzielanych pojedynczymi odstępami i opisujących ciąg* x*. Trzeci wiersz zawiera* m *liczb całkowitych* y_i $(1 \leqslant y_i \leqslant 100)$*, pooddzielanych pojedynczymi odstępami i opisujących ciąg* y*.*

Wyjście

Wyjście powinno składać się z k wierszy; i-ty wiersz (dla $1 \leqslant i \leqslant k$) powinien zawierać jedną liczbę całkowitą — 0 lub 1 — oznaczającą wartość wyrażenia $\mathbf{F}(x,y)$ dla i-tego przypadku testowego.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`2`	`0`
`4 5`	`1`
`3 1 2 1`	
`1 3 1 2 1`	
`7 7`	
`1 1 2 1 2 1 3`	
`1 1 2 1 3 1 3`	

Rozwiązanie

Wstęp

Zadanie o *Przyspieszeniu algorytmu* jest dosyć abstrakcyjne i jego treść nie zawiera żadnej „bajtockiej" fabuły, co sprawiło zawodnikom pewną dodatkową trudność. Było to również pierwsze zadanie w zawodach Olimpiady Informatycznej, w którego treści podany był algorytm i należało napisać program obliczający wynik tego algorytmu. Główną trudnością zadania było zrozumienie struktury działania podanego algorytmu, nie wiedząc nic o tym, jaki sens ma jego wynik (co było do rozwiązania zupełnie niepotrzebne).

Efektywne rozwiązanie wymagało strukturalnej zmiany nieefektywnego algorytmu na algorytm dający takie same wyniki, ale efektywniejszy. Po dokładnym zrozumieniu problem sprowadza się nieoczekiwanie do wielokrotnego sortowania trójek niedużych liczb całkowitych.

Ponieważ w zadaniu występują prefiksy, sufiksy i (w sposób niejawny) podsłowa, wygodniej jest omawiać rozwiązanie, interpretując ciągi jako słowa składające się z symboli (z elementów ciągu).

Rozwiązania brutalne

Najprostszym rozwiązaniem zadania jest implementacja funkcji **F** bezpośrednio z definicji rekurencyjnej, ale daje to algorytm o wykładniczej złożoności czasowej, o czym można się przekonać, analizując działanie takiego algorytmu dla danych $x = y = (1,2,\ldots,n)$. Najprostszą taką implementację można znaleźć w pliku `przs0.cpp`, a nieco ulepszone wersje w plikach `przs1.cpp` i `przs2.pas`.

Algorytm wielomianowy możemy otrzymać, stosując metodę *tablicowania* (czyli spamiętywania lub programowania dynamicznego) — obliczone wartości funkcji **F** zapamiętu-

jemy w tablicy, dzięki czemu nie musimy wywoływać funkcji wielokrotnie dla tych samych par argumentów. Algorytm taki już nie jest wykładniczy, ale jego złożoność czasowa jest wciąż istotnie zbyt duża — rzędu $O(n^2m^2(n+m))$. Faktycznie, na pierwszym z argumentów **F** zawsze występuje jakieś *podsłowo* (czyli spójny fragment) słowa x, a na drugim jakieś podsłowo słowa y, co daje $O(n^2m^2)$ możliwych par argumentów, a obliczenie dla każdej pary można wykonać w czasie $O(n+m)$ (przy założeniu, że liczba liter słów x i y jest ograniczona przez stałą). Implementacje takiego podejścia można znaleźć w plikach `przs3.cpp` i `przs4.pas`.

Rozwiązanie wzorcowe

Niech A będzie alfabetem — zbiorem symboli występujących w obu słowach. Istotnym parametrem w zadaniu jest liczba $K = |A|$ (będziemy zakładać dla uproszczenia, że elementami A są liczby od 1 do K) — z ograniczeń z treści zadania wynika, że $K \leqslant 100$. Słowo puste będziemy oznaczać jako ε.

Zauważmy, że funkcję **F** możemy traktować jako relację. Powiemy, że dwa słowa x i y są w tej relacji (co zapisujemy jako $x \approx y$), jeśli $\mathbf{F}(x,y) = 1$. Okazuje się, że ta relacja jest relacją równoważności[1], co można uzasadnić, korzystając z faktu, że jeżeli $\mathbf{F}(x,y) = 1$, to ciąg argumentów na pierwszej współrzędnej **F** zależy tylko od słowa x (i symetrycznie dla y). Relacja ta wyznacza podział zbioru wszystkich słów nad alfabetem A na klasy równoważności[2] — wiedząc, do jakich klas należą słowa x i y, możemy od razu obliczyć $\mathbf{F}(x,y)$.

Aby istotnie przyspieszyć nasz algorytm, zastosujemy zatem podejście natury strukturalnej: zastąpimy wejściową funkcję **dwuargumentową** funkcją (faktycznie tablicą) **jednoargumentową** obliczającą numer pewnej klasy równoważności.

Oznaczmy przez L_k (*k-ta warstwa*) zbiór podsłów u początkowych słów x i y, takich że $|\mathbf{W}(u)| = k$. Zauważmy, że dla u, v należących do różnych warstw zachodzi $\mathbf{F}(u,v) = 0$, więc zasadniczo interesuje nas sprawdzanie, czy $u \approx v$ dla u i v należących do tej samej warstwy. Zamiast zajmować się funkcją dwuargumentową **F**, będziemy obliczać tablicę jednoargumentową $NUM[u]$, określającą numer klasy równoważności, do której należy u w swojej warstwie. Innymi słowy, dla u, v należących do tej samej warstwy będzie zachodziło:

$$\mathbf{F}(u,v) = 1 \quad \Longleftrightarrow \quad NUM[u] = NUM[v].$$

Dla słowa u zdefiniujmy jego δ-*reprezentację* $\delta(u)$ jako uporządkowaną trójkę obiektów:

$$\delta(u) \;=\; (p,\, a,\, q),$$

przy czym:

- $p = \mathbf{p}(u)$; $q = \mathbf{s}(u)$;

[1] *Relacja równoważności* to taka relacja pomiędzy elementami pewnego zbioru, która jest *zwrotna* (czyli każdy element jest sam ze sobą w relacji), *symetryczna* (jeżeli a jest w relacji z b, to b jest w relacji z a) oraz *przechodnia* (jeżeli a jest w relacji z b oraz b jest w relacji z c, to również a jest w relacji z c).

[2] *Klasa równoważności* (inaczej klasa abstrakcji) to maksymalny ze względu na zawieranie podzbiór elementów, którego każde dwa elementy są ze sobą w danej relacji równoważności.

- $a \in A$ i pa (słowo p z dopisanym symbolem a na końcu) jest prefiksem u.

Przykład.

$$\delta(ababbbcbcbc) \;=\; (ababbb,\; c,\; bbbcbcbc).$$

Zauważmy, że dla $\delta(u) = (p,a,q)$, jeżeli słowo u należy do k-tej warstwy, to słowa p i q należą do warstwy o numerze $k-1$ oraz $\mathbf{W}(u) = \mathbf{W}(p) \cup \{a\}$.

Przypomnijmy, że chcemy sprowadzić problem do obliczania tablicy NUM, która dla słów u,v z tej samej warstwy spełnia $(u \approx v) \iff (NUM[u] = NUM[v])$ oraz zbiór wartości w tej tablicy składa się z małych liczb całkowitych (konkretniej będzie on postaci $\{1,2,\ldots,w\}$). W tym celu, zamiast δ-reprezentacji $\delta(u) = (p,a,q)$ będziemy używać jej skompresowanej wersji (oznaczanej Δ), będącej trójką liczb naturalnych:

$$\Delta(u) \;=\; (NUM[p],\; a,\; NUM[q]).$$

Okazuje się teraz, że skompresowana postać δ-reprezentacji posiada następującą, istotną dla nas własność. Niech x i y należą do tej samej warstwy oraz $\delta(x) = (p,a,q)$, $\delta(y) = (p',a',q')$. Wtedy:

$$\begin{aligned}
\mathbf{F}(x,y)=1 \quad &\iff \quad (\mathbf{F}(p,p')=1 \;\wedge\; \mathbf{F}(q,q')=1 \;\wedge\; \mathbf{W}(x)=\mathbf{W}(y)) && (1)\\
&\iff \quad (\mathbf{F}(p,p')=1 \;\wedge\; \mathbf{F}(q,q')=1 \;\wedge\; \mathbf{W}(p)\cup\{a\}=\mathbf{W}(p')\cup\{a'\}) && (2)\\
&\iff \quad (p\approx p' \;\wedge\; q\approx q' \;\wedge\; a=a') && (3)\\
&\iff \quad (NUM[p]=NUM[p'] \;\wedge\; NUM[q]=NUM[q'] \;\wedge\; a=a') && (4)\\
&\iff \quad \Delta(x)=\Delta(y). && (5)
\end{aligned}$$

Jedyne przejście niewynikające bezpośrednio z definicji to (2)–(3). Korzystamy w nim z faktu, że jeśli $\mathbf{F}(p,p') = 1$, to $\mathbf{W}(p) = \mathbf{W}(p')$ i ponadto $\mathbf{W}(p) \cap \{a\} = \varnothing$ oraz $\mathbf{W}(p') \cap \{a'\} = \varnothing$.

Powyższą własność możemy wykorzystać do obliczania wartości NUM warstwa po warstwie. Aby wyznaczyć numerację NUM dla L_k, możemy posortować elementy k-tej warstwy względem ich wartości Δ. Wówczas elementy z tą samą wartością Δ będą stanowiły spójny fragment posortowanej listy trójek. Przeglądając ją, możemy łatwo przydzielić im kolejne numery w tablicy NUM.

Do sortowania będziemy używać sortowania pozycyjnego (Radix Sort), którego opis Czytelnik może znaleźć np. w książce [20]. Na potrzeby tego zadania wystarczy wiedzieć, że działa ono w czasie $O(n+m+K)$.

Nasz algorytm możemy więc opisać nieformalnie w następujący sposób:

```
1: { numeracja warstwy zerowej: }
2: NUM[ε] := 1;
3:
4: for k := 1 to K do
5:     { numeracja k-tej warstwy: }
6:     wygeneruj elementy L_k;
7:     oblicz wartości Δ dla elementów L_k;
```

posortuj L_k względem wartości Δ;
kolejnym spójnym fragmentom L_k o jednakowej wartości Δ
przypisz kolejne liczby naturalne w tablicy NUM;

Implementacja w czasie $O(n^2)$

Skonstruowaliśmy już pewien abstrakcyjny algorytm, lecz brakuje nam sensownej implementacji podziału podsłów na warstwy i obliczania wartości Δ. Odtąd będziemy zakładać, że rozważamy tylko jedno słowo i jego podsłowa — możemy na przykład dopisać do początkowego słowa x słowo y, oddzielając je jakimś separatorem # (symbolem niewystępującym w x i y). Tak więc w dalszym tekście opiszemy, jak zaimplementować algorytm obliczający tablicę NUM dla podsłów pojedynczego słowa x o długości n.

Podsłowa możemy utożsamiać z przedziałami postaci $[i, j]$ dla $1 \leqslant i \leqslant j \leqslant n$. Każdy taki przedział reprezentuje podsłowo $x[i..j] = x_i x_{i+1} \ldots x_j$.

Wprowadźmy tablicę

$$ILE[i, j] = |\mathbf{W}(x[i..j])| \text{ dla } 1 \leqslant i \leqslant j \leqslant n.$$

Inaczej mówiąc, $ILE[i, j]$ jest liczbą różnych symboli w podsłowie $x[i..j]$. Na podstawie wartości $ILE[i, j]$ możemy stablicować wartości funkcji **p** oraz **s** (w tablicach dwuwymiarowych, podobnie jak w przypadku $ILE[i, j]$). Mając wyznaczone te wartości, możemy już zrealizować omówiony wcześniej ogólny schemat rozwiązania.

Wszystkie te obliczenia można wykonać w czasie $O(n^2)$, co zostawiamy jako proste ćwiczenie, tym bardziej że w następnej części dokładniej opiszemy, jak to wszystko zrobić w czasie $O(nK)$. Dodajmy tylko, że implementacje stosownego rozwiązania można znaleźć w plikach `przs7.cpp` i `przs8.pas`.

Implementacja w czasie $O(nK)$

Implementacja taka ma sens, gdy K jest znacznie mniejsze niż n, a taka sytuacja ma miejsce w naszym zadaniu. W tym rozwiązaniu ograniczymy zbiór podsłów, z którymi mamy do czynienia. Motywacją do tego jest definicja funkcji **p** oraz **s** — wartościami tych funkcji są szczególne słowa odpowiadające szczególnym przedziałom, które nazwiemy *k-przedziałami*.

Przedział $[i, j]$ nazwiemy k-przedziałem, gdy $x[i..j]$ jest najdłuższym prefiksem słowa $x[i..n]$ zawierającym dokładnie k różnych symboli (co zapisujemy jako $j = PREF_k[i]$) lub $x[i..j]$ jest najdłuższym sufiksem słowa $x[1..j]$ zawierającym dokładnie k różnych symboli (co zapisujemy jako $i = SUF_k[j]$). Przyjmijmy, że $SUF_k[j] = PREF_k[i] = 0$, jeśli wartości te nie są zdefiniowane przez k-przedziały, czyli $|\mathbf{W}(x[1..j])| < k$ lub $|\mathbf{W}(x[i..n])| < k$.

W szczególności, jeśli rozpatrujemy słowo $x\#y$, gdzie # jest separatorem, to przedział odpowiadający x jest k-przedziałem, gdyż x jest najdłuższym prefiksem całego słowa o $|\mathbf{W}(x)|$ różnych literach (formalnie $PREF_{|\mathbf{W}(x)|}[1] = |x|$). Z analogicznych powodów przedział odpowiadający y też jest k-przedziałem.

Okazuje się, że wszystkie przedziały z k-tej warstwy osiągalne za pomocą operacji **p** oraz **s** to dokładnie k-przedziały reprezentowane przez tablice $PREF_k$ i SUF_k. Pokażemy teraz, jak obliczyć te tablice dla ustalonego k w czasie liniowym względem n.

W algorytmie użyjemy struktury danych Z reprezentującej *multizbiór* (czyli zbiór z powtórzeniami) symboli alfabetu A. Z implementujemy jako tablicę $Z[1..K]$ rozmiaru K, zliczającą krotności poszczególnych symboli. Dodatkowo pamiętamy liczbę niezerowych komórek tej tablicy — $|Z|$, czyli rozmiar *zbioru* odpowiadającego Z.

Wstawienie elementu do Z implementujemy jako zwiększenie o jeden odpowiedniego licznika w tablicy; powoduje to zwiększenie $|Z|$, jeżeli licznik ten był w tym momencie równy zeru. Podobnie, usunięcie elementu polega na zmniejszeniu odpowiedniego licznika o jeden i zmniejszeniu $|Z|$ w przypadku wyzerowania tego licznika. W ten sposób każda z tych operacji, podobnie jak wyznaczenie wartości $|Z|$, może zostać wykonana w czasie stałym.

Poniższy pseudokod pokazuje, jak za pomocą multizbioru Z obliczyć $PREF_k$ w czasie $O(n)$; obliczenie SUF_k może zostać wykonane analogicznie.

```
1: Z := ∅;
2: j := 1;
3: for i := 1 to n do
4:    if i > 1 then Z := Z \ {w[i−1]};
5:    while j < n and |Z ∪ {w[j+1]}| ⩽ k do
6:       j := j + 1;
7:       Z := Z ∪ {w[j]};
8:    if |Z| = k then PREF_k[i] := j;
```

Oszacowanie $O(n+K) = O(n)$ na złożoność czasową powyższego algorytmu wynika z tego, że łączna liczba obrotów pętli **while** jest mniejsza niż n, gdyż w każdym obrocie wartość zmiennej j wzrasta o 1.

Jak zapowiadaliśmy, mając wyznaczone tablice $PREF_k$ i SUF_k, można już łatwo obliczać wartości funkcji **p** i **s**. Faktycznie, jeżeli $|\mathbf{W}(x[i..j])| = k$, to

$$\mathbf{p}(x[i..j]) = x[i..PREF_{k-1}[i]] \quad \text{oraz} \quad \mathbf{s}(x[i..j]) = x[SUF_{k-1}[j]..j].$$

Używając tych wartości, możemy z kolei obliczać wartości NUM dla wszystkich podsłów odpowiadających k-przedziałom.

Nasz algorytm działa teraz w czasie $O(nK)$ i pamięci również $O(nK)$. Możemy jednak bardzo łatwo zmniejszyć zużycie pamięci do $O(n)$. Wystarczy jedynie trzymać w algorytmie dane związane z bieżącą warstwą L_k i poprzednią warstwą L_{k-1}. Pozostałe dane można usuwać na bieżąco.

W ten sposób uzyskujemy rozwiązanie wzorcowe, zaimplementowane w plikach `prz.cpp`, `prz0.pas` i `prz1.java`.

Testy

Zadanie było sprawdzane na 10 zestawach danych, z których każdy składał się z kilku bądź kilkunastu pojedynczych testów zawartych w jednym pliku. Z tego względu zadanie to miało rekordowe w historii Olimpiady Informatycznej limity czasowe, rzędu kilku minut, co stanowiło kolejną jego nietypową właściwość.

Ponieważ zupełnie losowe testy z ogromnym prawdopodobieństwem dają odpowiedź 0, większość testów była generowana dosyć skomplikowanymi metodami rekurencyjnymi, jakby „naśladującymi" kolejne wywołania rekurencyjne podanego w treści zadania algorytmu obliczania funkcji **F**.

Poniższa tabelka podsumowuje parametry użytych testów.

Nazwa	k	n, m	K
prz1.in	11	≈ 15	4–7
prz2.in	11	≈ 30	6–15
prz3.in	13	≈ 50	15–25
prz4.in	13	≈ 60	10–30
prz5.in	11	$\approx 50\,000$	80–100
prz6.in	9	$\approx 60\,000$	100
prz7.in	8	$\approx 70\,000$	100
prz8.in	8	$\approx 80\,000$	100
prz9.in	8	$\approx 90\,000$	100
prz10.in	8	$\approx 100\,000$	100

Czy funkcja **F** wzięła się z sufitu?

Co prawda do rozwiązania niniejszego zadania zrozumienie istoty funkcji **F** faktycznie okazało się zupełnie niepotrzebne, jednakże na koniec pragniemy wyjaśnić, że funkcja ta nie wzięła się znikąd. Otóż można pokazać, że $\mathbf{F}(x,y) = 1$ wtedy i tylko wtedy, gdy słowo x można przekształcić w y za pomocą pewnej liczby operacji, z których każda polega na:

1. zastąpieniu podsłowa słowa x będącego kwadratem u^2 (sklejenie u z u) pojedynczym słowem u albo
2. zastąpieniu dowolnego podsłowa u słowa x jego kwadratem u^2.

Ciąg takich przekształceń dla drugiego przykładu z treści zadania wygląda następująco (nad strzałkami zapisano numery stosowanych operacji):

$$1\ \underbrace{1\ 2\ 1\ 2}\ 1\ 3 \xrightarrow{1} 1\ 1\ 2\ \underbrace{1\ 3} \xrightarrow{2} 1\ 1\ 2\ 1\ 3\ 1\ 3.$$

Ktoś mógłby zapytać, dlaczego w treści zadania zamiast występującej tam abstrakcyjnej i technicznej definicji nie pojawiła się powyższa, wszak dużo przyjemniejsza i bardziej zrozumiała? Odpowiedź tkwi w dowodzie równoważności tych definicji, który jest istotnie trudniejszy do wymyślenia niż całe rozwiązanie zadania o *Przyspieszeniu algorytmu* w obecnym jego sformułowaniu. A zatem, wbrew pozorom obecna treść tego zadania *ułatwia* jego rozwiązanie!

Czytelników zainteresowanych wspomnianym dowodem zachęcamy do zajrzenia do książki [38] albo poszperania po Internecie pod hasłem *free idempotent monoid*. (Ostrzegamy jednak, że w ten sposób można się „doklikać" do całkiem trudnych i skomplikowanych materiałów).

Jakub Radoszewski, Wojciech Rytter
Treść zadania, Opracowanie

Paweł Parys
Program

Dostępna pamięć: 64 MB.

OI, Etap I, 20.10–17.11.2008

Słonie

W Bajtockim Zoo ma się za chwilę odbyć parada, w której uczestniczyć będą wszystkie znajdujące się w nim słonie. Pracownicy zoo zachęcili już zwierzęta do ustawienia się w jednym rzędzie, gdyż zgodnie z zarządzeniem dyrektora zoo taka powinna być początkowa figura parady.

Niestety, na miejsce przybył sam dyrektor i zupełnie nie spodobała mu się wybrana przez pracowników kolejność słoni. Dyrektor zaproponował kolejność, w której według niego zwierzęta będą się najlepiej prezentować, i wydał pracownikom polecenie poprzestawiania słoni zgodnie z tą propozycją.

Aby uniknąć nadmiernego chaosu tuż przed paradą, pracownicy postanowili przestawiać słonie, zamieniając miejscami kolejno pewne pary słoni. Przestawiane zwierzęta nie muszą sąsiadować ze sobą w rzędzie. Wysiłek potrzebny do nakłonienia słonia do ruszenia się z miejsca jest wprost proporcjonalny do jego masy, a zatem wysiłek związany z zamianą miejscami dwóch słoni o masach m_1 oraz m_2 można oszacować przez $m_1 + m_2$. Jakim minimalnym wysiłkiem pracownicy mogą poprzestawiać słonie tak, aby usatysfakcjonować dyrektora?

Napisz program, który:

- *wczyta ze standardowego wejścia masy wszystkich słoni z zoo oraz aktualną i docelową kolejność słoni w rzędzie,*
- *wyznaczy taki sposób poprzestawiania słoni, który prowadzi od kolejności początkowej do docelowej i minimalizuje sumę wysiłków związanych ze wszystkimi wykonanymi zamianami pozycji słoni,*
- *wypisze sumę wartości tych wysiłków na standardowe wyjście.*

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą n ($2 \leqslant n \leqslant 1\,000\,000$), oznaczającą liczbę słoni w Bajtockim Zoo. Dla uproszczenia zakładamy, że słonie są ponumerowane od 1 do n. Drugi wiersz zawiera n liczb całkowitych m_i ($100 \leqslant m_i \leqslant 6\,500$ dla $1 \leqslant i \leqslant n$), pooddzielanych pojedynczymi odstępami i oznaczających masy poszczególnych słoni (wyrażone w kilogramach).

Trzeci wiersz wejścia zawiera n różnych liczb całkowitych a_i ($1 \leqslant a_i \leqslant n$), pooddzielanych pojedynczymi odstępami i oznaczających numery kolejnych słoni w aktualnym ustawieniu. Czwarty wiersz zawiera n różnych liczb całkowitych b_i ($1 \leqslant b_i \leqslant n$), pooddzielanych pojedynczymi odstępami i oznaczających numery kolejnych słoni w ustawieniu proponowanym przez dyrektora zoo. Możesz założyć, że ustawienia reprezentowane przez ciągi (a_i) oraz (b_i) są różne.

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczbę całkowitą, oznaczającą minimalny łączny wysiłek związany z poprzestawianiem słoni, w wyniku którego z ustawienia reprezentowanego przez (a_i) uzyskuje się ustawienie (b_i).

Przykład

Dla danych wejściowych:

```
6
2400 2000 1200 2400 1600 4000
1 4 5 3 6 2
5 3 2 4 6 1
```

poprawnym wynikiem jest:

```
11200
```

Jeden z najlepszych sposobów poprzestawiania słoni uzyskujemy, zamieniając miejscami kolejno pary słoni:

- *2 i 5 — wysiłek związany z zamianą to $2\,000 + 1\,600 = 3\,600$, a uzyskane ustawienie to* `1 4 2 3 6 5`,
- *3 i 4 — wysiłek to $1\,200 + 2\,400 = 3\,600$, a uzyskane ustawienie to* `1 3 2 4 6 5`,
- *1 i 5 — wysiłek to $2\,400 + 1\,600 = 4\,000$, a uzyskane ustawienie to* `5 3 2 4 6 1`, *czyli ustawienie docelowe.*

Rozwiązanie

Wstęp

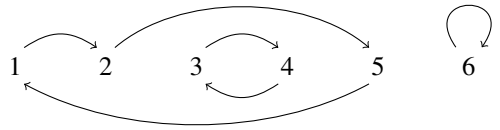
Aby łatwiej wyobrazić sobie zadanie, jakie przed pracownikami zoo postawił dyrektor, spróbujemy przedstawić je graficznie. W tym celu zdefiniujemy funkcję $p\colon \{1,2,\ldots,n\} \to \{1,2,\ldots,n\}$ w następujący sposób:

$$p(b_1) = a_1, \quad p(b_2) = a_2, \quad \ldots, \quad p(b_n) = a_n.$$

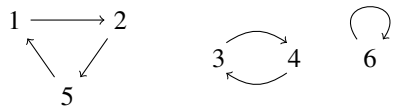
Zauważmy, że wówczas $p(x) = y$ będzie oznaczało, że słoń o numerze x powinien znaleźć się w końcowym ustawieniu w miejscu, które jest aktualnie zajmowane przez słonia o numerze y. Wszystkie liczby b_i są różne, zatem p jest poprawnie zdefiniowaną funkcją, a ponieważ wszystkie liczby a_i są różne, p jest *permutacją* zbioru $\{1,2,\ldots,n\}$. Sytuację z zadania możemy zatem przedstawić w postaci grafu skierowanego, w którym wierzchołkami są numery $1,2,\ldots,n$ słoni, krawędziami natomiast wartości funkcji p (patrz rys. 1).

Dalej, jak każdą permutację, funkcję p można rozłożyć na tak zwane *cykle proste* $C_1, C_2, \ldots, C_c$. W tym celu należy wystartować z dowolnego wierzchołka grafu i podążać po krawędziach, aż dojdzie się z powrotem do tego wierzchołka (dlaczego zawsze trafia

się w końcu w wierzchołek początkowy trasy?), po czym usunąć znaleziony cykl z grafu i kontynuować proces aż do wyczerpania wszystkich wierzchołków — patrz rys. 2.

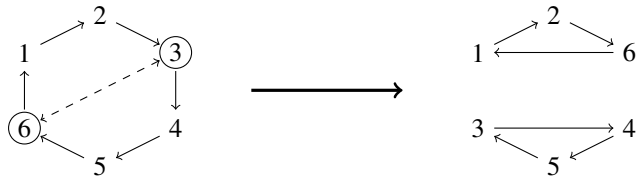


Rys. 1: Graf wyznaczony przez funkcję p dla przykładu z treści zadania. Wierzchołki grafu reprezentują numery słoni, natomiast strzałka z x do y obrazuje relację „słoń x powinien zająć miejsce słonia y".



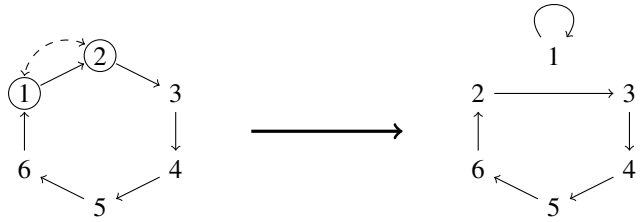
Rys. 2: Rozkład grafu z rys. 1 na cykle proste: trójelementowy, dwuelementowy i jednoelementowy.

Zastanówmy się teraz, jak na tle opisanego rozkładu na cykle proste wygląda operacja zamiany miejscami słoni o numerach e oraz f. Koszt takiej zamiany to $m_e + m_f$. Jeżeli słonie e oraz f znajdują się w tym samym cyklu, to następuje wówczas podział tego cyklu na dwa rozłączne, z których jeden zawiera jednego z tych słoni, a drugi drugiego — patrz rys. 3.



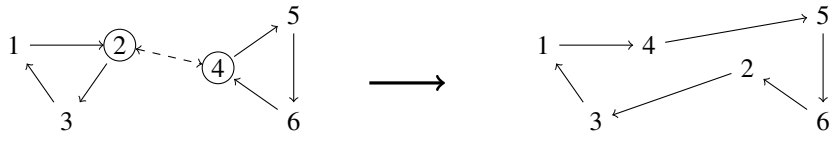
Rys. 3: Zamiana miejscami trzeciego i szóstego słonia w cyklu prowadzi do powstania dwóch cykli trójelementowych.

W szczególności, w wyniku zamiany miejscami dwóch słoni, które sąsiadują na cyklu, jeden z powstałych cykli jest jednoelementowy, co oznacza dokładnie tyle, że po tej zamianie jeden ze słoni znajduje się na swojej pozycji docelowej (rys. 4).



Rys. 4: Zamiana miejscami pierwszego i drugiego słonia w cyklu powoduje, że pierwszy słoń trafia na właściwą pozycję.

Z kolei jeżeli słonie e oraz f należą do różnych cykli, to zamiana ich miejscami powoduje sklejenie tych cykli w jeden (rys. 5).



Rys. 5: Zamiana miejscami słoni należących do różnych cykli.

Rozwiązanie wzorcowe[1]

Przyjmijmy następujące oznaczenia:

- $|C|$ — długość cyklu C, czyli liczba wierzchołków grafu w nim zawartych
- $suma(C)$ — suma mas słoni należących do cyklu C
- $min(C)$ — masa najlżejszego słonia na cyklu C
- min — masa najlżejszego słonia w ogóle.

Naszym celem jest sprowadzenie układu cykli reprezentowanego przez permutację p do takiego, który będzie złożony wyłącznie z cykli jednoelementowych. W rozwiązaniu wzorcowym każdy cykl C rozpatrujemy osobno, a wspomniane przekształcenie realizujemy, stosując do C jedną z następujących metod przestawiania słoni.

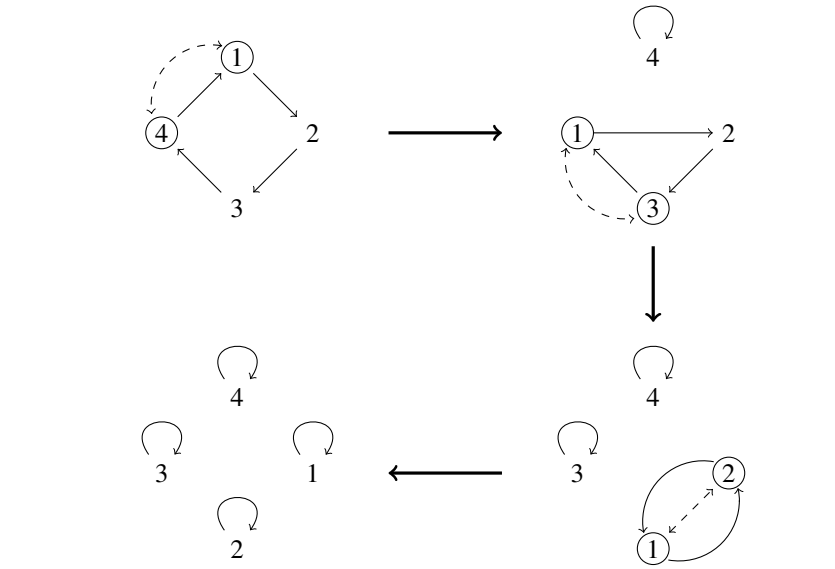
Metoda 1. Porządkujemy cały cykl pojedynczymi zamianami sąsiednich słoni (jak na rys. 4). Za każdym razem zamieniamy najlżejszego słonia z całego cyklu ($min(C)$) z jego poprzednikiem na cyklu, w wyniku czego poprzednik ten trafia na właściwe miejsce (patrz rys. 6). W ten sposób najlżejszego słonia przemieszczamy łącznie $|C|-1$ razy, natomiast każdego z pozostałych słoni z cyklu — dokładnie raz. Łączny koszt porządkowania cyklu tą metodą wynosi zatem:

$$\text{metoda}_1(C) = suma(C) + \big(|C|-2\big)\cdot min(C). \tag{1}$$

Metoda 2. Tym razem postępujemy bardzo podobnie, jednakże do obsłużenia całego cyklu wykorzystujemy globalnie najlżejszego słonia (min). W tym celu zamieniamy go z najlżejszym słoniem cyklu ($min(C)$), a następnie za jego pomocą przestawiamy kolejno wszystkie słonie z cyklu C (poza $min(C)$) jak poprzednio, po czym z powrotem zamieniamy min z $min(C)$. Na ten ciąg zamian można też spojrzeć jak na sklejenie C z cyklem zawierającym min, po którym następuje ustawienie wszystkich słoni z C na właściwych miejscach za pomocą pojedynczych zamian ze słoniem min. W ten sposób najlżejszy słoń w całym zoo zostaje przemieszczony $|C|+1$ razy, najlżejszy w cyklu — 2 razy, a każdy z pozostałych słoni tego cyklu — dokładnie raz. Łączny koszt wszystkich przestawień to wówczas:

$$\text{metoda}_2(C) = suma(C) + min(C) + \big(|C|+1\big)\cdot min. \tag{2}$$

[1] Duża część opisu rozwiązania wzorcowego została zaczerpnięta z pracy [37].



Rys. 6: Porządkowanie czteroelementowego cyklu za pomocą pierwszej metody. Zakładamy, że najlżejszy słoń ma numer 1.

Okazuje się, że w całym rozwiązaniu wystarczy wziąć pod uwagę jedynie dwie opisane metody i dla każdego cyklu w rozkładzie do uporządkowania użyć korzystniejszej z nich[2]. Ostatecznie otrzymujemy następujący minimalny koszt poprzestawiania słoni:

$$\sum_{i=1}^{c} \min(\text{metoda}_1(C_i),\ \text{metoda}_2(C_i)). \tag{3}$$

Implementacja

Jako podsumowanie słownego opisu rozwiązania wzorcowego umieszczamy poniższy jego pseudokod. Łatwo sprawdzić, że cały algorytm ma złożoność czasową $O(n)$. Konkretne implementacje tego algorytmu można znaleźć w plikach `slo.cpp`, `slo1.java` oraz `slo2.pas`.

```
1: { Konstrukcja permutacji p. }
2: for i := 1 to n do p[b_i] := a_i;
3:
4: { Rozkład p na cykle proste. }
5: odw : array[1..n] := (false, false, ..., false);
6: c := 0;
7: for i := 1 to n do
8:    if not odw[i] then begin
9:       c := c + 1; x := i;
```

[2]Dodajmy tylko, że dla niektórych cykli zachodzi $min(C) = min$, i wówczas druga metoda traci sens, jednakże nie trzeba się tym faktem przejmować, gdyż wówczas i tak $\text{metoda}_2(C) > \text{metoda}_1(C)$.

```
10:       while not odw[x] do begin
11:         odw[x] := true;
12:         C_c := C_c ∪ {x};
13:         x := p[x];
14:       end
15:     end
16:
17: { Wyznaczenie parametrów cykli. }
18: min := ∞;
19: for i := 1 to c do begin
20:     suma(C_i) := 0; min(C_i) := ∞;
21:     forall e ∈ C_i do begin
22:       suma(C_i) := suma(C_i) + m_e;
23:       min(C_i) := min(min(C_i), m_e);
24:     end
25:     min := min(min, min(C_i));
26: end
27:
28: { Obliczenie wyniku. }
29: w := 0;
30: for i := 1 to c do begin
31:     metoda_1(C_i) := suma(C_i) + (|C_i| − 2) · min(C_i);
32:     metoda_2(C_i) := suma(C_i) + min(C_i) + (|C_i| + 1) · min;
33:     w := w + min(metoda_1(C_i), metoda_2(C_i));
34: end
35: return w;
```

Uzasadnienie poprawności

Na sam koniec opisu rozwiązania wzorcowego pozostawiliśmy jego dowód poprawności. Jest on, niestety, trochę skomplikowany, jednakże jest to jedyna niezawodna metoda sprawdzenia tego, czy w rozwiązaniu nie zapomnieliśmy o żadnym z przypadków.

Jeżeli C jest cyklem, to przez

$$\text{koszt}(C) = \min(\text{metoda}_1(C),\ \text{metoda}_2(C))$$

oznaczmy koszt uporządkowania tego cyklu w rozwiązaniu wzorcowym. Niech dalej $OPT(p)$ oznacza najmniejszy możliwy koszt poprzestawiania wszystkich słoni zgodnie z zarządzeniem dyrektora. Aby wykazać poprawność rozwiązania wzorcowego, wystarczy udowodnić, że

$$OPT(p) \geqslant \sum_{i=1}^{c} \text{koszt}(C_i). \tag{4}$$

Nierówność (4) udowodnimy przez indukcję względem liczby zamian wykonywanych w rozwiązaniu optymalnym. Przypadek zerowej liczby zamian jest trywialny. Załóżmy

więc, że (4) zachodzi dla wszystkich permutacji, dla których algorytm optymalny wykonuje $k-1$ zamian, i niech p będzie dowolną permutacją, do której uporządkowania algorytm ten potrzebuje k zamian. Musimy udowodnić, że (4) zachodzi także dla p.

Przyjrzyjmy się pierwszemu krokowi rozwiązania optymalnego „OPT" uruchomionego dla p. Załóżmy, że są w nim zamieniane słonie e oraz f. Permutację po ich zamianie oznaczmy przez p' — uporządkowanie jej przez algorytm optymalny wymaga $k-1$ zamian, więc wiadomo, że nierówność (4) zachodzi dla p'. W zależności od wzajemnego położenia e oraz f wyróżniamy teraz kilka przypadków.

Przypadek 1: e i f należą do tego samego cyklu w p. Dla ustalenia uwagi niech $e, f \in C_1$. Po zamianie słoni cykl C_1 rozpada się na dwa mniejsze cykle (patrz rys. 3). Oznaczmy te cykle przez A i B oraz niech $e \in A$ i $f \in B$. Wówczas rozkład p' na cykle proste wygląda tak: $\{A, B, C_2, C_3, \ldots, C_c\}$, a stąd i z (4) dla p' mamy:

$$OPT(p) = OPT(p') + m_e + m_f \geqslant \text{koszt}(A) + \text{koszt}(B) + \sum_{i=2}^{c} \text{koszt}(C_i) + m_e + m_f.$$

Aby pokazać (4) dla p, wystarczy zatem udowodnić, że:

$$\text{koszt}(A) + \text{koszt}(B) + m_e + m_f \geqslant \text{koszt}(C_1). \tag{5}$$

Dalsze uzasadnienie możemy podzielić na trzy podprzypadki:

Przypadek 1.1: $\text{koszt}(A) = \text{metoda}_1(A)$ i $\text{koszt}(B) = \text{metoda}_1(B)$. Korzystając z faktu, że zbiór słoni zawartych w cyklu C_1 jest sumą zbiorów słoni odpowiadających A oraz B, oraz z tego, że każda z wartości $min(A)$, $min(B)$, m_e, m_f jest nie mniejsza niż $min(C_1)$, mamy:

$$\begin{aligned}
&\text{metoda}_1(A) + \text{metoda}_1(B) + m_e + m_f = \\
&\quad = suma(A) + \big(|A| - 2\big) \cdot min(A) + suma(B) + \big(|B| - 2\big) \cdot min(B) + m_e + m_f \geqslant \\
&\quad \geqslant suma(C_1) + \big(|C_1| - 2\big) \cdot min(C_1) = \text{metoda}_1(C_1) \geqslant \text{koszt}(C_1).
\end{aligned}$$

Intuicyjnie, pokazaliśmy, że zamiast rozbijać cykl C_1 na cykle A i B (za pomocą zamiany e z f) i każdy z nich porządkować za pomocą pierwszej metody, można równie dobrze od razu uporządkować cały cykl C_1 za pomocą tej metody.

Przypadek 1.2: $\text{koszt}(A) = \text{metoda}_1(A)$ i $\text{koszt}(B) = \text{metoda}_2(B)$ (lub odwrotnie). Podobnie jak poprzednio, w poniższej nierówności liczba składników nie zmienia się, a jedynie zastępujemy cięższe słonie przez $min(C_1)$ lub min:

$$\begin{aligned}
&\text{metoda}_1(A) + \text{metoda}_2(B) + m_e + m_f = \\
&\quad = suma(A) + \big(|A| - 2\big) \cdot min(A) + suma(B) + min(B) + \big(|B| + 1\big) \cdot min + m_e + m_f \geqslant \\
&\quad \geqslant suma(C_1) + min(C_1) + \big(|C_1| + 1\big) \cdot min = \text{metoda}_2(C_1) \geqslant \text{koszt}(C_1).
\end{aligned}$$

Tym razem intuicja stojąca za powyższym ciągiem przekształceń jest taka, że zamiast wprowadzać najlżejszego słonia tylko do cyklu B, można go wprowadzić od razu do całego C_1, co nie przynosi żadnej straty, a może się dodatkowo opłacić przy porządkowaniu fragmentu C_1 odpowiadającego cyklowi A.

Przypadek 1.3: $\text{koszt}(A) = \text{metoda}_2(A)$ **i** $\text{koszt}(B) = \text{metoda}_2(B)$**.** I tym razem do wyprowadzenia (5) wykorzystujemy te same spostrzeżenia. W tym przypadku strata wynikła z rozbicia cyklu C_1 jest ewidentna: intuicyjnie, zamiast wprowadzać słonia *min* osobno do każdego z cykli A, B, można na samym początku wprowadzić go do całego C_1:

$$\begin{aligned}&\text{metoda}_2(A)+\text{metoda}_2(B)+m_e+m_f=\\&\quad=suma(A)+min(A)+\big(|A|+1\big)\cdot min+suma(B)+min(B)+\big(|B|+1\big)\cdot min+m_e+m_f\geqslant\\&\quad\geqslant suma(C_1)+min(C_1)+\big(|C_1|+1\big)\cdot min=\text{metoda}_2(C_1)\geqslant\text{koszt}(C_1).\end{aligned}$$

Przypadek 2: e **i** f **należą do różnych cykli.** Dla ustalenia uwagi niech tym razem $e\in C_1, f\in C_2$. Po zamianie słoni cykle C_1 i C_2 łączą się w jeden cykl A (patrz rys. 5), stąd rozkład p' na cykle to: $\{A,C_3,C_4,\ldots,C_c\}$. Stąd i z (4) dla p' otrzymujemy:

$$OPT(p)=OPT(p')+m_e+m_f\geqslant\text{koszt}(A)+\sum_{i=3}^{c}\text{koszt}(C_i)+m_e+m_f.$$

Aby pokazać (4) dla p, wystarczy udowodnić, że:

$$\text{koszt}(A)+m_e+m_f\geqslant\text{koszt}(C_1)+\text{koszt}(C_2). \tag{6}$$

Bez straty ogólności załóżmy, że $min(C_1)\leqslant min(C_2)$, czyli $min(A)=min(C_1)$. Tym razem mamy dwa podprzypadki:

Przypadek 2.1: $\text{koszt}(A) = \text{metoda}_1(A)$**.** Korzystając z tego, że cykl A jest sumą (pod względem zbioru zawartych w nim słoni) cykli C_1 oraz C_2, a także z założenia $min(A)=min(C_1)$ oraz nierówności: $m_f\geqslant min(C_2)$ i $m_e, min(A)\geqslant min$, otrzymujemy następujący ciąg przekształceń:

$$\begin{aligned}&\text{metoda}_1(A)+m_e+m_f=\\&\quad=suma(A)+\big(|A|-2\big)\cdot min(A)+m_e+m_f\geqslant\\&\quad\geqslant suma(C_1)+\big(|C_1|-2\big)\cdot min(C_1)+suma(C_2)+min(C_2)+\big(|C_2|+1\big)\cdot min=\\&\quad=\text{metoda}_1(C_1)+\text{metoda}_2(C_2)\geqslant\text{koszt}(C_1)+\text{koszt}(C_2).\end{aligned}$$

Intuicja tym razem jest taka, że zamiast łączyć cykle C_1 i C_2 i porządkować otrzymany cykl A metodą 1, można sam cykl C_1 uporządkować tą metodą (tu nic nie tracimy, gdyż $min(A)=min(C_1)$), natomiast cykl C_2 połączyć nie z C_1, ale z cyklem zawierającym najlżejszego słonia *min*, co odpowiada zastosowaniu do C_2 drugiej metody porządkowania.

Przypadek 2.2: $\text{koszt}(A) = \text{metoda}_2(A)$**.** Podobnie jak poprzednio, na mocy warunków $min(A)=min(C_1)$, $m_f\geqslant min(C_2)$ oraz $m_e\geqslant min$, mamy:

$$\begin{aligned}&\text{metoda}_2(A)+m_e+m_f=\\&\quad=suma(A)+min(A)+\big(|A|+1\big)\cdot min+m_e+m_f\geqslant\\&\quad\geqslant suma(C_1)+min(C_1)+\big(|C_1|+1\big)\cdot min+suma(C_2)+min(C_2)+\big(|C_2|+1\big)\cdot min=\\&\quad=\text{metoda}_2(C_1)+\text{metoda}_2(C_2)\geqslant\text{koszt}(C_1)+\text{koszt}(C_2).\end{aligned}$$

Intuicyjnie, nie opłaca się ponosić kosztu połączenia cykli C_1 i C_2, żeby potem uporządkować wynikowy cykl A za pomocą najlżejszego słonia *min*, gdyż na pewno nie gorszym rozwiązaniem jest wprowadzenie słonia *min* do każdego z cykli C_1, C_2 z osobna. Innymi słowy, $min + min(A) + m_e + m_f \geqslant 2 \cdot min + min(C_1) + min(C_2)$.

Inne rozwiązania

Wśród potencjalnych rozwiązań błędnych można wyróżnić przede wszystkim takie, które przy porządkowaniu cykli zapominają o jednej z metod: drugiej (plik `slob1.cpp`, 20% punktów) lub pierwszej (plik `slob2.cpp`, 10% punktów). Przypomnijmy, że takich błędów można uniknąć, jeżeli przeprowadzi się dowód poprawności rozwiązania lub chociażby sprawdzi poprawność swojego rozwiązania na większej grupie losowych testów, porównując jego wyniki z jakimkolwiek rozwiązaniem na pewno poprawnym, chociażby wykładniczym. Innym błędem było wykonywanie wszystkich obliczeń na liczbach całkowitych 32-bitowych — takie rozwiązanie, wskutek błędu przepełnienia typu, zdobywało 60% punktów za to zadanie (implementacja w pliku `slob3.cpp`).

Wśród rozwiązań wolniejszych można wymienić rozwiązanie kwadratowe względem n (plik `slos1.cpp`), będące nieefektywną implementacją rozwiązania wzorcowego i zdobywające 40% punktów, oraz zaimplementowane w pliku `slos2.cpp` i uzyskujące 10% punktów rozwiązanie siłowe, rozważające wszystkie możliwości zamian słoni, poczynając od najtańszych.

Testy

Zadanie było sprawdzane na 10 zestawach danych testowych. Wszystkie testy za wyjątkiem tych z grupy `b` to testy w jakimś sensie losowe. Większość testów zawiera losową permutację p słoni. Ponieważ zupełnie losowa permutacja zawiera statystycznie stosunkowo mało cykli, to w testach `4` i `10a` wygenerowano permutacje o dużych liczbach cykli. Poza tym specjalną postać mają testy `9a` oraz te z grupy `b` — patrz opisy poniżej.

W następującym zestawieniu testów n oznacza liczbę słoni, natomiast pozostałe parametry charakteryzują własności permutacji p: c_1 to liczba cykli jednoelementowych (czyli takich, które nie wymagają żadnych zamian), m_1 to liczba cykli, których optymalne uporządkowanie otrzymuje się za pomocą metody 1, natomiast m_2 to liczba cykli, które należy porządkować metodą 2.

Nazwa	n	c_1	m_1	m_2	Opis
slo1.in	10	1	1	1	test losowy
slo2.in	100	2	4	1	test losowy
slo3.in	1 000	2	8	0	test losowy
slo4.in	10 000	24	55	24	test losowy o zwiększonej liczbie cykli
slo5.in	100 000	2	8	1	test losowy

Nazwa	n	c_1	m_1	m_2	Opis
slo6.in	920 000	1	9	7	test losowy
slo7.in	960 000	2	6	11	test losowy
slo8a.in	980 000	0	6	8	test losowy
slo8b.in	980 000	979 998	1	0	potrzeba tylko jednej zamiany
slo9a.in	1 000 000	904 788	44 788	424	90% słoni na swoim miejscu
slo9b.in	1 000 000	0	500 000	0	wszystkie cykle dwuelementowe
slo10a.in	1 000 000	307	330	1212	test losowy o zwiększonej liczbie cykli
slo10b.in	1 000 000	0	1	0	jeden długi cykl

Marian M. Kędzierski, Jakub Radoszewski — Treść zadania, Opracowanie
Tomasz Kulczyński — Program

Dostępna pamięć: 64 MB. *OI, Etap I, 20.10–17.11.2008*

Straż pożarna

W stolicy Bajtocji, Bajtawie, ulice mają bardzo regularny układ. Wszystkie biegną albo z północy na południe, albo z zachodu na wschód. Łatwo zauważyć, że każda ulica z północy na południe przecina każdą ulicę z zachodu na wschód w dokładnie jednym miejscu. Ponadto, wzdłuż każdej ulicy kolejne skrzyżowania są odległe o dokładnie 1 km.

W Bajtawie jest z zabytkowych budynków, z których każdy znajduje się przy jednym ze skrzyżowań. Radzie Miejskiej bardzo zależy na ochronie tych unikalnych zabytków, dlatego postanowiono wybudować w mieście dwa duże posterunki straży pożarnej. Każdy z zabytków będzie chroniony przez posterunek jemu najbliższy; w przypadku równych odległości od każdego z posterunków, budynek będzie pod ochroną ich obu.

Zabudowa w Bajtawie jest bardzo gęsta. Nie należy więc patrzeć na odległość do zabytków w linii prostej. Zamiast tego, jako odległość od posterunku do zabytku należy przyjąć długość najkrótszej trasy biegnącej ulicami.

Rada Miejska przygotowała kilka projektów lokalizacji posterunków straży. Zostałeś poproszony o wyznaczenie, dla każdego z nich, liczby zabytków chronionych: tylko przez pierwszy posterunek, tylko przez drugi posterunek oraz przez oba posterunki.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się cztery liczby całkowite n, m, z oraz p ($1 \leqslant n,m \leqslant 1\,000\,000\,000$, $1 \leqslant z,p \leqslant 100\,000$) pooddzielane pojedynczymi odstępami, oznaczające odpowiednio: liczbę ulic biegnących z północy na południe, liczbę ulic biegnących z zachodu na wschód, liczbę zabytkowych budynków w Bajtawie oraz liczbę projektów zaproponowanych przez Radę Miejską. Ulice biegnące z północy na południe są ponumerowane od 1 do n, w kierunku z zachodu na wschód. Ulice biegnące z zachodu na wschód są ponumerowane od 1 do m, w kierunku z północy na południe. Skrzyżowaniu x-tej ulicy biegnącej z północy na południe z y-tą ulicą biegnącą z zachodu na wschód dla uproszczenia przypisujemy współrzędne (x,y).

W każdym z kolejnych z wierszy znajdują się dwie liczby całkowite x_i oraz y_i ($1 \leqslant x_i \leqslant n$, $1 \leqslant y_i \leqslant m$) oddzielone pojedynczym odstępem i oznaczające współrzędne i-tego zabytku. Żadna para zabytków nie znajduje się przy tym samym skrzyżowaniu.

Każdy z kolejnych p wierszy zawiera jedną propozycję Rady Miejskiej — cztery liczby całkowite $x_{j,1}$, $y_{j,1}$, $x_{j,2}$, $y_{j,2}$ pooddzielane pojedynczymi odstępami, $1 \leqslant x_{j,1},x_{j,2} \leqslant n$, $1 \leqslant y_{j,1},y_{j,2} \leqslant m$, $(x_{j,1},y_{j,1}) \neq (x_{j,2},y_{j,2})$. Współrzędne $(x_{j,1},y_{j,1})$ oraz $(x_{j,2},y_{j,2})$ opisują skrzyżowania, przy których mają być umiejscowione posterunki straży zgodnie z j-tą propozycją ($1 \leqslant j \leqslant p$).

Wyjście

Twój program powinien wypisać na standardowe wyjście p wierszy. W j-tym wierszu powinny się znaleźć trzy liczby całkowite, oznaczające: liczbę zabytków chronionych tylko przez pierwszy

posterunek z j-tej propozycji Rady Miejskiej, liczbę zabytków chronionych tylko przez drugi posterunek oraz liczbę budynków chronionych przez oba posterunki. Liczby te powinny być oddzielone pojedynczymi odstępami.

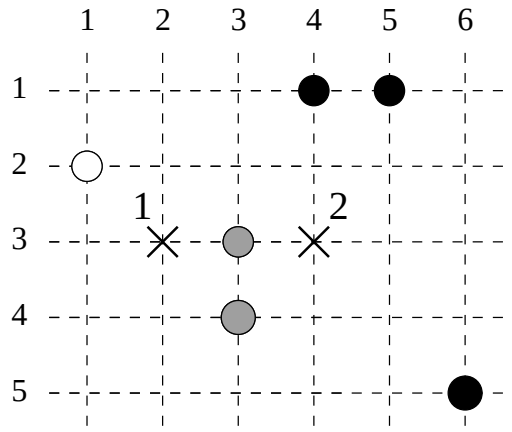
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
1 3 2
```



Na rysunku linie przerywane przedstawiają ulice, kółka — lokalizacje zabytków, a krzyżyki — proponowane lokalizacje posterunków straży pożarnej. Białe kółko przedstawia zabytek chroniony przez pierwszy posterunek, czarne kółka — przez drugi posterunek, natomiast szare kółka — przez oba posterunki.

Rozwiązanie

Wstęp

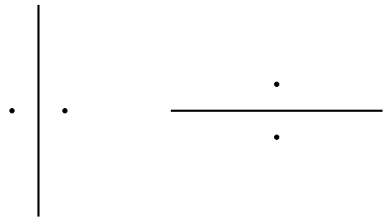
Sformułujmy treść naszego zadania w języku geometrii. Na płaszczyźnie znajduje się z punktów $P_1, P_2, \ldots, P_z$ reprezentujących zabytki Bajtawy. Dodatkowo, danych jest p par punktów S_j, T_j będących propozycjami lokalizacji posterunków straży pożarnej. Naszym zadaniem jest wyznaczenie, dla każdej propozycji, liczby zabytków położonych: bliżej S_j niż T_j (obszar A), bliżej T_j niż S_j (obszar B) oraz w jednakowej odległości do S_j i T_j (obszar R). Zgodnie z treścią zadania, odległość między punktami (x_1, y_1) i (x_2, y_2) jest mierzona w tzw. *metryce miejskiej* (nazywanej też metryką Manhattan), czyli:

$$d\big((x_1, y_1), (x_2, y_2)\big) = |x_1 - x_2| + |y_1 - y_2|.$$

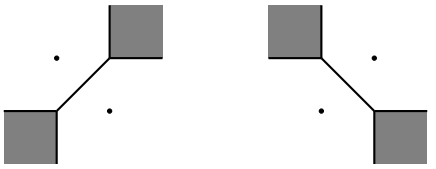
Tak sformułowane zadanie bardzo łatwo rozwiązać siłowo, po prostu wyznaczając wszystkie potrzebne odległości i porównując je. W ten sposób uzyskuje się rozwiązanie o złożoności czasowej $O(p \cdot z)$, czyli mało efektywne, biorąc pod uwagę ograniczenia z zadania. Jego implementacje można znaleźć w plikach `strs1.cpp` i `strs2.pas`; zdobywało ono na zawodach 20% punktów.

Podział na obszary

Okazuje się, że dobrym krokiem ku skonstruowaniu efektywniejszego rozwiązania jest dokładniejsze przyjrzenie się kształtom zdefiniowanych powyżej obszarów A, B i R. Jak łatwo się domyślić, są one istotnie zależne od wzajemnego położenia posterunków $S_j = (x_1, y_1)$ i $T_j = (x_2, y_2)$. W sumie można wyliczyć osiem różnych przypadków, zilustrowanych poniżej. Na rysunkach czarne kropki oznaczają położenia posterunków, natomiast linie oraz zacieniowane prostokąty reprezentują obszar remisowy R i odgraniczają od siebie obszary typu A oraz B.



Przypadek 1. $x_1 = x_2$ lub $y_1 = y_2$.



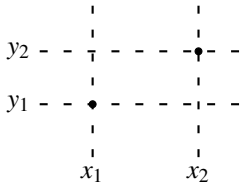
Przypadek 2. $|x_1 - x_2| = |y_1 - y_2|$.

Przypadek 3. Pozostałe konfiguracje.

Wnikliwy Czytelnik zapewne zdążył już sobie postawić pytanie, skąd taki akurat podział na przypadki i w jaki sposób zostały skonstruowane poszczególne diagramy[1]. Istnieją różne możliwe odpowiedzi na to pytanie. Jedna z metod polega na wybieraniu „do skutku" różnych wzajemnych położeń par posterunków i rysowaniu linii podziału na obszary, na przykład poprzez przyglądanie się punktom o współrzędnych całkowitoliczbowych. W ten sposób jednak łatwo o przeoczenie — na przykład w przypadku $|x_1 - x_2| = |y_1 - y_2|$ można nie zauważyć dużych obszarów remisowych.

[1] Rozważane przez nas diagramy stanowią szczególny przypadek tak zwanych *diagramów Voronoi*. W ogólnej wersji dana jest pewna liczba punktów, a naszym zadaniem jest podzielenie płaszczyzny na obszary najbliższe poszczególnym z tych punktów, zazwyczaj w standardowej metryce euklidesowej. Więcej o diagramach Voronoi można poczytać np. w książkach [36] i [39].

Istnieje także bardziej metodyczne podejście do tego problemu. Najłatwiej zidentyfikować i rozpatrzyć przypadki typu 1. We wszystkich pozostałych mamy $x_1 \neq x_2$ i $y_1 \neq y_2$, co pozwala wydzielić dwie możliwe sytuacje: $x_1 < x_2$ i $y_1 < y_2$ oraz $x_1 < x_2$ i $y_1 > y_2$ (pozostałe możliwości są takie same z dokładnością do zamiany posterunków miejscami). Dla każdej z nich możemy podzielić płaszczyznę na 9 obszarów, odpowiadających produktom kartezjańskim przedziałów $[-\infty, x_1]$, $[x_1, x_2]$ i $[x_2, \infty]$ oraz $[-\infty, y_1]$, $[y_1, y_2]$ i $[y_2, \infty]$, co widać na rys. 1.



Rys. 1: Podział płaszczyzny na 9 obszarów w przypadku $x_1 < x_2$, $y_1 < y_2$.

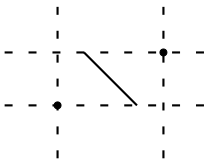
Jeżeli teraz punkt $P = (x, y)$ należy do ustalonego spośród tych obszarów, to każdą z nierówności: $d(P, S_j) < d(P, T_j)$, $d(P, S_j) > d(P, T_j)$ oraz równanie $d(P, S_j) = d(P, T_j)$ można już łatwo rozwiązać. Dla przykładu, jeżeli P należy do środkowego obszaru $[x_1, x_2] \times [y_1, y_2]$, to pierwsza ze wspomnianych nierówności ma postać

$$x - x_1 + y - y_1 < x_2 - x + y_2 - y,$$

czyli

$$x + y < \frac{x_1 + y_1 + x_2 + y_2}{2}, \tag{1}$$

co odpowiada środkowej ukośnej kresce na każdym z diagramów w przypadkach 2 i 3:



Rys. 2: Pierwsza kreska diagramu.

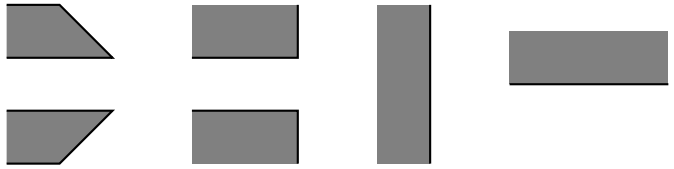
Kontynuując to postępowanie, można wypisać wspomniane nierówności dla poszczególnych obszarów — czasem okazują się one trywialne, czyli zawsze prawdziwe lub zawsze fałszywe (jak na przykład w przypadku lewego dolnego i prawego górnego obszaru na rys. 2), a czasem definiują pewne zależności, które po jednej stronie mają stałą, a po drugiej x, y, $x + y$ lub $x - y$. Dokładniejsza analiza tych nierówności pozwala wykryć, że ich postać zależy już tylko od tego, jak wyrażenie $|x_1 - x_2|$ ma się do wyrażenia $|y_1 - y_2|$, co prowadzi do takiej klasyfikacji przypadków 2 i 3, jak zaprezentowana powyżej.

Podział na podobszary

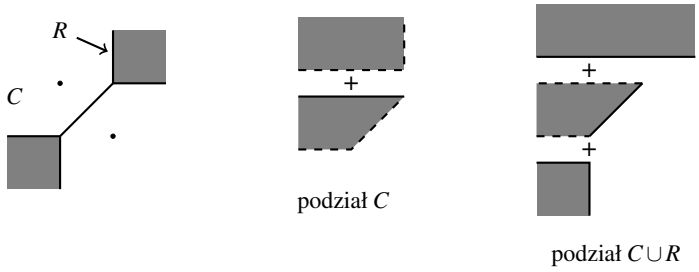
Jako że w zadaniu musimy wyznaczać liczby zabytków P_i, które należą do obszarów A, B, R, to powinno nam zależeć na tym, żeby kształty rozważanych obszarów były możliwie proste i jednolite. Ponieważ diagramy przypadków 1–3 mają stosunkowo urozmaicony kształt, podzielimy poszczególne obszary na mniejsze, regularniejsze części. Im sprytniej to zrobimy, tym łatwiejsza będzie dalsza część rozwiązania.

Zacznijmy od tego, że nigdy nie musimy zliczać zabytków we wszystkich trzech obszarach A, B, R, gdyż zawsze liczby te sumują się do z. W szczególności, możemy pominąć jeden z obszarów A, B. Dla ujednolicenia, do zliczania wybierzemy zawsze ten obszar, który dotyczy górnego lewego spośród posterunków (tzn. lewego, a w razie remisu górnego) — oznaczmy ten obszar przez C. Dodatkowo, zamiast zliczać zabytki w ramach R, wykonamy to dla obszaru $C \cup R$, czyli dla sumy zbiorów C i R. Jest to korzystne, gdyż obszar R ma zazwyczaj kształt zupełnie nieprzypominający C, natomiast kształty C i $C \cup R$ są stosunkowo zbliżone.

Jak teraz łatwo sprawdzić, każdy z obszarów C, $C \cup R$ w każdym z naszych ośmiu przypadków można rozłożyć na stałą liczbę fragmentów sześciu rodzajów przedstawionych na rys. 3. Przykład takiego rozkładu dla lewego podprzypadku przypadku 2 obrazuje rys. 4.



Rys. 3: Podstawowe klocki budulcowe obszarów C i $C \cup R$, od lewej: jednostronnie nieskończone trapezy, dwustronnie nieskończone prostokąty i półpłaszczyzny: pionowa i pozioma. Czarne kreski na brzegach reprezentują fragmenty brzegu, które mogą należeć do figury bądź nie (na każdym prostym fragmencie obwodu charakterystyka przynależności wszystkich punktów będzie jednak zawsze taka sama).

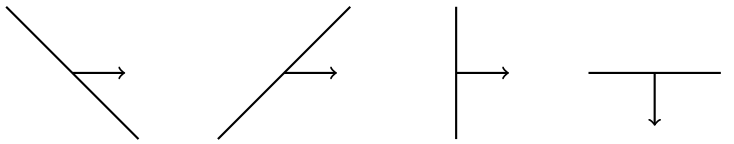


Rys. 4: Rozkład obszarów C oraz $C \cup R$ w drugim przypadku na klocki budulcowe z rys. 3. Linie ciągłe obrazują brzegi zaliczające się do figury, linie przerywane — brzegi otwarte, wreszcie brak linii reprezentuje nieskończony wymiar w danym kierunku.

Zamiatanie

Kluczem do dokończenia naszego rozwiązania jest odstąpienie od podejścia siłowego i rozważanie wszystkich zapytań naraz. Mamy wówczas układ posterunków P_i oraz pewien zbiór obszarów O_j takich jak na rys. 3, a naszym celem jest wyznaczenie liczby posterunków w każdym z tych obszarów. Jeżeli to zrobimy, to na podstawie dotychczasowych rozważań można będzie odtworzyć wyniki dla poszczególnych zapytań.

Przy tak postawionym problemie do rozwiązania możemy wykorzystać klasyczną technikę *zamiatania*[2], która polega na systematycznym (czyli w jakiejś ustalonej kolejności) przejrzeniu płaszczyzny i kolejnym obsługiwaniu napotykanych zdarzeń. W naszym przypadku zdarzeniami są: dodanie punktu P_i oraz zapytanie o liczbę punktów w obszarze O_j. Aby ustalić, jaki konkretnie rodzaj zamiatania jest nam potrzebny, zauważmy, że każdy z obszarów O_j może zostać opisany za pomocą co najwyżej dwóch prostych nierówności na x, y, $x+y$ lub $x-y$, z czego dopuszczamy, aby co najwyżej jedna była podwójna, czyli narzucająca zarówno ograniczenie górne, jak i dolne. To może stanowić wskazówkę, że będzie nam potrzebne nie jedno, ale cztery osobne zamiatania, w których zdarzenia będą uporządkowane odpowiednio po x, y, $x+y$ i $x-y$ (patrz rys. 5).



Rys. 5: Kierunki zamiatania potrzebne do obsłużenia obszarów z rys. 3.

Obszary będące nieskończonymi trapezami będziemy rozważać w zamiataniach, podczas których miotła przemieszcza się w kolejności rosnącej względem odpowiednio $x+y$ (górny obszar na rys. 3) i $x-y$ (dolny). Obsługa zapytania odpowiadającego danemu obszarowi nastąpi w momencie napotkania ukośnego fragmentu brzegu obszaru i polegać będzie na zliczeniu już napotkanych punktów P_i, których współrzędna y należy do przedziału zadanego przez dolny i górny brzeg obszaru. Aby efektywnie odpowiadać na takie zapytania, z miotłą związemy strukturę danych zwaną *drzewem przedziałowym*[3]. Umożliwia ono wykonywanie operacji dodania punktu o danej rzędnej i zapytania o liczbę dodanych punktów o rzędnej z zadanego przedziału w złożoności czasowej $O(\log M)$, gdzie M jest rozważanym zakresem rzędnych. Analogicznie można rozpatrzyć obszary będące nieskończonymi prostokątami i półpłaszczyzny pionowe (tym razem zamiatając w kolejności wzrastających odciętych) oraz półpłaszczyzny poziome (zamiatanie w kolejności malejących rzędnych, czyli „w dół"). Zauważmy, że w drugim z tych przypadków miotła zawierająca same rzędne nie bardzo ma

[2] O zamiataniu lub, jak kto woli, wymiataniu można poczytać we wspomnianych już książkach o geometrii obliczeniowej [36] oraz [39], a także posłuchać na stronie `http://was.zaa.mimuw.edu.pl`. Technika ta była wykorzystywana w rozwiązaniach wielu zadań olimpijskich, jak np. Trójkąty z XV Olimpiady Informatycznej [15] lub Gazociągi z XIV Olimpiady Informatycznej [14].

[3] Więcej o drzewach przedziałowych można przeczytać np. w opisie rozwiązania zadania Tetris 3D z książeczki XIII Olimpiady Informatycznej [13], albo poczytać i posłuchać na stronie `http://was.zaa.mimuw.edu.pl`

sens, jednakże wówczas i tak wszystkie zapytania będą dotyczyły zliczania wszystkich już napotkanych punktów, więc jest to bez znaczenia.

Do dokończenia rozwiązania pozostało nam jeszcze kilka istotnych szczegółów. Przede wszystkim nie wspomnieliśmy jeszcze, że drzewo przedziałowe odpowiadające zakresowi współrzędnej o rozmiarze M ma złożoność pamięciową $O(M)$. To stanowi w naszym przypadku istotny problem, gdyż górne ograniczenia na wartości współrzędnych w zadaniu (tj. n oraz m) są bardzo duże. W takim przypadku należy jednak zastosować jedną ze standardowych technik (także opisanych na stronie `http://was.zaa.mimuw.edu.pl`), na przykład przenumerowanie współrzędnych. Dokładniej, ponieważ przed rozpoczęciem zamiatania z góry możemy przewidzieć wszystkie wartości rzędnych, które wystąpią przy wstawianiu oraz w zapytaniach, więc wystarczy prenumerować te faktycznie dla nas istotne wartości kolejnymi liczbami naturalnymi $0, 1, 2, \ldots$ Samo prenumerowanie wymaga zaledwie jednego sortowania — koszt czasowy $O\big((p+z)\log(p+z)\big)$ — natomiast po jego wykonaniu pozostaje nam już tylko $M = O(p+z)$ istotnych rzędnych — korzystamy w tym miejscu z tego, że każdy z obszarów C, $C \cup R$ rozkłada się na stałą liczbę klocków budulcowych z rys. 3. Innym rozwiązaniem może być użycie dynamicznej (opartej o wskaźniki) wersji drzewa przedziałowego (również opisanej na wspomnianej stronie).

Kolejna trudność wiąże się z otwartością i domkniętością poszczególnych brzegów figur z rys. 3. Jak dotychczas kompletnie to ignorowaliśmy, ale już przykład z rys. 4 pokazuje, że z przynależnością brzegów do obszarów budulcowych może być w rozkładach bardzo różnie. Właściwą metodę radzenia sobie z tym problemem opiszemy znów na przykładzie nieskończonych trapezów. Otwartość bądź domkniętość górnych i dolnych brzegów łatwo obsłużyć, po prostu zadając zapytania do drzewa przedziałowego zawierające przedziały otwarte tudzież domknięte z odpowiednich stron. Natomiast ukośny fragment brzegu musi być rozpatrywany w zupełnie innym miejscu algorytmu, a mianowicie podczas sortowania zdarzeń względem $x+y$ (odpowiednio $x-y$). W tym celu należy podczas sortowania odpowiednio rozstrzygać remisy między punktami P_i a obszarami O_j: w przypadku takiego remisu pierwszeństwo mają obszary o otwartym ukośnym fragmencie brzegu (w dowolnej już kolejności między sobą), dalej następują wszystkie posterunki P_i, a na końcu zostają umieszczone obszary o brzegu domkniętym.

Podsumowanie

Rozwiązanie wzorcowe składa się z następujących kroków:

1. Podział wszystkich zapytań na przypadki 1–3 i dalsze podprzypadki.
2. Podział każdego z fragmentów C oraz $C \cup R$ na obszary budulcowe.
3. Przenumerowanie rzędnych punktów P_i oraz wyznaczających brzegi obszarów O_j za pomocą sortowania i scalenia posortowanych list (opcjonalnie: wyszukiwania binarnego zamiast scalania).
4. Cztery zamiatania:
   (a) sortowanie punktów i odpowiednich obszarów względem, odpowiednio, x, y, $x+y$ bądź $x-y$ (tutaj y reprezentuje współrzędną nieprzenumerowaną);

(b) obsługa kolejnych zdarzeń w posortowanej kolejności z wykorzystaniem drzewa przedziałowego względem rzędnych.

Pierwsze dwa kroki można zrealizować w złożoności czasowej $O(z)$, natomiast dwa następne w koszcie czasowym $O\big((p+z)\log(p+z)\big)$, zdeterminowanym przez sortowania bądź sekwencje operacji na drzewach przedziałowych. Złożoność pamięciowa to $O(p+z)$. Gdybyśmy użyli dynamicznego drzewa przedziałowego zamiast przenumerowania rzędnych, złożoność pamięciowa wyniosłaby $O((p+z)\log M)$, co przy odpowiedniej implementacji również pozwala zmieścić się w limicie 64 MB.

Właściwie wszystkie rozwiązania jurorskie poza tym opisanym we wstępie opierają się na powyższym schemacie. Poprawne implementacje można znaleźć w plikach: `str.cpp`, `str1.pas`, `str2.java` i `str3.cpp`. W pliku `strb1.cpp` znajduje się rozwiązanie błędne, które w niewłaściwy sposób generowało diagramy w przypadku 2 (wspominaliśmy już o tym błędzie) — uzyskiwało ono 30% punktów. Z kolei plik `strb2.cpp` zawiera implementację rozwiązania, które uzyskiwało 70% punktów wskutek błędów przepełnienia typu całkowitego 32-bitowego — zauważmy, że współrzędne w zadaniu ograniczone są z góry przez 10^9, a obliczenie ograniczeń obszarów może wymagać np. czterech dodawań takich liczb, patrz nierówność (1). Dodajmy także, że mniej umiejętny podział obszarów A, B, R na podobszary mógłby spowodować konieczność wykonania większej liczby zamiatań (sześciu, ośmiu...) — implementacje takich rozwiązań jurorzy pozostawili jednak zawodnikom.

Testy

Rozwiązania zawodników były sprawdzane na następujących dziesięciu testach:

Nazwa	z	p	max(n, m)	Opis
str1.in	3	100	10	mały test losowy bez przypadku 2
str2.in	101	513	10^6	mały test losowy bez przypadku 2
str3.in	9 000	83 810	100	pełna kratownica zabytków 100×90
str4.in	5 213	90 213	$\approx 10^7$	średni test losowy
str5.in	28 000	30 000	$\approx 10^6$	średni test losowy
str6.in	63 453	52 342	$\approx 2 \cdot 10^7$	średni test losowy
str7.in	70 013	57 000	$\approx 10^7$	średni test losowy
str8.in	80 000	80 000	$\approx 10^8$	duży test losowy
str9.in	99 876	95 123	$\approx 10^9$	duży test losowy bez przypadku 2
str10.in	100 000	100 000	10^9	maksymalny test losowy

Zawody II stopnia

opracowania zadań

Andrzej Walat	Redakcja	Jakub Łącki
Treść zadania	Opracowanie	Program

Dostępna pamięć: 64 MB. *OI, Etap II, dzień próbny, 10.02.2009*

Wyspy na trójkątnej sieci

Sieć trójkątów jest zbudowana z równobocznych trójkątów o boku 1 (ilustracja na końcu treści zadania). **Ścieżką** *na sieci trójkątów nazywamy dowolny skończony ciąg trójkątów (o boku 1) sieci, taki, że każde kolejne dwa trójkąty w tym ciągu mają wspólny bok.*

Figurę utworzoną przez punkty skończonej liczby trójkątów sieci nazywamy **wyspą**, *jeśli dowolne dwa zawarte w niej trójkąty sieci można połączyć ścieżką utworzoną z trójkątów zawartych w tej figurze.*

Figury przedstawione na rysunkach 1.1, 1.2 i 1.3 są wyspami. Figura na rysunku 1.4 nie jest wyspą. Figury na rysunkach 2.2, 2.3 i 2.5 są przystające.

Zamierzamy dla każdego $n \leqslant 10$ opisać w systematyczny sposób wszystkie nieprzystające wyspy, jakie można utworzyć z n trójkątów o boku 1, i policzyć, ile ich jest.

Brzeg każdej wyspy, zbudowanej z co najwyżej dziesięciu trójkątów, jest łamaną zamkniętą złożoną z jednostkowych odcinków siatki i można go obiec (obrysować bez odrywania ołówka od papieru) w ten sposób, że dokładnie raz przebiegamy każdy odcinek brzegu i wracamy do punktu wyjścia. Może się zdarzyć, że trzeba będzie przy tym przejechać więcej niż raz przez ten sam punkt (patrz rysunek 2.4).

W przypadku wysp zbudowanych z co najwyżej dziesięciu trójkątów nie jest możliwa sytuacja taka, jak na rysunku 1.2, że brzeg figury składa się z dwóch rozłącznych części i nie można go obiec (obrysować bez odrywania ołówka od papieru).

Obiegając brzeg, po każdym jednostkowym odcinku wykonujemy skręt jednego z następujących typów:

- `a` — *o 120 stopni w lewo,*
- `b` — *o 60 stopni w lewo,*
- `c` — *o 0 stopni (tzn. brak skrętu),*
- `d` — *o 60 stopni w prawo,*
- `e` — *o 120 stopni w prawo.*

Każdy obieg brzegu wyspy można opisać za pomocą słowa złożonego z liter ze zbioru {`a`,`b`,`c`,`d`,`e`}, *odnotowując za pomocą odpowiedniej litery skręt, jaki należy wykonać po każdym kolejnym, jednostkowym odcinku łamanej tworzącej brzeg. Opis obiegu brzegu wyspy ma tyle liter, z ilu jednostkowych odcinków składa się łamana tworząca ten brzeg. Odnotowujemy skręt także po ostatnim odcinku łamanej, chociaż nie jest to konieczne dla jednoznacznego określenia jej kształtu. Ta, w pewnym sensie nadmiarowa, litera ułatwia przekształcenie danego opisu w opis innego obiegu tej samej figury, który zaczyna się w innym punkcie.*

Słowa `cdddcddd`, `dcdddcdd`, `cbbbcbbb` *opisują różne obiegi figury na rysunku 2.1.*

Słowa `cbeddcde`, `adcabcbb`, `abcbbadc` *opisują różne obiegi figury na rysunku 2.2.*

Słowa `acdabbcb` *i* `cddebced` *opisują różne obiegi figury na rysunku 2.3.*

Jeśli obiegając brzeg wyspy mamy stale jej wnętrze po prawej stronie, to mówimy, że jest to **obieg prawoskrętny**.

92 *Wyspy na trójkątnej sieci*

Dla każdej wyspy można wyznaczyć zbiór wszystkich wysp do niej przystających oraz ich obiegi prawoskrętne. **Kodem wyspy** *nazwiemy słowo, które:*

1. *jest opisem pewnego prawoskrętnego obiegu brzegu jakiejś wyspy do niej przystającej,*
2. *jest wcześniejsze w porządku alfabetycznym od wszystkich innych słów spełniających warunek 1.*

Dla wysp przedstawionych na rysunkach 2.2 i 2.3, które są przystające, bierzemy pod uwagę wszystkie prawoskrętne obiegi obydwu:

`beddcdec`, `eddcdecb`, `ddcdecbe`, `dcdecbed`, `cdecbedd`, `decbeddc`, `ecbeddcd`, `cbeddcde`

oraz

`bcedcdde`, `cedcddeb`, `edcddebc`, `dcddebce`, `cddebced`, `ddebcedc`, `debcedcd`, `ebcedcdd`

i jako kod każdej z nich bierzemy to słowo, które w porządku alfabetycznym należy umieścić na pierwszym miejscu: `bcedcdde`.

Kodem wyspy przedstawionej na rysunku 2.4 jest słowo: `aadecddcddde`.

Napisz program, który:

- *dla podanego kodu wyspy o rozmiarze k wygeneruje kody wszystkich wysp o rozmiarze $k+1$, które można utworzyć przez dodanie do niej jednego trójkąta,*
- *dla podanej liczby całkowitej n wygeneruje kody wszystkich wysp rozmiaru n.*

Wejście

W pierwszym wierszu standardowego wejścia podana jest liczba całkowita t ($1 \leqslant t \leqslant 5$), oznaczająca liczbę zapytań. Każdy z kolejnych t wierszy zawiera opis zapytania pewnego typu. Zapytanie pierwszego typu składa się z litery `K` *oraz kodu wyspy składającej się z co najwyżej dziewięciu trójkątów, oddzielonych pojedynczym odstępem. Zapytanie drugiego typu składa się z litery* `N` *oraz liczby całkowitej n ($1 \leqslant n \leqslant 10$), oddzielonych pojedynczym odstępem.*

Wyjście

Na standardowe wyjście wypisz odpowiedzi na poszczególne zapytania.

Dla zapytań pierwszego typu należy najpierw wypisać liczbę różnych kodów wysp, które można utworzyć z przystających wysp opisanych podanym kodem poprzez dodanie jednego trójkąta. W następnym wierszu należy wypisać wszystkie te kody w kolejności alfabetycznej, pooddzielane pojedynczymi odstępami.

Dla zapytań drugiego typu należy wypisać liczbę różnych kodów wysp utworzonych z n trójkątów. W kolejnym wierszu należy wypisać wszystkie te kody w kolejności alfabetycznej.

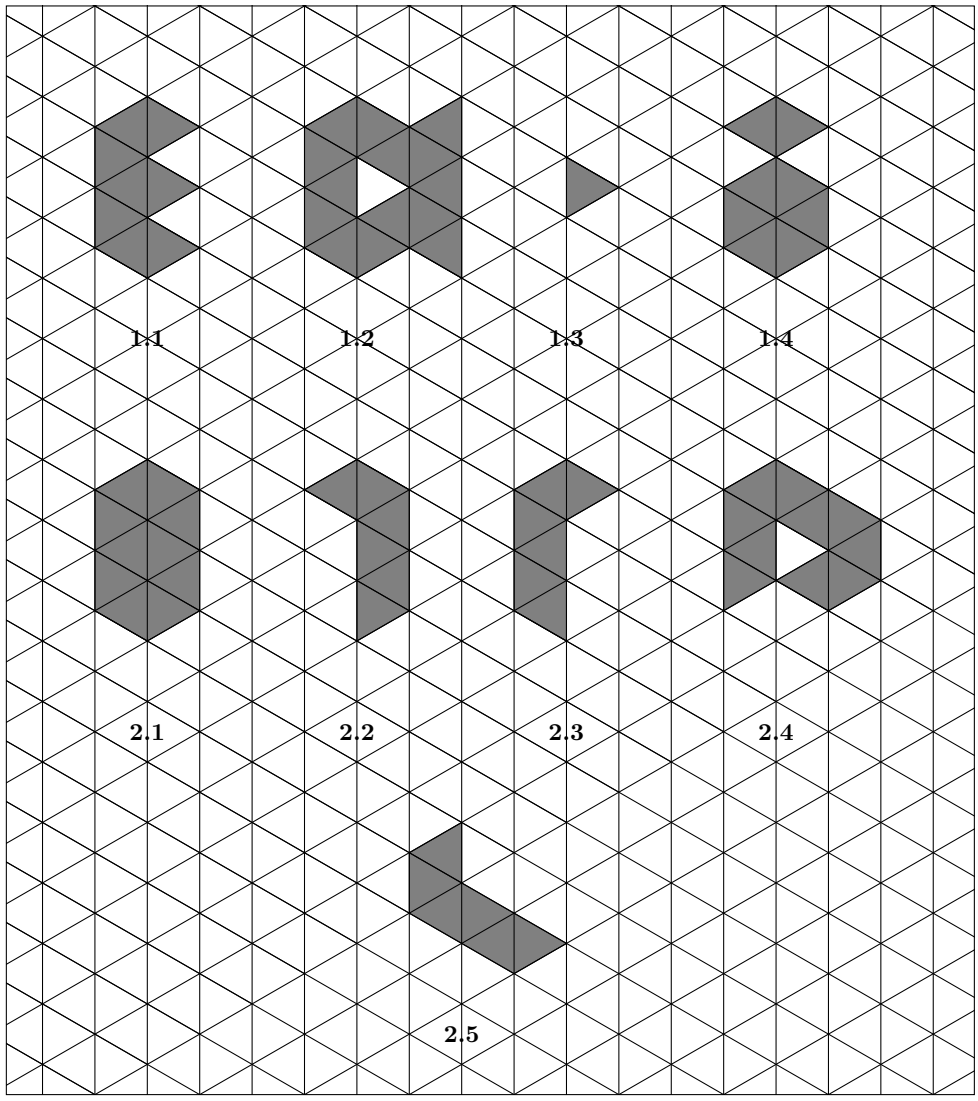
Przykład

Dla danych wejściowych:

```
2
K adeccecced
N 5
```

poprawnym wynikiem jest:

```
5
acedccecced addebcecced adebebecced adecbedcced cceccecce
4
aedddde bdecdde bececde ccedcde
```



Rozwiązanie

Zadanie o *Wyspach na trójkątnej sieci* zajmuje wyjątkowe miejsce wśród zadań olimpijskich, gdyż pojawiło się na zawodach Olimpiady Informatycznej już dwukrotnie: na II etapie pierwszej OI i teraz, na II etapie XVI OI. Celem ponownego wykorzystania tego zadania było m.in. porównanie, jak zmienił się charakter i poziom Olimpiady w ciągu tych piętnastu lat. Jakąś wskazówką w tym zakresie mogą być statystyki tego zadania z I Olimpiady — wówczas spośród 64 uczestników II etapu, troje uzyskało maksymalną liczbę 100 punktów, dziewięcioro miało ponad 90 punktów, a 0 punktów otrzymało 16 uczestników. Trudno jednak wysnuć jakieś konkretne wnioski z zestawienia tego typu statystyk, gdyż podczas owej olimpiady sposób sprawdzania był inny niż obecnie (m.in. przydzielano punkty uznaniowe, tudzież sprawdzano zachowanie programów na testach niezgodnych z opisem wejścia), no a poza tym zadanie o Wyspach nie było wtedy zadaniem próbnym. Dlatego dalsze dywagacje na ten temat pozostawiamy Czytelnikom.

Jako że opis rozwiązania tego zadania można znaleźć w książeczce I Olimpiady [1], nie publikujemy go w niniejszej książeczce. Ponieważ jednak publikacja [1] jest, niestety, w obecnych czasach trudno dostępna, więc umieszczamy dwie wskazówki dotyczące rozwiązania:

- Łączna liczba wszystkich wysp złożonych z co najwyżej 10 trójkątów jest stosunkowo mała (np. dla $n = 10$ mamy 448 takich wysp). Z tego względu odpowiadanie na zapytania drugiego typu można zrealizować za pomocą zapytań pierwszego typu.

- Dokładany trójkąt może mieć z wyspą 1, 2 lub 3 boki wspólne. Ostatnia z tych sytuacji ma jednak miejsce tylko dla jednego typu wyspy (2.4 na rysunku w treści zadania).

Poza powyższymi, trudność zadania jest czysto implementacyjna — w oryginalnym rozwiązaniu z I OI czyniono pewne sprytne spostrzeżenia dotyczące własności wysp, lecz można sobie poradzić i bez nich, np. konwertując wyspy z postaci tekstowej na wielokątową i nachalnie dodając jeden trójkąt wszędzie, gdzie się da. Po szczegóły odsyłamy Czytelników do programów wzorcowych, które reprezentują rozmaite podejścia do rozwiązania: `wys.java`, `wys1.pas`, `wys2.cpp` i `wys3.cpp`.

Dodajmy na koniec, że zestaw testów użytych na tegorocznych zawodach był inny niż oryginalny. Ułożono 10 testów, z czego testy 2, 4, 6, 7 i 8 zawierają tylko zapytania pierwszego typu, testy 1, 3 i 5 — drugiego typu, a testy 9 i 10 — obydwu typów.

Marcin Kubica
Treść zadania, Opracowanie

Paweł Parys
Program

Dostępna pamięć: 32 MB. *OI, Etap II, dzień pierwszy, 11.02.2009*

Konduktor

Bajtazar pracuje jako konduktor w ekspresie Bajtockich Kolei Państwowych (BKP) relacji Bajtogród-Bitowice. Rozpoczął się trzeci etap reformy BKP[1]*, w ramach którego zmieniany jest system wynagrodzeń tak, aby lepiej motywował pracowników do efektywnej pracy. Wynagrodzenie Bajtazara będzie teraz zależało od liczby sprawdzonych przez niego biletów (pasażerów). Bajtazar jest w stanie sprawdzić bilety wszystkich pasażerów w trakcie przejazdu między dwiema kolejnymi stacjami. Nie ma jednak ani siły, ani chęci, żeby sprawdzać bilety między każdymi dwiema kolejnymi stacjami. Po namyśle zdecydował się, że będzie sprawdzał bilety k razy na trasie pociągu.*

Przed wyruszeniem w trasę Bajtazar otrzymuje zbiorcze zestawienie, ilu pasażerów będzie jechało z danej stacji do danej innej stacji. Na podstawie tych danych Bajtazar chciałby tak dobrać momenty kontroli biletów, aby sprawdzić jak najwięcej biletów. Niestety, powtórne sprawdzenie biletu pasażera, który już raz został skontrolowany, nie wpływa na wynagrodzenie Bajtazara. Napisz program, który pomoże ustalić Bajtazarowi, kiedy powinien kontrolować bilety, aby jego zarobki były jak największe.

Wejście

W pierwszym wierszu standardowego wejścia zapisane są dwie dodatnie liczby całkowite n i k ($1 \leqslant k < n \leqslant 600$, $k \leqslant 50$), oddzielone pojedynczym odstępem i oznaczające liczbę stacji na trasie pociągu oraz liczbę kontroli biletów, które chce przeprowadzić Bajtazar. Stacje są ponumerowane kolejno od 1 do n.

W kolejnych $n-1$ wierszach jest zapisane zbiorcze zestawienie pasażerów. Wiersz $(i+1)$-szy zawiera informacje o pasażerach, którzy wsiądą na stacji i — jest to ciąg $n-i$ nieujemnych liczb całkowitych pooddzielanych pojedynczymi odstępami: $x_{i,i+1}, x_{i,i+2}, \ldots, x_{i,n}$. Liczba $x_{i,j}$ (dla $1 \leqslant i < j \leqslant n$) oznacza liczbę pasażerów, którzy wsiedli na stacji i i jadą do stacji j. Łączna liczba pasażerów (czyli suma wszystkich $x_{i,j}$) wynosi nie więcej niż $2\,000\,000\,000$.

Wyjście

Twój program powinien wypisać na standardowe wyjście (w jednym wierszu) rosnący ciąg k liczb całkowitych z przedziału od 1 do $n-1$, pooddzielanych pojedynczymi odstępami. Liczby te mają być numerami stacji, po ruszeniu z których Bajtazar powinien sprawdzić bilety pasażerów.

[1] Historia reform BKP i stacji Bitowice została opisana w zadaniach **Koleje** z III etapu XIV OI oraz **Stacja** z III etapu XV OI. Znajomość tych zadań nie jest jednak w najmniejszym stopniu potrzebna do rozwiązania niniejszego zadania.

Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
2 5
```

lub

```
3 5
```

W obu przypadkach Bajtazar skontroluje 42 bilety.

Rozwiązanie

Analiza problemu

Główną część danych wejściowych stanowi informacja o liczbie pasażerów, którzy wsiadają na danej stacji, a wysiadają na innej danej stacji. Na rys. 1.a przedstawiono te dane (z przykładu z treści zadania) w postaci tabeli — wiersz określa numer stacji, na której pasażerowie wsiadają, a kolumna określa numer stacji, na której wysiadają. Zastanówmy się, jak z takiej tabeli odczytać, ile biletów sprawdzi Bajtazar w trakcie jednej kontroli. Jeżeli narysujemy prostokąt o dolnym lewym rogu na przekątnej (w miejscu odpowiadającym momentowi kontroli), a górnym prawym rogu w górnym prawym rogu tabeli, to będzie to suma liczb wewnątrz takiego prostokąta. Na rys. 1.a przedstawiono prostokąt odpowiadający kontroli biletów między stacjami nr 2 i 3.

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

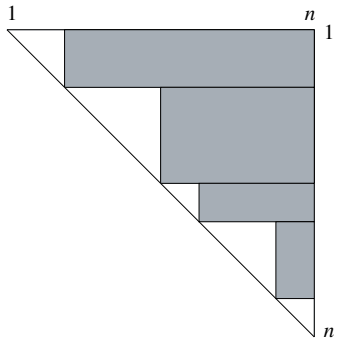
a)

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

b)

Rys. 1: Przedstawienie danych o pasażerach w postaci tabeli. Prostokąt oznacza pasażerów, których bilety zostaną skontrolowane po tym, jak pociąg ruszy ze stacji nr 2 (a) oraz 2 i 5 (b).

Możemy więc sformułować nasze zadanie w następujący sposób: znaleźć takie k prostokątów (o dolnym lewym rogu na przekątnej i górnym prawym rogu w górnym prawym rogu tabeli), że pokrywają one liczby o maksymalnej sumie.

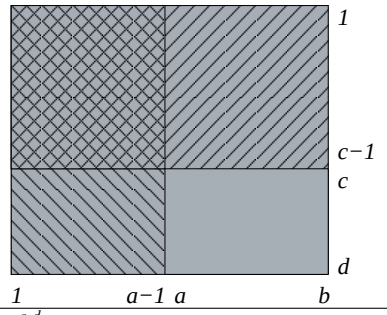


Rys. 2: Prostokąty obejmujące wszystkie sprawdzane bilety.

Jedno z rozwiązań przykładu z treści zadania, w którym Bajtazar sprawdza bilety między stacjami 2 i 3 oraz 5 i 6, jest przedstawione na rys. 1.b. Zwróćmy jednak uwagę, że liczby znajdujące się w obu prostokątach reprezentujących te dwie kontrole biletów powinny być liczone tylko raz. Byłoby najwygodniej, gdybyśmy rozważając konkretne momenty k kontroli biletów, sumowali liczby w prostokątach, które nie zachodzą na siebie. Możemy to zrobić, na przykład, w taki sposób, żeby prostokąty odpowiadające późniejszym kontrolom biletów nie zachodziły na prostokąty odpowiadające wcześniejszym kontrolom, jak pokazano na rys. 2.

Szybkie wyznaczanie sumy liczb w prostokącie

Widać już, że rozwiązując nasze zadanie, będziemy często obliczać sumy liczb zawartych w różnych prostokątach. Jest to dobry moment, żeby przypomnieć standardową technikę pozwalającą wyznaczać takie sumy w czasie stałym, po uprzednim wstępnym przetworzeniu tabeli z danymi o pasażerach (w czasie $O(n^2)$). Technika ta stanowiła sedno zadania *Mapa gęstości* z VIII OI [8] i pojawiała się już w innych zadaniach olimpijskich, np. w *Kupnie gruntu* z XV OI [15].



Rys. 3: Aby obliczyć $K_{a,b}^{c,d}$ (szary niezakreskowany prostokąt), musimy od $S[b,d]$ (cały prostokąt) odjąć $S[a-1,d]$ oraz $S[b,c-1]$ (obszary zakreskowane ukośnie), a do tego wszystkiego dodać, uprzednio odjęte dwukrotnie, $S[a-1,c-1]$ (podwójnie zakreskowany prostokąt).

Przypomnijmy, że $x_{i,j}$ dla $1 \leqslant i < j \leqslant n$ oznacza liczbę pasażerów podróżujących od stacji i do j, a dla $1 \leqslant j \leqslant i \leqslant n$ przyjmijmy $x_{i,j} = 0$. Oznaczmy przez $K_{a,b}^{c,d}$ sumę liczb $x_{i,j}$ zawartych w prostokącie, którego górny lewy róg ma współrzędne (a,c), a dolny prawy (b,d):

$$K_{a,b}^{c,d} = \sum_{i=a}^{b} \sum_{j=c}^{d} x_{i,j}.$$

Przyjmijmy oznaczenie $S[i,j] = K_{1,i}^{1,j}$ i załóżmy, że znamy wartości $S[i,j]$ dla wszystkich $0 \leqslant i,j \leqslant n$ (zakładamy, że $S[i,0] = S[0,j] = 0$). Chcąc wyznaczyć wartość $K_{a,b}^{c,d}$, korzystamy ze wzoru:

$$K_{a,b}^{c,d} = S[b,d] - S[a-1,d] - S[b,c-1] + S[a-1,c-1].$$

Ideę tej techniki przedstawia rysunek 3. Aby wyznaczyć wartości $S[i,j]$, wystarczy przejrzeć kolejne wiersze tabeli od góry do dołu (a w obrębie wiersza poruszać się od lewej do prawej) i skorzystać z tożsamości:

$$S[i,j] = x_{i,j} + S[i-1,j] + S[i,j-1] - S[i-1,j-1].$$

Rozwiązanie wzorcowe

Wiemy już, jak efektywnie wyznaczyć liczbę sprawdzanych biletów dla konkretnych k momentów kontroli biletów. Pozostaje problem, w jaki sposób rozpatrzyć wszystkie takie konfiguracje kontroli i wybrać z nich tę najkorzystniejszą. Możemy tu zastosować programowanie dynamiczne.

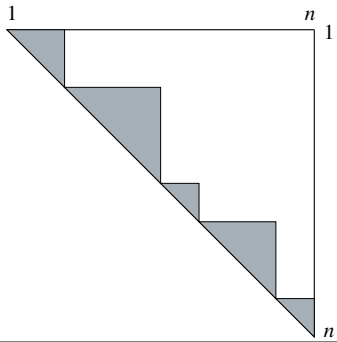
Tak jak w przypadku każdego zastosowania programowania dynamicznego, zanim rozwiążemy jeden konkretny problem, musimy go sparametryzować i rozwiązać wiele podobnych problemów. W tym przypadku, dla każdego $1 \leqslant l \leqslant n$ oraz $0 \leqslant h \leqslant k$ wyznaczymy maksymalną liczbę biletów należących do pasażerów, którzy wsiedli na stacji l lub później, które można sprawdzić, wykonując h kontroli. Oznaczmy tę liczbę biletów przez $T[l,h]$. Oczywiście $T[1,k]$ jest maksymalną liczbą biletów, jakie mogą zostać sprawdzone.

W przypadkach brzegowych, dla $h = 0$ (tzn. gdy bilety nie są kontrolowane) lub $n - l < h$ (tzn. gdy nie da się wykonać wszystkich kontroli), przyjmujemy $T[l,h] = 0$. W pozostałych przypadkach możemy skorzystać z następującej zależności rekurencyjnej:

$$T[l,h] = \max_{i=l,\ldots,n-h} \left(T[i+1,h-1] + K_{l,i}^{i+1,n} \right). \tag{1}$$

W powyższym wzorze i oznacza numer stacji, po której następuje pierwsza kontrola (wśród stacji o numerach nie mniejszych niż l). Obliczając $T[l,h]$, korzystamy wyłącznie z wartości $T[i,j]$ dla $i > l$ oraz $j < h$. Wystarczy więc obliczać wartości $T[l,h]$ w kolejności rosnących h i malejących l. Przypomnijmy, że wartości $K_{l,i}^{i+1,n}$ możemy wyznaczać w czasie stałym. Tak więc obliczenie maksimum w powyższym wzorze zajmuje czas $O(n)$. Mamy $O(n \cdot k)$ wartości $T[l,h]$ do obliczenia, tak więc wyznaczenie wszystkich wartości $T[l,h]$ wymaga czasu rzędu $O(n^2 \cdot k)$. Złożoność pamięciowa jest rzędu $O(n^2)$.

Jednak tym, czego szukamy, nie jest $T[1,k]$, ale numery stacji, po których mają nastąpić kontrole biletów. W tym celu możemy, wyznaczając wartości $T[l,h]$, zapamiętywać, dla



Rys. 4: Zaznaczone obszary trójkątne odpowiadają pasażerom, których bilety nie zostaną sprawdzone. Momenty kontroli są dokładnie takie same jak na rys. 2.

jakiego i osiągnęliśmy maksimum. Pozwoli nam to, po obliczeniu wszystkich $T[l,h]$, odtworzyć numery stacji, po których następują kontrole, w czasie $O(k)$.

Istnieje wiele analogicznych rozwiązań opartych na programowaniu dynamicznym — wszystkie one działają w tej samej złożoności czasowej i pamięciowej co opisane wcześniej rozwiązanie. Przykładowo, można inaczej wykonać pierwszą fazę rozwiązania wzorcowego, używając innego algorytmu wyznaczania wartości $S[i,j]$, albo też stosując mniej ogólne wstępne obliczenia — jako że w rozwiązaniu wzorcowym wykorzystywane są tylko wartości $K_{a,b}^{c,d}$ dla $d = n$, patrz wzór (1).

Można też, zamiast maksymalizować liczbę pasażerów, których bilety zostaną sprawdzone, minimalizować liczbę pasażerów, których bilety nie zostaną sprawdzone. Wówczas wystarczy skupić się na pasażerach, którzy zdążą wsiąść i wysiąść między dwiema kolejnymi kontrolami. Liczba takich pasażerów odpowiada sumie liczb w odpowiednim trójkącie, tak jak to zaznaczono na rys. 4. Przy tym, trójkąty te nie zachodzą na siebie, wystarczy więc sumować wyniki wyznaczone dla poszczególnych trójkątów.

Podobnie jak w poprzednim rozwiązaniu, zaczynamy od wstępnego przetworzenia danych, tak aby móc w czasie stałym wyznaczać sumy elementów w trójkątach (takich, jak na rys. 4). Oznaczmy przez $A[i,j]$ (dla $1 \leqslant i < j \leqslant n$) liczbę pasażerów, którzy wsiedli najwcześniej na stacji i, a wysiedli najpóźniej na stacji j. Łatwo wówczas zauważyć, że $A[i,j] = K_{i,j}^{i,j}$. Przyjmijmy, że $A[i,j] = 0$ dla $i \geqslant j$.

W dalszej części ponownie stosujemy programowanie dynamiczne. Dla każdej liczby kontroli h (dla $0 \leqslant h \leqslant k$) oraz dla każdej stacji l obliczamy minimalną liczbę pasażerów, którzy podróżują nie dalej niż do stacji l, a których bilety nie zostaną sprawdzone, przy założeniu, że na odcinku od stacji 1 do l jest h kontroli. Oznaczmy tę liczbę przez $B[l,h]$. Korzystamy tutaj z następujących wzorów:

$$\begin{aligned} B[l,0] &= A[1,l] \\ B[l,h] &= \min_{i=h,\ldots,l-1} (B[i,h-1] + A[i+1,l]) \text{ dla } l > h. \end{aligned}$$

W powyższym wzorze i oznacza numer stacji, po której następuje ostatnia kontrola (na odcinku do l-tej stacji). Wartości $B[l,h]$ obliczamy dla kolejnych, coraz większych wartości l oraz h (wynikiem końcowym jest $B[n,k]$). Obliczenie jednego minimum zajmuje czas $O(n)$, tak więc obliczenie wszystkich wartości $B[l,h]$ zajmuje czas rzędu $O(n^2 \cdot k)$. Aby móc

odtworzyć numery stacji, po których następują kontrole, możemy, podobnie jak poprzednio, zapamiętywać, dla jakich i otrzymaliśmy minimum. Tak więc, złożoności czasowa i pamięciowa są takie same jak w przypadku pierwszego rozwiązania. Rozwiązanie takie zostało zaimplementowane w pliku `kon1.cpp`, a z pewnymi drobnymi ulepszeniami również w plikach `kon.cpp`, `kon2.pas` i `kon4.java`.

Inne rozwiązania

Jeżeli nie zastosujemy wstępnego przetwarzania czy obliczania sum częściowych, które pozwolą na wyznaczanie wartości, z których bierzemy maksimum (lub minimum), w czasie stałym, to złożoność czasowa ulegnie pogorszeniu. Obliczenie jednej wartości $T[l,h]$ czy $B[l,h]$ będzie wymagać czasu rzędu $O(n^3)$. Takie rozwiązanie zostało zaimplementowane w pliku `kons6.cpp`. Możliwe jest też obliczanie sum częściowych, które przyspieszą rozwiązanie, ale tylko o czynnik n, dając złożoność czasową rzędu $O(n^3 \cdot k)$. Rozwiązania takie zostały zaimplementowane w plikach `kons5.cpp` i `kons7.cpp`.

Jeżeli nie zastosujemy w ogóle techniki programowania dynamicznego i będziemy przeglądać wszystkie możliwe kombinacje kontroli biletów, to uzyskamy rozwiązanie o złożoności wykładniczej. Zostało ono zaimplementowane w pliku `kons8.cpp`.

Testy

Większość testów została wygenerowana losowo. Jedynie trzy testy są inne. W teście 8b mamy $k = n-1$, czyli między każdymi dwiema stacjami muszą być sprawdzane bilety. W teście 9b wszyscy pasażerowie podróżują na dużej odległości, a w teście 10b każdy jedzie tylko do następnej stacji. Poniższa tabelka podsumowuje parametry użytych testów.

Nazwa	n	k	Opis
kon1.in	20	10	test losowy
kon2.in	50	25	test losowy
kon3.in	100	30	test losowy
kon4.in	250	50	test losowy
kon5.in	500	50	test losowy
kon6.in	600	40	test losowy
kon7.in	600	45	test losowy
kon8a.in	600	50	test losowy
kon8b.in	51	50	kontrola biletów między każdymi dwiema stacjami
kon9a.in	600	50	test losowy
kon9b.in	600	50	długie trasy przejazdu pasażerów
kon10a.in	600	50	test losowy
kon10b.in	600	50	przejazdy do następnej stacji

Mirosław Michalski	Mirosław Michalski, Jakub Łącki	Jakub Łącki
Treść zadania	Opracowanie	Program

Dostępna pamięć: 128 MB. *OI, Etap II, dzień pierwszy, 11.02.2009*

Przechadzka Bajtusia

Bajtuś jest jednym z najmłodszych mieszkańców Bajtogrodu. Dopiero niedawno nauczył się pisać i czytać. Jest jednak na tyle duży, że sam już chodzi do szkoły. Codziennie rano wychodzi z domu, a następnie wstępuje po kolei do wszystkich swoich kolegów; dopiero po dołączeniu się ostatniego kolegi cała grupa idzie na lekcje.

Pewnego dnia nauczyciel Bajtusia poprosił go o sporządzenie listy ulic, którymi Bajtuś przechodzi po drodze z domu do szkoły, i odczytanie tej listy na następnych zajęciach. Szybko jednak okazało się, że taka lista mogłaby pochłonąć ogromne ilości papieru. Ustalono więc, że Bajtuś zapisze jedynie pierwszą literę z nazwy każdej ulicy, po której przejdzie. Każda ulica w Bajtogrodzie jest jednokierunkowa i łączy dwa różne skrzyżowania.

Bajtuś podczas odczytywania opisu trasy robi przerwy jedynie w miejscach, w których wstępował do kolegów. Możemy więc traktować opis każdego fragmentu jego drogi z domu do szkoły jako jedno słowo. Chłopiec wciąż ma problemy z prawidłowym czytaniem, tj. czytaniem od lewej strony do prawej, i zamiast tego zdarza mu się czytać od strony prawej do lewej. Czasem przeczyta wyraz **mleko** *jako* **mleko**, *a czasem jako* **okelm**. *Rodzice Bajtusia wiedzą o tych problemach syna, więc postanowili mu pomóc i znaleźć taką trasę, by opis każdego fragmentu, niezależnie od tego, z której strony będzie czytany, brzmiał tak samo. Jednocześnie chcieliby, żeby długość każdego kawałka tej trasy była jak najmniejsza. Zwrócili się do Ciebie z prośbą o pomoc.*

Napisz program, który:

- *wczyta ze standardowego wejścia opis miasta,*
- *wyznaczy taką trasę z domu do szkoły, w której każdy fragment jest możliwie najkrótszy, a jego opis, niezależnie od kierunku czytania, brzmi dokładnie tak samo,*
- *wypisze wyznaczone opisy fragmentów drogi na standardowe wyjście.*

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite n i m oddzielone pojedynczym odstępem ($2 \leqslant n \leqslant 400$, $1 \leqslant m \leqslant 60\,000$). Oznaczają one odpowiednio liczbę skrzyżowań w Bajtogrodzie oraz liczbę łączących je ulic. W kolejnych m wierszach znajdują się opisy ulic. W wierszu $(i+1)$-szym znajdują się trzy wartości x_i, y_i, c_i ($1 \leqslant x_i \leqslant n$, $1 \leqslant y_i \leqslant n$, $x_i \neq y_i$), pooddzielane pojedynczymi odstępami i oznaczające odpowiednio początek ulicy, koniec ulicy oraz pierwszą literę jej nazwy w postaci małej litery alfabetu angielskiego. Między dowolnymi dwoma skrzyżowaniami istnieje maksymalnie jedna ulica w danym kierunku. Kolejny wiersz zawiera jedną liczbę całkowitą d ($2 \leqslant d \leqslant 100$), oznaczającą liczbę skrzyżowań, pomiędzy którymi idzie Bajtuś w drodze do szkoły. Następny wiersz zawiera d liczb całkowitych s_i ($1 \leqslant s_i \leqslant n$), pooddzielanych pojedynczymi odstępami. Oznaczają one, że Bajtuś mieszka przy skrzyżowaniu numer s_1, szkoła znajduje się przy skrzyżowaniu s_d, zaś po drodze Bajtuś idzie **kolejno** *po kolegów mieszkających przy skrzyżowaniach $s_2, s_3, \ldots, s_{d-1}$. Każde dwa następujące po sobie*

numery skrzyżowań na liście są różne. Może się jednak zdarzyć, że pewne numery skrzyżowań na liście będą takie same. Ponadto, jeśli pomiędzy dwoma kolejnymi skrzyżowaniami nie da się przejść, stosując się do ograniczeń podanych w zadaniu, to Bajtuś idzie na przełaj i niczego nie zapisuje na kartce.

Wyjście

Wyjście powinno składać się z $d-1$ wierszy. W i-tym wierszu powinna znajdować się liczba r_i oraz ciąg znaków w_i, oddzielone pojedynczym odstępem. Liczba r_i oznacza długość najkrótszej drogi, spełniającej wymogi zadania, łączącej skrzyżowania s_i oraz s_{i+1}, zaś w_i to ciąg pierwszych liter nazw ulic na tej drodze. W przypadku gdy pomiędzy danymi dwoma skrzyżowaniami nie istnieje trasa spełniająca warunki zadania, należy wypisać -1. Jeśli zaś istnieje kilka możliwych ciągów znaków w_i zgodnych z warunkami zadania, należy wypisać dowolny z nich.

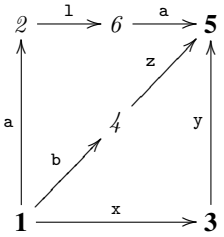
Przykład

Dla danych wejściowych:

```
6 7
1 2 a
1 3 x
1 4 b
2 6 l
3 5 y
4 5 z
6 5 a
3
1 5 3
```

poprawnym wynikiem jest:

```
3 ala
-1
```



Rozwiązanie

Podejście grafowe

Postawiony w zadaniu problem można w naturalny sposób opisać za pomocą grafu, w którym wierzchołkami są skrzyżowania w Bajtogrodzie. Krawędzie w tym grafie są skierowane, łączą pary różnych skrzyżowań i posiadają etykiety oznaczające pierwsze litery nazw

poszczególnych ulic. Oznaczmy tak skonstruowany graf przez $G = (V, E)$ — zgodnie z treścią zadania ma on n wierzchołków i m krawędzi. Naszym zadaniem jest (wielokrotne) wyszukiwanie w grafie G takich ścieżek, by litery na kolejnych krawędziach ścieżki tworzyły palindrom[1].

W pierwszej wersji algorytmu będziemy takie wyszukiwanie wykonywać osobno dla każdego fragmentu trasy Bajtusia. Zauważmy, że jeśli z pierwszego wierzchołka takiego fragmentu wychodzimy krawędzią o etykiecie c, to do ostatniego wierzchołka tego fragmentu musimy również wejść krawędzią etykietowaną tą literą. To rodzi następujący pomysł: wykonujmy jednocześnie ruch z wierzchołka początkowego do końcowego oraz z końcowego w kierunku początku (idąc *pod prąd*), po krawędziach o tej samej etykiecie. Jeśli w którymś momencie oba przemieszczające się wskaźniki spotkają się, będziemy wiedzieć, że znaleźliśmy szukany palindrom.

Mówiąc bardziej formalnie, niech $G' = (V', E')$ będzie grafem par wierzchołków grafu G. Pomiędzy wierzchołkami $(a_1, b_1),\ (a_2, b_2) \in V'$ dodajemy krawędź skierowaną o etykiecie c, jeśli w grafie G istnieją krawędzie $a_1 \xrightarrow{c} a_2$ oraz $b_2 \xrightarrow{c} b_1$ o etykietach c. W tak zdefiniowanym grafie ścieżka $(a_1, b_1) \to (a_2, b_2) \to \ldots \to (a_k, b_k)$ odpowiada sytuacji, w której początkowymi wierzchołkami na trasie Bajtusia są $a_1, a_2, \ldots, a_k$, końcowymi — wierzchołki: $b_k, b_{k-1}, \ldots, b_1$, i dodatkowo wiemy, że pierwsze $k-1$ liter z opisu drogi jest równe ostatnim $k-1$ literom, w odwróconej kolejności.

Aby znaleźć drogę biegnącą w grafie G z wierzchołka *start* do wierzchołka *meta*, której opis jest palindromem parzystej długości, wystarczy w G' znaleźć ścieżkę postaci:

$$(start, meta) \to (a_2, b_2) \to \ldots \to (v, v)$$

dla pewnego $v \in V$. Natomiast trasie w G, którą da się opisać palindromem nieparzystej długości, w G' odpowiada ścieżka postaci:

$$(start, meta) \to (a_2, b_2) \to \ldots \to (v, w),$$

przy czym w grafie G musi istnieć krawędź $v \to w$. Wierzchołki postaci (v, v) oraz takie wierzchołki (v, w), że w G istnieje krawędź $v \to w$, będziemy odtąd nazywać *wierzchołkami końcowymi* grafu G'.

Ponieważ każdorazowo szukamy trasy o najmniejszej możliwej długości, do wyszukiwania ścieżek możemy użyć algorytmu przeszukiwania grafu wszerz, czyli BFS[2]. Przypomnijmy, że algorytm ten pozwala na znalezienie nie tylko długości najkrótszej ścieżki, ale także, na podstawie tablicy przodków (przodkiem wierzchołka v nazywamy wierzchołek, z którego do v przyszliśmy), na odtworzenie samej tej ścieżki, co jest wymagane w zadaniu. Pozostaje obliczyć rozmiar grafu G', co pozwoli nam ocenić złożoność algorytmu. Każdy wierzchołek G' powstaje z pary wierzchołków grafu G, więc sumaryczna liczba wierzchołków w G' to n^2. Podobnie, z każdej pary krawędzi w G o takiej samej etykiecie powstaje jedna krawędź w G', więc pesymistycznie — tj. w przypadku, gdy dużo krawędzi ma takie same etykiety — liczba krawędzi w G' może być rzędu m^2. Ponieważ dla każdego fragmentu podróży Bajtusia musimy wykonać od nowa przeszukiwanie grafu, więc złożoność czasowa takiego algorytmu to $O(d \cdot (n^2 + m^2))$, zaś pamięciowa — $O(n^2 + m^2)$. Jest on zaimplementowany w pliku `przb2.cpp` — takie rozwiązanie zdobywało na zawodach 40% punktów.

[1] *Palindrom* — słowo wyglądające tak samo czytane zarówno normalnie, jak i od tyłu, np. `kajak`.

[2] Opis algorytmu BFS można znaleźć w większości książek poświęconych algorytmice, np. [20].

Złożoność pamięciową można w łatwy sposób poprawić do $O(n^2+m) = O(n^2)$, jeśli zauważymy, że nie musimy trzymać w pamięci całego grafu — wszystkie potrzebne krawędzie możemy generować na bieżąco na podstawie tych z grafu G (składnik n^2 w oszacowaniu złożoności wynika z rozmiaru struktur danych wykorzystywanych w algorytmie BFS). Implementacje tak poprawionego algorytmu można znaleźć w plikach `przs3.cpp`, `przs13.pas` oraz `przs23.java`; takie rozwiązania zdobywały na zawodach już 50% punktów.

Odwracamy krawędzie

Dla danych o maksymalnych rozmiarach powyższe rozwiązanie jest zdecydowanie zbyt wolne. Jego koncepcja jest jednak słuszna — wystarczy kilka usprawnień, aby dojść do rozwiązania wzorcowego. W pierwszej kolejności zmniejszymy wpływ czynnika d na złożoność algorytmu. Każdorazowe uruchomienie algorytmu BFS jest dosyć podobne: zawsze szukamy najkrótszej ścieżki ze wskazanego wierzchołka do jednego z wierzchołków końcowych. Co ważne, zbiór wierzchołków końcowych jest zawsze taki sam. Odwróćmy więc krawędzie naszego grafu i rozpocznijmy przeszukiwanie wszerz od wierzchołków końcowych. Wystarczy jedynie wstawić je wszystkie do kolejki na początku przeszukiwania BFS, a wyznaczymy odległości od każdego wierzchołka w grafie do wierzchołka końcowego (w grafie nieodwróconym)[3]. Ponadto, jak łatwo zauważyć, także w tym przypadku na podstawie tablicy przodków obliczonej podczas przeszukiwania można skonstruować wszystkie wynikowe palindromy. Takie rozwiązanie działa w złożoności czasowej $O(dn^2+m^2)$, gdyż rozmiar każdego z d palindromów jest w najgorszym przypadku rzędu $O(n^2)$. Skąd to wiadomo? Trzeba zauważyć, że wynik jest najkrótszą ścieżką (oczywiście w przypadku, gdy takowa istnieje) pomiędzy pewnymi dwoma wierzchołkami w grafie o n^2 wierzchołkach. Takie rozwiązanie, zaimplementowane w plikach `przs2.cpp`, `przs12.pas` oraz `przs22.java`, na zawodach zdobywało 70% punktów.

Rozwiązanie wzorcowe

Rozwiązanie wzorcowe opiera się na pomyśle podobnym do przedstawionego powyżej, jednak realizuje go nieco efektywniej. W tym celu stosuje sztuczkę, która została zaproponowana w rozwiązaniu zadania *Agenci* z VII Olimpiady Informatycznej (więcej w [7]). Zauważmy, że w grafie G' przejście po jednej krawędzi odpowiada dwóm przejściom w grafie G: wykonujemy krok naprzód z początku oraz krok wstecz z końca drogi. Nieco zaskakujący może wydawać się fakt, że jeśli kroki te będziemy wykonywać osobno, uzyskamy algorytm o istotnie lepszym czasie działania.

Skonstruujmy graf $G'' = (V'', E'')$, który będzie modelował takie podejście. Wierzchołkami G'' są trójki (u,v,c), przy czym $u,v \in V$, natomiast $c \in \{\mathtt{a},\mathtt{b},\ldots,\mathtt{z},\#\} \stackrel{\text{def}}{=} A$. Trójka

[3]Przeszukiwanie rozpoczęte z wielu wierzchołków naraz można wyobrażać sobie tak, że dokładamy do grafu jedno sztuczne źródło, które łączymy krawędziami ze wszystkimi wierzchołkami końcowymi, i to od niego rozpoczynamy przeszukiwanie. Wówczas tuż po przetworzeniu owego źródła kolejka wykorzystywana w algorytmie będzie zawierała właśnie wszystkie wierzchołki końcowe. Taka interpretacja stanowi więc zarazem uzasadnienie poprawności zmodyfikowanego algorytmu BFS.

$(u,v,\#)$ odpowiada „zwykłemu" wierzchołkowi $(u,v) \in V'$; w szczególności, taki wierzchołek nazwiemy wierzchołkiem końcowym grafu G'', jeżeli (u,v) był wierzchołkiem końcowym grafu G'. Wszystkie pozostałe wierzchołki grafu G'', tj. wierzchołki postaci (u,v,c) dla $c \neq \#$, reprezentują pozycje pośrednie, osiągane za pomocą pojedynczych kroków z tych „zwykłych" wierzchołków.

Dokładniej, osiągnięcie wierzchołka $(u,v,\#)$ w przeszukiwaniu z wierzchołków końcowych będzie odpowiadało znalezieniu ścieżki z u do v w grafie G, której opis jest palindromem (czyli dokładnie tak samo, jak w przypadku wierzchołka (u,v) w G'). Natomiast trójka (u,v,c) dla $c \in \{\mathtt{a},\mathtt{b},\ldots,\mathtt{z}\}$, napotkana podczas przeszukiwania, będzie oznaczać znalezienie takiej ścieżki z u do v w G, która po doklejeniu na samym początku litery c staje się palindromem. W tym celu, krawędzie w G'' są zdefiniowane następująco:

$$\begin{aligned} (u,v,\#) &\longrightarrow (u',v,c) \qquad \text{jeżeli w } G \text{ jest krawędź } u \xrightarrow{c} u' \\ (u,v,c) &\longrightarrow (u,v',\#) \qquad \text{jeżeli w } G \text{ jest krawędź } v' \xrightarrow{c} v. \end{aligned}$$

Jeśli przez $|A|$ oznaczymy rozmiar alfabetu A, to tak skonstruowany graf ma, jak łatwo sprawdzić, $O(n^2 \cdot |A|)$ wierzchołków oraz $O(nm)$ krawędzi. W rozwiązaniu wzorcowym wszystkie kroki wcześniej opisanego rozwiązania wykonujemy na grafie G'' zamiast G', tzn.: odwracamy krawędzie G'', wykonujemy przeszukiwanie BFS z wierzchołków końcowych i odczytujemy wyniki z wierzchołków postaci $(start, meta, \#)$. Otrzymany w ten sposób algorytm działa w czasie $O\big(n^2 \cdot (|A|+d) + mn\big)$ i pamięci rzędu $O(n^2 \cdot |A|)$, ponieważ krawędzi G'', podobnie jak w przypadku G', nie musimy pamiętać bezpośrednio, tylko konstruujemy je na bieżąco na podstawie krawędzi G. Implementacje rozwiązania wzorcowego można znaleźć w plikach `prz.cpp`, `prz1.pas` oraz `prz2.java`.

Testy

Do oceny rozwiązań zawodników użyto 10 zestawów testowych. Przygotowane testy można podzielić na pięć kategorii:

Graf z losowymi palindromami Generujemy pewną liczbę losowych palindromów, a następnie dla każdego palindromu dodajemy do grafu losowo umiejscowioną ścieżkę o etykietach zgodnych z kolejnymi literami palindromu.

Graf z długim wynikiem Łączymy wierzchołek 1 z wierzchołkiem 2 krawędzią o etykiecie `a` oraz wierzchołek 2 z wierzchołkiem 3 krawędzią o etykiecie `z`. Dodatkowo wierzchołki 2 i 3 wpinamy w cykle długości odpowiednio c_1 i c_2, w których wszystkie krawędzie mają etykietę `a`. Na koniec dokładamy trochę losowych krawędzi z drugiego cyklu do pierwszego. W takim grafie, jeśli $\mathrm{NWD}(c_1,c_2) = 1$, najkrótszy palindrom z 1 do 3 ma długość zbliżoną do $c_1 \cdot c_2$.

Graf z cyklem Cykl złożony z krawędzi o ustalonej etykiecie, z dodanymi losowymi krawędziami.

Graf gęsty Dwa bardzo gęste losowe grafy połączone jedną krawędzią w każdą stronę.

Graf długi Graf, w którym dodajemy losowe krawędzie, ale tylko między wierzchołkami, których indeksy różnią się co najwyżej o ustaloną wartość.

W poniższej tabelce znajdują się opisy poszczególnych testów, przy czym n oznacza liczbę skrzyżowań, m — liczbę ulic, a d — liczbę punktów występujących na trasie Bajtusia z domu do szkoły.

Nazwa	n	m	d	Opis
prz1.in	5	12	4	graf z losowymi palindromami
prz2.in	7	12	25	graf z losowymi palindromami
prz3.in	60	244	40	graf z losowymi palindromami
prz4.in	100	200	100	graf z cyklem
prz5a.in	87	315	30	graf długi
prz5b.in	104	4030	96	graf gęsty
prz6a.in	200	10408	41	graf z losowymi palindromami
prz6b.in	277	3031	9	graf z długim wynikiem
prz7a.in	398	11500	5	graf długi
prz7b.in	400	400	3	graf z cyklem
prz8a.in	399	53472	94	graf długi
prz8b.in	378	13532	94	graf z losowymi palindromami
prz9a.in	400	401	100	graf z długim wynikiem
prz9b.in	398	60000	97	graf gęsty
prz10a.in	394	12532	94	graf długi
prz10b.in	400	60000	97	graf gęsty

Jakub Wojtaszczyk
Treść zadania, Opracowanie

Bartosz Górski
Program

Dostępna pamięć: 32 MB.

OI, Etap II, dzień drugi, 12.02.2009

Architekci

Król Bajtazar postanowił wybudować sobie nowy pałac. Ogłosił więc konkurs na najlepszy projekt architektoniczny pałacu. Chcąc zmotywować architektów do spiesznej pracy, ogłosił też, że projekty będzie rozpatrywał w takiej kolejności, w jakiej będą nadsyłane.

Z tym zleceniem wiąże się ogromny prestiż, dlatego też architekci z całego świata nadsyłają swoje propozycje do królewskiej kancelarii. Projektów nadchodzi bardzo dużo, a Bajtazar nie ma czasu ich wszystkich przeglądać. Zrezygnował zatem z samodzielnego wykonywania tej czynności i poprosił swojego kanclerza o to, by wstępnie przejrzał nadchodzące projekty zgodnie z następującymi zasadami:

- *kanclerz ma wybrać k projektów, odrzucając resztę od razu — Bajtazar wie, że więcej niż k projektów i tak nie będzie w stanie przejrzeć;*
- *projekty mają zostać przedstawione Bajtazarowi w takiej kolejności, w jakiej zostały nadesłane — w takiej też kolejności Bajtazar będzie je przeglądał, zgodnie z tym co ogłosił;*
- *spośród wszystkich ciągów k projektów spełniających powyższe warunki, kanclerz ma wybrać ciąg* **najlepszy**, *zgodnie z poniższą definicją.*

  Powiemy, że ciąg projektów $(p_1, p_2, \ldots, p_k)$ jest **lepszy** *od ciągu $(r_1, r_2, \ldots, r_k)$, jeśli dla pewnego $l \geqslant 1$ pierwszych $l-1$ projektów w obu ciągach jest równie dobrych, zaś l-ty projekt w ciągu p jest lepszy od l-tego projektu w ciągu r (czyli $p_i = r_i$ dla $i < l$ i $p_l > r_l$).*

Projekty cały czas nadchodzą i nie wiadomo, do kiedy Bajtazar rozkaże je przyjmować. Kanclerz nie chce zostawiać wyboru k projektów na ostatni moment, jednak bardzo boi się popełnić błąd i narazić na gniew króla. Dlatego poprosił Cię o pomoc.
Napisz program, który:

- *pobierze za pomocą dostarczonej biblioteki liczbę k oraz ciąg liczb reprezentujących jakość kolejnych projektów,*
- *wyznaczy najlepszy ciąg k projektów, zgodnie z podanymi zasadami,*
- *zwróci jakości wybranych projektów za pomocą biblioteki.*

Opis użycia biblioteki

Aby użyć biblioteki, należy wpisać w swoim programie:

- *C/C++:* `#include "carclib.h"`
- *Pascal:* `uses parclib;`
- *Java: nic nie trzeba robić, lecz by uruchomić rozwiązanie, należy mieć skompilowaną bibliotekę jarclib (plik* `jarclib.class`*) w tym samym katalogu co program.*

Biblioteka udostępnia trzy procedury, funkcje lub metody statyczne:

- `inicjuj` — *zwraca liczbę całkowitą k ($1 \leqslant k \leqslant 1\,000\,000$), określającą, jak wiele projektów ma zawierać ciąg wynikowy. Powinna być użyta dokładnie raz, na samym początku działania programu.*
  - *C/C++:* `int inicjuj();`
  - *Pascal:* `function inicjuj(): longint;`
  - *Java:* `public static int inicjuj();`, *będąca metodą statyczną klasy* `jarclib`.
- `wczytaj` — *i-te wywołanie zwraca liczbę całkowitą p_i ($1 \leqslant p_i \leqslant 1\,000\,000\,000$) oznaczającą jakość i-tego projektu (im większa liczba, tym lepszy projekt), albo 0, co oznacza, że nie ma już więcej projektów. Liczba projektów nie jest znana przed wczytaniem danych, jednak możesz założyć, że wszystkich projektów jest przynajmniej k, a co najwyżej $15\,000\,000$. Funkcja ta powinna być wywoływana do momentu, aż skończą się projekty, i* **ani razu więcej**.
  - *C/C++:* `int wczytaj();`
  - *Pascal:* `function wczytaj(): longint;`
  - *Java:* `public static int wczytaj();`, *będąca metodą statyczną klasy* `jarclib`.
- `wypisz` — *za pomocą tej procedury/funkcji wypisujesz jakości kolejnych projektów, które kanclerz przedstawi królowi. Powinna być ona użyta dokładnie k razy; w i-tym wywołaniu należy podać jakość i-tego w kolejności projektu. k-te wywołanie tej procedury/funkcji zakończy działanie Twojego programu.*
  - *C/C++:* `void wypisz(int jakoscProjektu);`
  - *Pascal:* `procedure wypisz(jakoscProjektu: longint);`
  - *Java:* `public static void wypisz(int jakoscProjektu);`, *będąca metodą statyczną klasy* `jarclib`.

Twój program nie może otwierać żadnych plików ani używać standardowego wejścia i wyjścia. Rozwiązanie będzie kompilowane wraz z biblioteką za pomocą następujących poleceń:

- *C:* `gcc -O2 -static carclib.c arc.c -lm`
- *C++:* `g++ -O2 -static carclib.c arc.cpp -lm`
- *Java:* `javac arc.java`, *a skompilowany plik biblioteki* `jarclib` — `jarclib.class` — *będzie się znajdował w tym samym katalogu.*
- *Pascal:* `ppc386 -O2 -XS -Xt arc.pas`, *a plik biblioteki* `parclib` *będzie znajdował się w tym samym katalogu.*

W katalogu `/home/zawodnik/rozw/lib` *możesz znaleźć przykładowe pliki bibliotek i przykładowe rozwiązania ilustrujące sposób ich użycia. Przykładowa biblioteka wczytuje scenariusz testowy ze standardowego wejścia w następującym formacie:*

- *Pierwszy wiersz wejścia zawiera jedną dodatnią liczbę całkowitą k.*
- *Kolejne wiersze wejścia zawierają po jednej dodatniej liczbie całkowitej; $(i+1)$-szy wiersz zawiera liczbę p_i, oznaczającą jakość i-tego zgłoszonego projektu.*
- *Ostatni wiersz wejścia zawiera liczbę 0, oznaczającą koniec listy projektów.*

Przykładowa biblioteka wypisuje na standardowe wyjście k wierszy — jakości projektów zgłoszone przez program.

Przykładowy przebieg programu

C/C++	Pascal	Java	Zwracane wartości i wyjaśnienia
`k = inicjuj();`	`k := inicjuj();`	`k = jarclib.inicjuj();`	Od tego momentu $k = 3$.
			Wczytywanie jakości kolejnych projektów.
`wczytaj();`	`wczytaj();`	`jarclib.wczytaj();`	12
`wczytaj();`	`wczytaj();`	`jarclib.wczytaj();`	5
`wczytaj();`	`wczytaj();`	`jarclib.wczytaj();`	8
`wczytaj();`	`wczytaj();`	`jarclib.wczytaj();`	3
`wczytaj();`	`wczytaj();`	`jarclib.wczytaj();`	15
`wczytaj();`	`wczytaj();`	`jarclib.wczytaj();`	8
`wczytaj();`	`wczytaj();`	`jarclib.wczytaj();`	0 — oznacza koniec ciągu projektów
			Wypisujemy rozwiązanie, czyli 3-elementowy ciąg:
`wypisz(12);`	`wypisz(12);`	`jarclib.wypisz(12);`	12
`wypisz(15);`	`wypisz(15);`	`jarclib.wypisz(15);`	15
`wypisz(8);`	`wypisz(8);`	`jarclib.wypisz(8);`	8

Rozwiązanie

Wprowadzenie

Ciąg nazywany w treści zadania „lepszym” najczęściej określa się mianem „leksykograficznie większego”; w niniejszym opracowaniu będziemy używali drugiego z tych terminów.

Wpierw zastanówmy się chwilę, co dokładnie oznacza określenie największy leksykograficznie podciąg długości k. Otóż jeżeli mamy trzy różne podciągi A, B i C i A jest większy leksykograficznie od B, zaś B — większy leksykograficznie od C, to oczekiwalibyśmy, że A jest większy leksykograficznie od C. I faktycznie — A różni się od B na pewnej pozycji l_{AB}, zaś B od C na pewnej pozycji l_{BC}. Jeśli $l_{AB} = l_{BC}$, to ciągi A i C są takie same aż do pozycji l_{AB}, a potem $A_{l_{AB}} > B_{l_{AB}} > C_{l_{AB}}$. Jeśli $l_{AB} > l_{BC}$, to ciągi A i C są takie same aż do l_{BC}, a potem $A_{l_{BC}} = B_{l_{BC}} > C_{l_{BC}}$; analogicznie, jeśli $l_{AB} < l_{BC}$, to $A_{l_{AB}} > B_{l_{AB}} = C_{l_{AB}}$. Czyli, w szczególności, jeśli zaczniemy od dowolnego ciągu k–elementowego, to — tak długo, jak istnieje jakikolwiek leksykograficznie większy — możemy przechodzić do tego większego, i nigdy

nie wrócimy do raz przejrzanego ciągu. Jako że wszystkich podciągów jest skończenie wiele, to w końcu zatrzymamy się i trafimy na ciąg, od którego żaden inny nie jest leksykograficznie większy. Taki ciąg nazwiemy najlepszym albo leksykograficznie największym.

Zauważmy teraz jeszcze, że jeśli dwa ciągi różnią się, to istnieje pierwsza pozycja l, na której się różnią, i porównując elementy na tej pozycji jesteśmy w stanie stwierdzić, który z nich jest leksykograficznie większy. Nie oznacza to, że dany ciąg ma tylko jeden leksykograficznie największy podciąg (bo np. w ciągu samych jedynek każdy podciąg jest leksykograficznie największy), ale oznacza, że każde dwa podciągi leksykograficznie największe są równe (a więc w szczególności to, co mamy wypisać, jest wyznaczone jednoznacznie).

To, że może istnieć więcej niż jeden największy ciąg, nastręcza pewnych trudności przy analizowaniu rozwiązania. Poradzimy sobie z tym problemem, pokazując jeden konkretny największy ciąg i starając się właśnie ten ciąg skonstruować. Załóżmy, że największy ciąg będzie miał wartości $(a_1, a_2, \ldots, a_k)$. Wówczas wybierzemy ten ciąg, który zawiera pierwsze wystąpienie a_1, potem pierwsze wystąpienie a_2 występujące po wybranym już a_1, następnie pierwsze wystąpienie a_3 po wybranym a_2 itd. Taki ciąg otrzymamy, jeżeli ustalimy, że z dwóch równie dobrych projektów za lepszy uznamy ten, który wpłynął wcześniej. Przy takim założeniu największy leksykograficznie ciąg jest już wyznaczony jednoznacznie.

Naszym zadaniem jest znalezienie największego leksykograficznie podciągu długości k w ciągu o nieznanej długości $n \leqslant 15\,000\,000$. Cały ciąg n liczb nie mieści się w pamięci — faktycznie, pojedyncza liczba z przedziału $[1,\ 1\,000\,000\,000]$ musi zająć przynajmniej $\log_2 1\,000\,000\,000 > 29$ bitów, zatem 15 000 000 takich liczb musi zająć ponad 43 MB, a więc istotnie więcej niż dostępne 32 MB. Będziemy zatem musieli w trakcie działania programu w jakiś sposób odrzucać z pamięci wartości, o których wiemy, że już się nam nie przydadzą.

Rozwiązanie wzorcowe

W tym rozwiązaniu będziemy w każdym kroku przechowywać w pamięci wyłącznie największy leksykograficznie ciąg długości k wybrany z dotychczas wczytanych elementów. Żeby to podejście mogło zadziałać, musimy zrozumieć, jak — mając największy leksykograficznie podciąg długości k w ciągu $(x_1, x_2, \ldots, x_m)$ — wybrać największy leksykograficznie podciąg długości k w ciągu $(x_1, x_2, \ldots, x_m, x_{m+1})$. Pomogą nam w tym następujące spostrzeżenia:

Fakt 1. *Niech $(x_1, x_2, \ldots, x_m)$ będzie dowolnym ciągiem liczb. Niech dalej $(x_{i_1}, x_{i_2}, \ldots, x_{i_k})$ będzie największym leksykograficznie podciągiem długości k tego ciągu (oczywiście $k \leqslant m$ oraz $i_1 < i_2 < \ldots < i_k$). Wtedy $x_{i_1} = \max\{x_1, \ldots, x_{m-k+1}\}$ (przypomnijmy, że w przypadku remisu jako i_1 wybieramy pierwsze wystąpienie maksimum), zaś $(x_{i_2}, x_{i_3}, \ldots, x_{i_k})$ jest największym $(k-1)$–elementowym podciągiem ciągu $(x_{i_1+1}, x_{i_1+2}, \ldots, x_m)$.*

Dowód: Skoro $(x_{i_1}, x_{i_2}, \ldots, x_{i_k})$ jest podciągiem $(x_1, x_2, \ldots, x_m)$, to

$$\begin{aligned} i_k &\leqslant m \\ i_{k-1} &\leqslant m-1 \\ &\ldots \\ i_1 &\leqslant m-k+1 \end{aligned}$$

Zatem x_{i_1} musi być jedną z liczb $x_1,\ldots,x_{m-k+1}$. Załóżmy, że $x_{i_1} \neq \max\{x_1,\ldots,x_{m-k+1}\}$. Wówczas $x_{i_1} < \max\{x_1,\ldots,x_{m-k+1}\} \stackrel{\text{def}}{=} x_j$. Ciąg $(x_j,x_{j+1},\ldots,x_{j+k-1})$ jest więc podciągiem $(x_1,x_2,\ldots,x_m)$ (bo $j+k-1 \leqslant m$) oraz jest leksykograficznie większy od ciągu $(x_{i_1},x_{i_2},\ldots,x_{i_k})$ (co otrzymujemy, wstawiając w definicji $l=1$), a zatem ciąg $(x_{i_1},x_{i_2},\ldots,x_{i_k})$, wbrew założeniu, nie mógł być największym leksykograficznie podciągiem długości k.

Drugą część faktu otrzymujemy już teraz wprost z definicji leksykograficznej większości. ■

Fakt 2. *Jeśli* $(x_{i_1},x_{i_2},\ldots,x_{i_k})$ *jest największym leksykograficznie podciągiem długości* k *ciągu* $(x_1,x_2,\ldots,x_m)$*, to największy leksykograficznie podciąg* $(x_{j_1},x_{j_2},\ldots,x_{j_k})$ *długości* k *ciągu* $(x_1,x_2,\ldots,x_m,x_{m+1})$ *jest podciągiem ciągu* $(x_{i_1},x_{i_2},\ldots,x_{i_k},x_{m+1})$.

Dowód: Dowód przebiega przez indukcję ze względu na k. Dla $k=1$ największy leksykograficznie podciąg danego ciągu to po prostu jego największy element — i faktycznie, największy element $(x_1,x_2,\ldots,x_{m+1})$ to albo największy element $(x_1,x_2,\ldots,x_m)$, albo też x_{m+1}.

Załóżmy teraz, że teza faktu zachodzi dla wszystkich ciągów długości $k-1$, i przyjrzyjmy się ciągom długości k. Na mocy Faktu 1 wiemy, że x_{j_1} jest równe największej spośród liczb $x_1,x_2,\ldots,x_{m-k+2}$. Wiemy też, że

$$x_{i_1} = \max\{x_1,x_2,\ldots,x_{m-k+1}\} \quad \text{oraz} \quad x_{i_2} = \max\{x_{i_1+1},x_{i_1+2},\ldots,x_{m-k+2}\}.$$

Zatem $\max\{x_{i_1},x_{i_2}\} = x_{j_1}$.

Jeśli $x_{i_1} = x_{j_1}$, to ciąg $(x_{i_2},x_{i_3},\ldots,x_{i_k})$ jest największym podciągiem długości $k-1$ ciągu $(x_{i_1+1},x_{i_1+2},\ldots,x_m)$, zaś $(x_{j_2},x_{j_3},\ldots,x_{j_k})$ jest największym podciągiem długości $k-1$ ciągu $(x_{i_1+1},x_{i_1+2},\ldots,x_m,x_{m+1})$. Zatem na mocy założenia indukcyjnego $(x_{j_2},x_{j_3},\ldots,x_{j_k})$ jest podciągiem $(x_{i_2},x_{i_3},\ldots,x_{i_k},x_{m+1})$, wobec czego $(x_{j_1},x_{j_2},x_{j_3},\ldots,x_{j_k})$ jest podciągiem $(x_{i_1},x_{i_2},x_{i_3},\ldots,x_{i_k},x_{m+1})$. Jeśli natomiast $x_{j_1} = x_{i_2}$, to $j_1 = i_2 = m-k+2$. To oznacza, że

$$i_w = m-k+w \text{ dla } w=2,3,\ldots,k \quad \text{oraz} \quad j_{w-1} = m-k+w \text{ dla } w=2,3,\ldots,k+1,$$

czyli znowu $(x_{j_1},x_{j_2},x_{j_3},\ldots,x_{j_k})$ jest podciągiem $(x_{i_1},x_{i_2},x_{i_3},\ldots,x_{i_k},x_{m+1})$. ■

Umiemy już zatem ominąć problem braku pamięci — wystarczy po wczytaniu pierwszych m elementów przechowywać w pamięci wyłącznie największy leksykograficznie podciąg długości k ciągu $(x_1,x_2,\ldots,x_m)$ i wczytawszy kolejny element, poprawiać ten ciąg tak, by był największym leksykograficznie podciągiem ciągu $(x_1,x_2,\ldots,x_{m+1})$. Dochodzimy teraz do algorytmicznego problemu efektywnego znajdowania tego największego k–elementowego podciągu spośród $k+1$ elementów.

Ponieważ wszystkich takich podciągów jest $k+1$ (wystarczy wybrać element, który nie należy do naszego podciągu), a porównanie leksykograficzne dwóch podciągów można wykonać w czasie $O(k)$, więc koszt czasowy skrajnie siłowego podejścia to $O(nk^2)$. Wobec ograniczeń na n i k potrzebne jest zastosowanie innego podejścia. Zastanówmy się mianowicie, w jakich sytuacjach usunięcie pierwszego spośród $k+1$ elementów $(x_{i_1},x_{i_2},\ldots,x_{i_k},x_{m+1})$ daje największy leksykograficznie ciąg długości k. Oczywiście usunięcie tego elementu prowadzi do ciągu rozpoczynającego się od x_{i_2}. Jeżeli $x_{i_1} > x_{i_2}$, to widzimy, że w ten sposób nie możemy otrzymać największego ciągu, gdyż na pewno lepszym

kandydatem jest ciąg złożony z k początkowych elementów. Przypadek $x_{i_1} = x_{i_2}$ na chwilę pomińmy. Jeżeli zaś $x_{i_1} < x_{i_2}$, to na pewno usunięcie x_{i_1} prowadzi do ciągu większego niż $(x_{i_1}, x_{i_2}, \ldots, x_{i_k})$. Co więcej, ponieważ każdy k–elementowy podciąg wyjściowego ciągu poza tym z usuniętym x_{i_1} zaczyna się od x_{i_1}, widzimy, że usuwając x_{i_1}, w tym przypadku otrzymujemy na pewno największy leksykograficznie podciąg.

Podobnie sytuacja ma się dla kolejnych elementów: jeżeli i-ty element jest większy od następnego, to na pewno nie opłaca się go usuwać, a jeżeli mniejszy, to na pewno jego usunięcie jest najlepszą możliwą decyzją — oczywiście o ile przypadek ten nie wystąpił dla żadnego z wcześniejszych elementów. Zauważmy dalej, że w pomijanym dotychczas przypadku równości kolejnych elementów tak naprawdę nie musimy rozważać możliwości usunięcia pierwszego z nich, gdyż dokładnie taki sam ciąg otrzymamy, jeżeli usuniemy drugi — to pokazuje, że ten przypadek sprowadza się w końcu do jednego z dwóch pozostałych. Podsumowując: aby uzyskać podciąg największy leksykograficznie, musimy znaleźć najdłuższy fragment ciągu, w którym jest on nierosnący, a następnie usunąć jego ostatni element (zauważmy, że to sformułowanie uwzględnia przypadek, w którym cały wyjściowy ciąg jest nierosnący). W ten sposób możemy zredukować złożoność czasową rozwiązania do $O(nk)$:

```
1:  k := inicjuj();
2:  for i := 1 to k+1 do najwiekszy[i] := wczytaj();
3:  while najwiekszy[k+1] > 0 do
4:  begin
5:    opuszczany := 1;
6:    while opuszczany ⩽ k and
7:          najwiekszy[opuszczany] ⩾ najwiekszy[opuszczany+1] do
8:      opuszczany := opuszczany+1;
9:    for i := opuszczany to k do najwiekszy[i] := najwiekszy[i+1];
10:   najwiekszy[k+1] := wczytaj();
11: end
12: for i := 1 to k do wypisz(najwiekszy[i]);
```

Ale teraz możemy jeszcze zauważyć, że w powyższym rozwiązaniu po wielokroć przeglądamy i porównujemy te same elementy — a przecież jeżeli już przejrzeliśmy pewien początkowy odcinek ciągu i stwierdziliśmy, że jest on nierosnący, to możemy zrezygnować z przeglądania go ponownie. Musimy cofnąć się o jeden element ciągu (bo jeśli skasowaliśmy element na pozycji j, to teraz po pozycji $j-1$ następuje pozycja $j+1$, a pary $j-1, j+1$ jeszcze nie porównywaliśmy), natomiast wszystkie poprzednie pary już są sprawdzone.

```
1:  k := inicjuj();
2:  for i := 1 to k+1 do najwiekszy[i] := wczytaj();
3:  opuszczany := 1;
4:  while najwiekszy[k+1] > 0 do
5:  begin
6:    while opuszczany ⩽ k and
7:          najwiekszy[opuszczany] ⩾ najwiekszy[opuszczany+1] do
8:      opuszczany := opuszczany+1;
9:    for i := opuszczany to k do najwiekszy[i] := najwiekszy[i+1];
```

```
10:    if opuszczany ⩾ 1 then opuszczany := opuszczany − 1;
11:    najwiekszy[k+1] := wczytaj();
12: end
13: for i := 1 to k do wypisz(najwiekszy[i]);
```

Teraz już wyszukiwanie odbywa się w amortyzowanej złożoności czasowej $O(1)$. Dla każdego wczytanego elementu ciągu liczba *opuszczany* maleje o co najwyżej jeden, jej początkowa wartość to 1, końcowa to co najwyżej $k+1$, zatem może w trakcie działania całego programu wzrosnąć co najwyżej $n+k$ razy. Niestety, w powyższym programie mamy jeszcze przepisywanie ciągu (pętla **for** w linii 9), które może pesymistycznie wymagać $O(k)$ operacji przy każdym wczytanym elemencie. Ten problem jednak łatwo rozwiązać — wystarczy zamiast trzymać ciąg w tablicy, użyć do tego listy. W poniższym pseudokodzie korzystamy z dwukierunkowej listy *lista*, która umożliwia wykonywanie w stałej złożoności czasowej operacji: efektywnego dostępu do pierwszego i ostatniego elementu, przechodzenia do poprzedniego i następnego elementu listy, wstawiania i usuwania elementów i pobierania wartości elementu.

```
1: k := inicjuj();
2: for i := 1 to k+1 do lista.wstaw_na_koniec(wczytaj());
3: opuszczany := lista.poczatek();
4: while lista.koniec().wartosc > 0 do
5: begin
6:    while opuszczany ≠ lista.koniec() and
7:          opuszczany.wartosc ⩾ opuszczany.nastepny().wartosc do
8:       opuszczany := opuszczany.nastepny();
9:    if opuszczany ≠ lista.poczatek() then begin
10:      opuszczany := opuszczany.poprzedni();
11:      lista.usun(opuszczany.nastepny());
12:   end
13:   else begin
14:      opuszczany := opuszczany.nastepny();
15:      lista.usun(opuszczany.poprzedni());
16:   end
17:   lista.wstaw_na_koniec(wczytaj());
18: end
19: wsk := lista.poczatek();
20: while wsk ≠ lista.koniec() do
21: begin
22:    wypisz(wsk.wartosc);
23:    wsk := wsk.nastepny();
24: end
```

Rozwiązanie to zostało zaimplementowane w plikach `arc10.cpp` (samodzielna implementacja listy na tablicy), `arc.cpp` (lista z biblioteki STL), `arc1.c` i `arc2.pas` (samodzielne implementacje listy dynamicznej).

Rozwiązanie alternatywne

Zastanówmy się wpierw, jak rozwiązywalibyśmy nasze zadanie, gdybyśmy byli w stanie wczytać cały ciąg do pamięci. W tym rozwiązaniu również zakładamy, że spośród dwóch równie dobrych projektów lepszy jest ten, który pojawił się wcześniej.

Ponownie zacznijmy od Faktu 1. Wiemy, że x_{i_1} to największa spośród pierwszych $n-k+1$ liczb. Liczba x_{i_2} to największy element występujący po x_{i_1} spośród pierwszych $n-k+2$ liczb, i analogicznie x_{i_w} to zawsze największy element występujący po $x_{i_{w-1}}$ i nie dalej niż na pozycji $n-k+w$.

Aby wyznaczyć taki ciąg, wykorzystamy stos. Zakładamy, że stos jest początkowo pusty oraz że umożliwia wykonywanie każdej z następujących operacji w czasie $O(1)$: wstawienie elementu na szczyt stosu, zdjęcie elementu ze szczytu, sprawdzenie wartości elementu na szczycie i obliczenie rozmiaru struktury, czyli liczby zawartych w niej elementów.

```
1:  i := 1;
2:  while n+1-i+stos.rozmiar() > k do
3:    if stos.pusty() or stos.wierzch >= X[i] then
4:    begin
5:      if stos.rozmiar() < k then stos.wstaw(X[i]);
6:      i := i+1;
7:    end
8:    else stos.zdejmij_z_gory();
9:  while i <= n do
10: begin
11:   stos.wstaw(X[i]);
12:   i := i+1;
13: end
```

Powyższy pseudokod może wydawać się na pierwszy rzut oka nieco tajemniczy. W dalszej części tekstu pokażemy kolejno, że ma on własność stopu, czyli że w końcu zatrzymuje się, oraz że jest poprawny, czyli że ciąg uzyskany na stosie spełnia wymienione wcześniej warunki.

W uzasadnieniu własności stopu wystarczy skupić się na pętli **while** z linii 2. Zauważmy przede wszystkim, że liczba wykonań instrukcji z linii 8, odpowiadającej sytuacji, w której warunek z linii 3 nie zachodzi, jest nie większa niż liczba wykonań instrukcji z linii 5–6. Faktycznie, każdy element zdejmowany ze stosu musi wcześniej się tam znaleźć. To pokazuje, że w analizie wystarczy skupić się na instrukcjach z linii 5–6. Za każdym razem, gdy są one wykonywane, wartość zmiennej i wzrasta o 1. Wystarczy teraz zauważyć, że wartość i nie może nigdy przekroczyć $n+1$, co wynika z warunku pętli **while**, wobec faktu, że rozmiar stosu nie może przekroczyć k (patrz warunek w linii 5). Pokazaliśmy zatem, że łączna liczba obrotów pierwszej pętli **while** jest rzędu $O(n)$; analogiczne stwierdzenie w przypadku drugiej z pętli jest oczywiste.

Przejdźmy teraz do uzasadnienia poprawności wyniku. Po pierwsze, jak łatwo sprawdzić, w każdym obrocie pierwszej pętli **while** liczba $n+1-i+stos.rozmiar()$ albo nie zmienia się, albo maleje o jeden, zatem w momencie wyjścia z pętli będzie zachodzić

$n+1-i+stos.rozmiar() = k$. Wobec tego po wstawieniu pozostałych $n+1-i$ elementów na stos w drugiej pętli **while** faktycznie otrzymamy ciąg długości k.

Po drugie, kolejność elementów na stosie odpowiada kolejności elementów w pierwotnym ciągu, bo wstawiamy na stos w kolejności przeglądania.

Po trzecie, udowodnijmy, że otrzymany ciąg to faktycznie ten największy leksykograficznie. Zauważmy, że podczas wykonywania pierwszej pętli elementy na stosie tworzą ciąg posortowany (najmniejszy element znajduje się na szczycie stosu). Rozważmy największy element ciągu $x_1, x_2, \ldots, x_{n-k+1}$. Gdy go wczytamy, to zdejmiemy ze stosu wszystkie wcześniejsze elementy, a później już nigdy go nie zdejmiemy (elementy o numerach do $n-k+1$ są nie większe, więc go nie zdejmą, zaś elementy po $n-k+1$ nie zdejmą go, gdyż stos byłby zbyt mały). Wobec tego drugi element stosu będzie elementem występującym po x_{i_1}. Oczywiście będzie występował nie dalej niż na pozycji $n-k+2$ (bo po nim musi wystąpić jeszcze $k-2$ elementów). I będzie największym z elementów na rozpatrywanych pozycjach — znowu, kiedy ten największy element nadejdzie, to zdejmie wszystkie elementy poza pierwszym ze stosu, a następnie sam nie zostanie zdjęty. I dalej analogicznie — w–ty element stosu będzie największym elementem na pozycjach od $i_{w-1}+1$ do $n-k+w$, bo gdy największy element z tych pozycji nadejdzie, to zdejmie wszystkie elementy ze stosu poza pierwszymi $w-1$, a potem sam nie zostanie zdjęty. Zatem faktycznie otrzymany na stosie ciąg będzie największym leksykograficznie.

Niestety — w naszym zadaniu nie możemy wczytać całego ciągu do pamięci, w szczególności zaś nie znamy liczby n. Zauważmy jednak, że liczbę tę wykorzystujemy wyłącznie do sprawdzenia warunku $n+1-i+stos.rozmiar() > k$. Zatem, w szczególności, jeśli $n-i > k$, to nie musimy znać dokładnej wartości n, gdyż wiemy, że ten warunek jest spełniony. Możemy zatem nie wczytywać całego ciągu na raz, a jedynie czytać „o k elementów do przodu", by upewnić się, czy jest jeszcze wystarczająco wiele elementów ciągu przed nami — a jeżeli nie, to możemy już śmiało stosować algorytm z poprzedniego pseudokodu. Do implementacji opisanego podejście możemy zastosować tzw. *bufor* będący np. kolejką cykliczną implementowaną za pomocą tablicy k–elementowej:

```
1:  { Operacje wykonywane na samym początku. }
2:  for i := 1 to k do bufor[i] := wczytaj();
3:  n := k;
4:  pozycja := 1;
5:  koniec := false;
6:
7:  { Funkcja implementująca dostęp do kolejnego elementu ciągu wejściowego. }
8:  function wczytaj_przez_bufor() : int
9:  begin
10:   elem := bufor[pozycja];
11:   if not koniec then
12:   begin
13:     bufor[pozycja] := wczytaj();
14:     if bufor[pozycja] = 0
15:       then koniec := true;
16:       else n := n+1;
17:   end
18:   pozycja := pozycja+1;
```

```
19:   if pozycja = k+1 then pozycja := 1;
20:   return elem;
21: end
```

Rozwiązanie to zostało zaimplementowane w `arc3.cpp`, `arc4.pas`, działa w złożoności czasowej $O(n+k)$ i pamięciowej $O(k)$.

Testy

Zadanie było sprawdzane na 10 zestawach danych testowych, po trzy testy w każdym zestawie. Testy „a" to testy tworzone przy pomocy generatora liczb pseudolosowych, które w tym zadaniu pełnią też rolę testów poprawnościowych. Testy „b" to testy, które zawierają długi podciąg niemalejący, a testy „c" zawierają długi podciąg nierosnący — te grupy pełniły rolę testów wydajnościowych. Wielkości testów zostały tak dobrane, aby rozwiązania, które działają w złożoności pamięciowej $O(n)$ lub większej, nie dostawały więcej niż 50 punktów, niezależnie od złożoności obliczeniowej.

Nazwa	n	k	Opis
arc1a.in	30	23	test losowy
arc1b.in	14	6	test zawierający długi podciąg niemalejący
arc1c.in	16	9	test zawierający długi podciąg nierosnący
arc2a.in	14045	547	test losowy
arc2b.in	16456	769	test zawierający długi podciąg niemalejący
arc2c.in	28683	1984	test zawierający długi podciąg nierosnący
arc3a.in	154632	6428	test losowy
arc3b.in	276543	18632	test zawierający długi podciąg niemalejący
arc3c.in	597843	29567	test zawierający długi podciąg nierosnący
arc4a.in	1000000	12343	test losowy
arc4b.in	1500000	28754	test zawierający długi podciąg niemalejący
arc4c.in	2000000	32462	test zawierający długi podciąg nierosnący
arc5a.in	4000000	53421	test losowy
arc5b.in	5000000	86223	test zawierający długi podciąg niemalejący
arc5c.in	6000000	83456	test zawierający długi podciąg nierosnący
arc6a.in	8000000	76243	test losowy
arc6b.in	9000000	74437	test zawierający długi podciąg niemalejący
arc6c.in	10000000	138643	test zawierający długi podciąg nierosnący
arc7a.in	11000000	1976	test losowy
arc7b.in	11500000	18653	test zawierający długi podciąg niemalejący

Nazwa	n	k	Opis
arc7c.in	12 000 000	23 442	test zawierający długi podciąg nierosnący
arc8a.in	12 000 000	32 124	test losowy
arc8b.in	12 500 000	87 234	test zawierający długi podciąg niemalejący
arc8c.in	13 000 000	123 676	test zawierający długi podciąg nierosnący
arc9a.in	13 000 000	213 534	test losowy
arc9b.in	13 500 000	431 223	test zawierający długi podciąg niemalejący
arc9c.in	14 000 000	744 456	test zawierający długi podciąg nierosnący
arc10a.in	14 500 000	989 974	test losowy
arc10b.in	15 000 000	989 563	test zawierający długi podciąg niemalejący
arc10c.in	15 000 000	1 000 000	test zawierający długi podciąg nierosnący

Jakub Radoszewski — Treść zadania, Opracowanie
Juliusz Sompolski — Program

Dostępna pamięć: 32 MB. *OI, Etap II, dzień drugi, 12.02.2009*

Łyżwy

Bajtazar prowadzi klub łyżwiarski. Członkowie klubu spotykają się regularnie i razem trenują, przy czym korzystają zawsze z łyżew klubowych. Rozmiary łyżew są umownie numerowane od 1 do n. Każdy członek klubu ma pewien rozmiar stopy. U łyżwiarzy występuje jednakże współczynnik tolerancji d na rozmiar łyżew: łyżwiarz o rozmiarze stopy r może nosić łyżwy rozmiarów od r do r + d. Należy przy tym zaznaczyć, że ***żaden łyżwiarz nie zakłada nigdy jednocześnie dwóch łyżew różnych rozmiarów.***

Bajtazar zakupił na potrzeby klubu po k **par** *łyżew każdego z rozmiarów od 1 do n. W miarę upływu czasu nowe osoby zapisują się do klubu, a niektóre osoby wypisują się. Bajtazar martwi się, czy na każdych zajęciach będzie miał dla wszystkich członków klubu łyżwy odpowiedniego rozmiaru.*

Zakładamy, że początkowo nikt nie należy do klubu. Bajtazar dostarczył Ci sekwencję m zdarzeń postaci: przybyło/ubyło x członków klubu o rozmiarze stopy r. Bajtazar chciałby wiedzieć, po każdym takim zdarzeniu, czy ma łyżwy odpowiedniego rozmiaru dla wszystkich członków klubu. Poprosił Cię o napisanie programu, który to sprawdzi.

Wejście

Pierwszy wiersz standardowego wejścia zawiera cztery liczby całkowite n, m, k oraz d ($1 \leqslant n \leqslant 200\,000$, $1 \leqslant m \leqslant 500\,000$, $1 \leqslant k \leqslant 10^9$, $0 \leqslant d < n$)*, pooddzielane pojedynczymi odstępami i oznaczające odpowiednio: największy rozmiar łyżew, liczbę zdarzeń, liczbę par łyżew każdego rozmiaru zakupionych przez Bajtazara oraz tolerancję rozmiarową stóp łyżwiarzy. Kolejne m wierszy zawiera sekwencję m zdarzeń, po jednym w wierszu. Wiersz (i + 1)-szy (dla* $1 \leqslant i \leqslant m$*) zawiera dwie liczby całkowite:* r_i *oraz* x_i *(*$1 \leqslant r_i \leqslant n - d$, $-10^9 \leqslant x_i \leqslant 10^9$*), oddzielone pojedynczym odstępem. Jeśli* $x_i \geqslant 0$*, to oznacza to, że do klubu zapisało się* x_i *nowych członków o rozmiarze stopy* r_i*. Jeśli natomiast* $x_i < 0$*, to oznacza to, że z klubu wypisało się* $-x_i$ *członków o rozmiarze stopy* r_i*. Możesz założyć, że podana sekwencja zdarzeń ma sens, tzn. z klubu nie mogą wypisać się osoby, które się do niego nie zapisały.*

Wyjście

Twój program powinien wypisać na standardowe wyjście m wierszy. Wiersz i-ty (dla $1 \leqslant i \leqslant m$*) powinien zawierać jedno słowo* `TAK` *lub* `NIE`*, w zależności od tego, czy po i-tym zdarzeniu Bajtazar ma łyżwy odpowiedniego rozmiaru dla wszystkich członków klubu, czy też nie.*

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`4 4 2 1`	`TAK`
`1 3`	`TAK`
`2 3`	`NIE`
`3 3`	`TAK`
`2 -1`	

Po zajściu wszystkich zdarzeń z podanej sekwencji mamy trzech członków klubu, którzy mogą nosić łyżwy rozmiaru 1 lub 2, dwóch członków, którzy mogą nosić łyżwy rozmiaru 2 lub 3, oraz trzech, którzy mogą nosić łyżwy rozmiaru 3 lub 4. Przy takim składzie klubu rzeczywiście wystarczą po dwie pary łyżew rozmiarów 1, 2, 3 i 4:

- *dwie osoby dostają łyżwy rozmiaru 1;*
- *łyżwy rozmiaru 2 dostaje jedna osoba, która może nosić łyżwy rozmiaru 1 lub 2, i jedna, która może nosić łyżwy rozmiaru 2 lub 3;*
- *łyżwy rozmiaru 3 dostaje jedna osoba, która może nosić łyżwy rozmiaru 2 lub 3, i jedna, która może nosić łyżwy rozmiaru 3 lub 4;*
- *pozostałe dwie osoby dostają łyżwy rozmiaru 4.*

Rozwiązanie

Wersja statyczna

W przypadku problemów, w których mamy do czynienia z pewnym zbiorem obiektów zmieniających się w czasie i naszym zadaniem jest odpowiadanie na zapytania związane ze stanem tych obiektów w różnych momentach, zazwyczaj warto zadać sobie pytanie, jak należałoby odpowiadać na takie zapytania w przypadku obiektów, które nie zmieniają się w ogóle. Innymi słowy, możemy chcieć uprościć problem z wersji *dynamicznej*, czyli zmiennej w czasie, do *statycznej*. Spróbujemy zastosować to podejście do zadania o łyżwiarzach i łyżwach.

Ponieważ w zadaniu pojawia się stosunkowo dużo różnych danych i parametrów, nasz opis rozpoczniemy od krótkiego sformułowania problemu, który zamierzamy rozwiązać.

Problem w wersji statycznej.
Dane wejściowe:

- w problemie występują łyżwy o rozmiarach od 1 do n;
- łyżew rozmiaru j jest $s_j = k$ par;
- mamy t_1 łyżwiarzy o rozmiarze stopy 1, t_2 — o rozmiarze 2, itd., wreszcie t_{n-d} łyżwiarzy o rozmiarze stopy $n-d$;

- łyżwiarze z grupy t_i mogą nosić łyżwy rozmiarów z przedziału $[i, i+d]$.

Należy odpowiedzieć `TAK` lub `NIE`, w zależności od tego, czy można wszystkim łyżwiarzom przydzielić odpowiednie łyżwy.

Zauważmy, że jeżeli znajdziemy rozwiązanie problemu w wersji statycznej, to będziemy mogli je zastosować bezpośrednio w wersji dynamicznej przy każdym z m zapytań z osobna. Patrząc na ograniczenia z zadania, łatwo wywnioskować, że zapewne nie uzyskamy w ten sposób dostatecznie efektywnego rozwiązania, ale będziemy mieli w ogóle od czego zacząć myśleć dalej.

Rozwiązanie zachłanne

Okazuje się, że przy tak postawionym problemie, jeżeli istnieje żądane przyporządkowanie, to można je skonstruować, przydzielając łyżwy w sposób zachłanny. Dokładniej, analizujemy łyżwiarzy w kolejności $t_1, t_2, \ldots, t_{n-d}$ i każdemu z nich przypisujemy wolne łyżwy o najmniejszym możliwym rozmiarze.

Zanim zaczniemy się zastanawiać, jak efektywnie zaimplementować takie rozwiązanie (wszak łączne liczby łyżwiarzy i łyżew mogą być bardzo duże!), odpowiedzmy sobie na pytanie, dlaczego opisany algorytm zachłanny działa. Oczywiście można pominąć formalne dowodzenie i założyć, że „na pewno się nie pomyliliśmy", ale przy algorytmach zachłannych często warto być ostrożnym — przekonała się o tym m.in. liczna grupa zawodników, którzy w zadaniu *Konduktor* z tego samego etapu zaimplementowali „równie oczywiste" rozwiązanie zachłanne i skończyli z niedodatnią punktacją. A zatem:

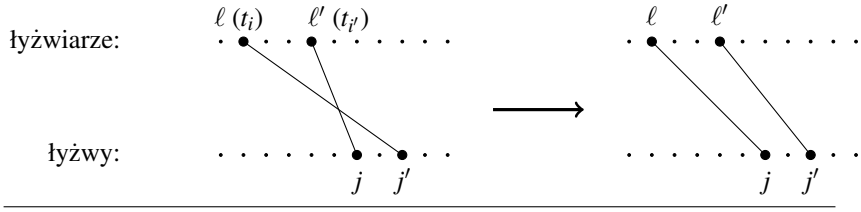
Twierdzenie 1. *Jeżeli problem w wersji statycznej posiada rozwiązanie, to opisany algorytm zachłanny znajdzie je.*

Dowód: Wystarczy pokazać, że jeżeli algorytm zachłanny udzieli odpowiedzi `NIE`, to w ogóle nie istnieje żadne poprawne przyporządkowanie łyżew łyżwiarzom (reguła kontrapozycji).

Dowód przeprowadzimy przez sprzeczność. Załóżmy, że algorytm zachłanny skonstruował jakieś przyporządkowanie cząstkowe C i następnie „zaciął" się na pewnym łyżwiarzu, któremu nie mógł już przypisać odpowiednich łyżew, istnieje natomiast pełne przyporządkowanie P, jednakże skonstruowane w jakiś inny sposób. Przyporządkowania C i P muszą się gdzieś różnić. Niech zatem i będzie najmniejszym takim rozmiarem stopy, że pewnemu łyżwiarzowi ℓ z grupy t_i w przyporządkowaniu C zostały przydzielone łyżwy rozmiaru j, natomiast w przyporządkowaniu P — o rozmiarze $j' \neq j$. Pokażemy, że możemy tak poprzestawiać przypisania w ramach P, żeby nie zmienić łyżew żadnego z wcześniejszych łyżwiarzy i przypisać łyżwiarzowi ℓ łyżwy rozmiaru j. Stąd uzyskamy żądaną sprzeczność, albowiem kontynuując tego typu przekształcanie przyporządkowania P, uzyskamy dokładnie taki przydział, jak w przyporządkowaniu C, co kłóci się z tym, że C nie da się już powiększyć o kolejne przypisanie.

Opiszemy teraz, jak wykonać zapowiadane przekształcenie. Zauważmy, że skoro C było konstruowane w sposób zachłanny, to $j < j'$. Zapytajmy się zatem, kto w przyporządkowaniu P jest właścicielem łyżew rozmiaru j. Ponieważ dla wszystkich wcześniejszych łyżwiarzy P i C są zgodne, więc w P jakieś łyżwy rozmiaru j są albo w ogóle nieprzypisane, albo znajdują się na nogach jakiegoś łyżwiarza ℓ' o rozmiarze stopy $i' \geqslant i$. W pierwszym z tych przypadków

opisane przekształcenie jest bardzo proste: zdejmujemy łyżwiarzowi ℓ jego aktualne łyżwy, a zakładamy wolne łyżwy rozmiaru j. W przeciwnym przypadku też nie będzie zbyt ciężko: wystarczy zamienić łyżwy łyżwiarzom ℓ i ℓ'. Jest to możliwe, gdyż przypisania łyżew tym łyżwiarzom „krzyżują się" (patrz rys. 1), tj. $i \leqslant i' \leqslant j \leqslant j'$. Skoro więc $j' \leqslant i + d$, to tym bardziej $j' \leqslant i' + d$ oraz $j \leqslant i + d$ i faktycznie zamiana łyżew jest możliwa. Jako że rozważyliśmy oba przypadki i w każdym wskazaliśmy żądane przekształcenie, to dowód jest zakończony. ■



Rys. 1: Zamiana łyżew (w ramach P) między łyżwiarzami ℓ oraz ℓ' o rozmiarze stopy odpowiednio i oraz i'.

Dodajmy, że powyższe uzasadnienie poprawności podejścia zachłannego jest dosyć typowe — warto zapamiętać tę metodę dowodzenia, gdyż może się ona przydać w przypadku innych zadań.

Implementacja rozwiązania zachłannego

Przyszła pora na zmierzenie się z implementacją algorytmu zachłannego. Ze względu na wspomniane już duże liczby łyżwiarzy i łyżew, przyporządkowań będziemy dokonywać „hurtowo", w jednym kroku obsługując wszystkich łyżwiarzy z danej grupy t_i.

```
1:  Algorytm zachłanny – implementacja 1
2:    for i := 1 to n − d do
3:    begin
4:      for j := i to i + d do
5:      begin
6:        g := min(t_i, s_j);
7:        t_i := t_i − g;
8:        s_j := s_j − g;
9:      end
10:     if t_i > 0 then return NIE;
11:   end
12:   return TAK;
```

Złożoność czasowa takiego rozwiązania to ewidentnie $O(nd)$. Okazuje się jednak, że w powyższej implementacji często nadrabiamy pracy, chociażby w sytuacji, gdy wielokrotnie próbujemy przydzielić łyżwiarzom łyżwy rozmiaru, który już dawno został wyczerpany. Można temu jednakże zaradzić. Zauważmy mianowicie, że w każdym kroku wewnętrznej pętli **for** jedna ze zmiennych t_i, s_j zostaje wyzerowana. Jeżeli jest to t_i, to możemy w ogóle wyjść z tej pętli. Jeżeli natomiast s_j, to wiemy, że łyżew rozmiaru j już nigdy więcej

nie musimy rozważać. Korzystając z tej obserwacji, możemy sprytniej zapisać powyższy pseudokod:

```
1:  Algorytm zachłanny – implementacja 2
2:    j := 1;
3:    for i := 1 to n − d do
4:    begin
5:      if j < i then j := i;
6:      while t_i > 0 and j ⩽ i + d do
7:      begin
8:        g := min(t_i, s_j);
9:        t_i := t_i − g;
10:       s_j := s_j − g;
11:       if s_j = 0 then j := j + 1;
12:     end
13:     if t_i > 0 then return NIE;
14:   end
15:   return TAK;
```

Jaka jest złożoność czasowa powyższego rozwiązania? Co prawda w pseudokodzie występują dwie zagnieżdżone pętle, jednak łączna liczba obrotów tych pętli jest niewielka, a to za sprawą tego, że w każdym kroku pętli **while** wartość zmiennej j wzrasta o jeden (jeśli zachodzi warunek w 11. linii), bądź też jest to ostatnie wykonanie wewnątrz danego obrotu pętli **for** (gdyż $t_i = 0$). To pokazuje, że pętla **while** wykona łącznie co najwyżej $2n$ obrotów, więc złożoność czasowa tej implementacji algorytmu to właśnie $O(n)$. Uzyskaliśmy zatem istotne przyspieszenie algorytmu za pomocą zaledwie drobnych modyfikacji pseudokodu.

Rozwiązania wersji dynamicznej problemu używające pierwszej z powyższych implementacji można znaleźć w plikach: `lyzs1.cpp`, `lyzs2.pas` i `lyzs3.java`. Mają one złożoność czasową $O(mnd)$ i zdobywały na zawodach 10% punktów. W przypadku drugiej z powyższych implementacji otrzymujemy rozwiązanie o złożoności $O(mn)$, które zdobywało na zawodach 30% punktów. Implementacje w plikach: `lyzs4.cpp`, `lyzs5.pas` i `lyzs6.java`.

Problem w wersji dynamicznej można rozwiązać efektywniej, ale nie tędy droga. Otóż niestety rozwiązanie zachłanne nie bardzo nadaje się do dalszych ulepszeń. W następnej sekcji zajmiemy się opisem innego rozwiązania problemu statycznego, co prawda o tej samej złożoności czasowej, ale efektywniejszego pod kątem wersji dynamicznej. Na szczęście praca nad rozwiązaniem zachłannym nie pójdzie na marne: fakt, że jest ono poprawne (Twierdzenie 1), przyda nam się do dowodu poprawności tego nowego rozwiązania.

Inne rozwiązanie wersji statycznej

Tym razem do rozwiązania podejdziemy zupełnie inaczej: wyznaczymy pewne zwięźle formułowalne kryterium na to, kiedy da się przydzielić wszystkim łyżwiarzom pasujące łyżwy, a kiedy nie.

Przyjrzyjmy się łyżwiarzom o rozmiarach stopy z przedziału $[a,b]$. Jasne jest, że wszystkie rozmiary łyżew, które mają szansę pasować tym łyżwiarzom, należą do przedziału $[a,b+d]$. Aby istniała w ogóle możliwość odpowiedzi pozytywnej w naszym problemie,

liczba par łyżew o rozmiarach z przedziału $[a, b+d]$ musi być nie mniejsza niż łączna liczba łyżwiarzy $t_a + t_{a+1} + \ldots + t_b$. Jest to zatem pewien warunek konieczny istnienia poprawnego przyporządkowania łyżew. Co ciekawe, warunek ten zapisany w odpowiedni sposób staje się także warunkiem dostatecznym! Mówi o tym następujące twierdzenie.

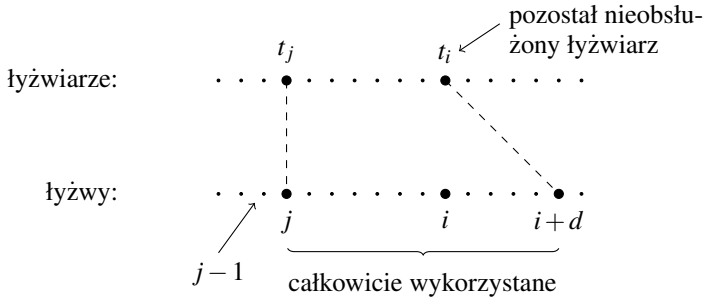
Twierdzenie 2. *Przydział łyżew jest możliwy wtedy i tylko wtedy, gdy dla każdego przedziału* $[a,b]$*, takiego że* $1 \leqslant a \leqslant b \leqslant n-d$*, zachodzi*

$$t_a + t_{a+1} + \ldots + t_b \leqslant (b+d-a+1) \cdot k. \tag{1}$$

Dowód: Uzasadniliśmy już, że każdy z warunków (1) z osobna stanowi warunek konieczny istnienia przydziału łyżew. Ograniczymy się zatem do pokazania dostateczności układu wszystkich warunków (1), tzn. tego, że jeżeli wszystkie warunki zachodzą, to istnieje żądane przyporządkowanie łyżew łyżwiarzom. To sformułowanie jest z kolei równoważne temu, że jeżeli takie przyporządkowanie nie istnieje, to któryś z warunków (1) nie zachodzi.

Załóżmy zatem, że nie istnieje sposób przydziału łyżwiarzom odpowiednich łyżew. Na mocy Twierdzenia 1 oznacza to, że dla takich danych wejściowych algorytm zachłanny nie znajdzie pełnego przyporządkowania. Korzystając z tego spostrzeżenia, wskażemy przedział $[a,b]$, dla którego kryterium (1) nie zachodzi.

Porażka algorytmu zachłannego polega na tym, że pewnemu łyżwiarzowi z grupy t_i nie udało się przyporządkować żadnych łyżew, gdyż wszystkie łyżwy o rozmiarach od i do $i+d$ zostały już wcześniej wykorzystane. Oznaczmy przez C znalezione przyporządkowanie częściowe. Niech j będzie najmniejszym rozmiarem łyżew takim, że wszystkie łyżwy o rozmiarach $j, j+1, \ldots, i+d$ zostały wykorzystane przy konstruowaniu C. Wykażemy, że szukanym przedziałem $[a,b]$ jest przedział $[j,i]$ (patrz rys. 2).



Rys. 2: W przyporządkowaniu częściowym C łyżwy o rozmiarach z przedziału $[j, i+d]$ zostały całkowicie wykorzystane (po k par), natomiast łyżwy rozmiaru $j-1$ już nie. To oznacza, że tylko łyżwiarze z grup $t_j, \ldots, t_i$ mają na nogach łyżwy o rozmiarach z przedziału $[j, i+d]$ i, co więcej, jeszcze tych łyżew dla nich zabrakło.

W tym celu wystarczy stwierdzić, że żaden z łyżwiarzy z grup $t_1, t_2, \ldots, t_{j-1}$ nie dostał w ramach C łyżew rozmiaru j lub większego. Jeżeli $j = 1$, to to zdanie jest oczywiście prawdziwe. W przeciwnym przypadku na mocy definicji j wiemy, że w ramach C pozostały niewykorzystane jakieś łyżwy rozmiaru $j-1$. A ponieważ C zostało skonstruowane w sposób zachłanny, więc żadnemu łyżwiarzowi z grupy t_q, $q < j$, nie mogły zostać przydzielone łyżwy

o rozmiarze nie mniejszym niż j, gdyż wcześniej otrzymałby jedną z wolnych par łyżew rozmiaru $j-1$.

Z dotychczasowych spostrzeżeń wynika, że wszystkie $(i+d-j+1)\cdot k$ łyżew o rozmiarach od j do $i+d$ zostało wykorzystanych oraz że łyżwy te zostały rozdzielone wśród $t_j+t_{j+1}+\ldots+t_i$ łyżwiarzy o rozmiarach stopy między j a i, i na dodatek jeszcze ich zabrakło. To pokazuje, że rzeczywiście kryterium nie zachodzi w przypadku $a=j$, $b=i$. ■

Ktoś mógłby zapytać, skąd właściwie wzięło się takie „magiczne" kryterium istnienia przyporządkowania i w jaki sposób można na nie wpaść. Jednym ze sposobów jest na pewno metoda prób i błędów — zauważamy konieczność układu warunków (1) i mimo usilnych starań nie jesteśmy w stanie wymyślić kontrprzykładu na ich dostateczność. Okazuje się, że istnieje także bardziej metodyczne podejście do problemu, które pozwala wywnioskować Twierdzenie 2 jako szczególny przypadek pewnej własności grafowej. Co ciekawe, sprowadzenie to zawdzięczamy samym zawodnikom — dziwnym trafem żaden z jurorów nie wykrył go przed zawodami.

Zauważmy mianowicie, że łyżwy i łyżwiarze tworzą pewien graf dwudzielny $G=(V_1,V_2,E)$, o czym zresztą mogły nas już przekonać dotychczasowe rysunki w opisie rozwiązania. Łyżwiarz $v_1\in V_1$ jest w nim połączony krawędzią z parą łyżew $v_2\in V_2$ wtedy i tylko wtedy, gdy może w ramach swojego przedziału tolerancji założyć te łyżwy. Szukane przyporządkowanie łyżew jest wówczas *skojarzeniem* w grafie G (czyli takim podzbiorem krawędzi, z których żadne dwie nie są incydentne), w którym *każdy* łyżwiarz jest skojarzony z jakąś parą łyżew. Kryterium istnienia takiego skojarzenia stanowi klasyczne Twierdzenie Halla (patrz [32]):

Twierdzenie 3 (Hall). *W grafie dwudzielnym $G=(V_1,V_2,E)$ istnieje skojarzenie pokrywające wszystkie wierzchołki ze zbioru V_1 wtedy i tylko wtedy, gdy dla każdego podzbioru $W_1\subseteq V_1$ moc zbioru wierzchołków sąsiadujących z jakimikolwiek wierzchołkami z W_1:*

$$W_2=\{v_2\in V_2:\ \exists_{v_1\in W_1}\ (v_1,v_2)\in E\}$$

jest nie mniejsza od mocy W_1, czyli $|W_2|\geqslant|W_1|$.

Aby z Twierdzenia Halla uzyskać Twierdzenie 2, wystarczy udowodnić:

Lemat 1. W przypadku grafu łyżwiarzy i łyżew warunek istnienia skojarzenia z Twierdzenia Halla jest równoważny warunkowi (1).

Dowód: Zauważmy, że warunek (1), jeżeli wyrazić go w języku teorii grafów, stanowi szczególny przypadek warunku z twierdzenia Halla, w którym zbiór W_1 składa się ze wszystkich łyżwiarzy o rozmiarach stopy od a do b. Teza lematu w jedną stronę jest więc oczywista.

W drugą stronę musimy pokazać, że warunek (1) implikuje warunek z Twierdzenia Halla. W tym celu wystarczy udowodnić, że jeżeli warunek $|W_2|\geqslant|W_1|$ nie zachodzi dla pewnego $W_1\subseteq V_1$, to istnieje także jakiś zbiór W_1' składający się ze wszystkich łyżwiarzy o rozmiarach stopy z pewnego przedziału $[a,b]$, dla którego nie zachodzi analogiczny warunek $|W_2'|\geqslant|W_1'|$.

Niech W_1 będzie najmniej licznym zbiorem, dla którego warunek z twierdzenia Halla nie zachodzi. Zauważmy, że wówczas nie może istnieć w W_1 para kolejnych pod względem rozmiaru stopy łyżwiarzy, których rozmiary stopy różnią się o więcej niż d, gdyż wówczas

wyznaczaliby oni podział W_1 na dwa mniejsze zbiory, z których dla co najmniej jednego nie zachodziłoby kryterium Halla. Niech l i u oznaczają odpowiednio najmniejszy i największy rozmiar stopy łyżwiarza w zbiorze W_1. Wówczas W_2 składa się ze wszystkich dostępnych par łyżew o rozmiarach z przedziału $[l, u+d]$. Ponadto, dołożenie do W_1 wszystkich łyżwiarzy o rozmiarach stopy od l do u nie zmienia postaci odpowiadającego zbioru W_2. W wyniku takiego powiększenia W_1 uzyskujemy szukany zbiór W_1' odpowiadający przedziałowi rozmiarów stopy $[a,b] = [l,u]$. ■

W ten sposób uzyskaliśmy alternatywne wyprowadzenie kryterium z Twierdzenia 2. Godne podkreślenia jest, że wykorzystaliśmy do tego celu twierdzenie, które na pierwszy rzut oka wydaje się być zupełnie bezużyteczne w praktyce — istnieją wszakże wielomianowe algorytmy znajdowania najliczniejszego skojarzenia w grafie dwudzielnym (patrz np. [20]), a przecież Twierdzenie Halla sprowadza problem skojarzenia do sprawdzenia wykładniczej liczby warunków! Siła tego twierdzenia ujawniła się w tym przypadku w związku ze szczególną postacią naszego grafu dwudzielnego — podobne przykłady można znaleźć w literaturze (patrz np. [32]).

Podciąg spójny o maksymalnej sumie

Aby skutecznie wykorzystać kryterium z Twierdzenia 2, trzeba trochę je przeformułować. Warunek (1) możemy zapisać równoważnie tak:

$$(t_a - k) + (t_{a+1} - k) + \ldots + (t_b - k) \leqslant d \cdot k.$$

Wprowadzając nowe zmienne $t'_1, t'_2, \ldots, t'_{n-d}$, tak aby zachodziła równość $t'_i = t_i - k$, otrzymujemy następujący zapis wyprowadzonego kryterium:

Twierdzenie 4. *Przydział łyżew jest możliwy wtedy i tylko wtedy, gdy dla każdego przedziału $[a,b]$, takiego że $1 \leqslant a \leqslant b \leqslant n-d$, zachodzi*

$$t'_a + t'_{a+1} + \ldots + t'_b \leqslant d \cdot k. \tag{2}$$

Jaką korzyść uzyskaliśmy z takiego przekształcenia? Zauważmy jedną istotną rzecz: po prawej stronie każdego z warunków (2) występuje wyrażenie niezależne od parametrów a i b, to znaczy *stałe* dla ustalonych danych wejściowych. Skoro tak, to możemy podejść do sprawdzenia warunku z nowego kryterium w trochę nietypowy sposób. Otóż wystarczy znaleźć takie a i b, dla których wartość wyrażenia $t'_a + t'_{a+1} + \ldots + t'_b$ jest *największa*, i sprawdzić, czy ta właśnie wartość przekracza $d \cdot k$, czy nie.

W ten sposób wyjściowy problem został sprowadzony do klasycznego problemu podciągu spójnego ciągu t'_i o maksymalnej sumie[1]. Znane są co najmniej dwa algorytmy rozwiązujące ten problem (patrz [19]) w złożoności czasowej liniowej względem długości ciągu, czyli w tym przypadku $O(n)$. Implementacje rozwiązania problemu dynamicznego opartego o tę metodę można znaleźć w plikach `lyzs7.cpp`, `lyzs8.pas` i `lyzs9.java`. Mają one złożoność czasową $O(mn)$ i zdobywały na zawodach 30% punktów.

[1] Warto zaznaczyć, że elementy ciągu t'_i, w przeciwieństwie do t_i, nie muszą być nieujemne.

Wersja dynamiczna

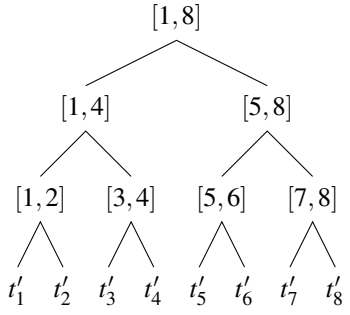
Podobnie jak w przypadku statycznym, na początku sformułujemy dokładnie problem, który chcemy rozwiązać. Zauważmy, że dzięki sprowadzeniu z poprzedniej sekcji możemy pominąć wiele parametrów problemu i opisać go abstrakcyjnie. W dalszym opisie będziemy stosowali skrótowe oznaczenie *psoms* = podciąg spójny o maksymalnej sumie.

Dynamiczny problem psoms.
Dany jest ciąg $t'_i: \ i = 1,2,\ldots,n-d$, początkowo $t'_i \equiv -k$. Należy zaprojektować strukturę danych, która udostępnia następujące operacje:

- `modyfikuj`(i, x): zwiększenie t'_i o stałą x (x może też być ujemne);
- `zapytanie`: wyznaczenie sumy psoms w ciągu t'_i.

Naszym celem jest jak najefektywniejsze przeprowadzenie inicjalizacji tej struktury (operacja `init`) oraz zasymulowanie m par podanych operacji.

Poszukiwaną strukturą danych może być na przykład statyczne drzewo przedziałowe[2], będące pełnym drzewem binarnym, w którego liściach umieszczone są elementy ciągu t'_i, a węzły odpowiadają przedziałom indeksów tego ciągu. Dla uproszczenia zakładamy, że liczba elementów ciągu t'_i jest całkowitą potęgą dwójki (jeżeli tak nie jest, to możemy zawsze dołożyć jakieś sztuczne elementy o wartości 0). Przykład takiego drzewa można znaleźć na rys. 3.



Rys. 3: Przykład drzewa przedziałowego dla ośmioelementowego ciągu t'_i.

Podstawowe własności drzewa przedziałowego, które będą nam potrzebne, to:

- liczba węzłów drzewa jest liniowa względem długości ciągu, a głębokość drzewa — logarytmiczna;
- ponumerowane w porządku kopcowym drzewo możemy reprezentować w zwykłej tablicy;

[2]Więcej o statycznych drzewach przedziałowych można przeczytać np. w opisie rozwiązania zadania Tetris 3D z książeczki XIII Olimpiady Informatycznej [13], albo poczytać i posłuchać na stronie `http://was.zaa.mimuw.edu.pl`

- przedziały zawarte w dzieciach węzła odpowiadają połowicznemu podziałowi przedziału tego węzła pod względem zbioru liczb całkowitych w nim zawartych;
- wszystkie przedziały zawierające dany indeks i stanowią ścieżkę od liścia odpowiadającego t'_i do korzenia.

Aby rozwiązać dynamiczny problem psoms, wzbogacimy drzewo przedziałowe, umieszczając we wszystkich węzłach pewne dodatkowe wartości, które pomogą nam efektywnie odpowiadać na zapytania. Zaczniemy od wprowadzenia pola `max_psoms`, w którym będziemy przechowywać maksymalną sumę psoms podciągu odpowiadającego danemu węzłowi (jeżeli wszystkie wyrazy tego podciągu są ujemne, to prawidłową wartością tego pola będzie 0). Jeżeli będziemy umieli utrzymywać te wartości, symulując operację `modyfikuj`, to obsługa zapytania będzie polegała zaledwie na odczytaniu wartości `max_psoms` z korzenia drzewa.

Aby dało się aktualizować wartości opisanego parametru podczas operacji modyfikacji, wystarczy zadbać o to, aby wartość `max_psoms` w węźle p zależała jedynie od wartości dla synów l i r tego węzła oraz ewentualnie pewnych dodatkowych pól, które jeszcze trzeba będzie wprowadzić. Jeżeli uda nam się to zagwarantować, to modyfikacja będzie polegała na zmianie wartości t'_i w odpowiednim liściu oraz na poprawieniu wartości pól na ścieżce od tego liścia do korzenia drzewa.

Jak zatem obliczać wartość `max_psoms`(p)? Na pewno zależy ona od maksimum z wartości `max_psoms`(l) oraz `max_psoms`(r) — dotyczy to przypadku, kiedy psoms w p jest w całości zawarty w jednej z połówek przedziału — ale jeżeli psoms zawiera zarówno pewne elementy z l, jak i z r, to do wyznaczenia tej wartości potrzebne jest coś jeszcze. W tym przypadku psoms dla p składa się z pewnego *sufiksu* (tj. końcowego fragmentu) podciągu odpowiadającego l oraz z pewnego *prefiksu* (tj. początkowego fragmentu) podciągu r. A jakiego sufiksu i prefiksu? Oczywiście tych o największych sumach!

W ten sposób wywnioskowaliśmy potrzebę wprowadzenia pól `max_pref` oraz `max_suf` dla węzłów. Po chwili namysłu można dalej wywnioskować, że z kolei do wyznaczania tych wartości potrzebne jest jeszcze pole `suma`, oznaczające sumę wszystkich elementów podciągu odpowiadającego danemu węzłowi drzewa.

Podsumujmy ten luźny wywód dokładnymi definicjami wprowadzonych pól dla węzła odpowiadającego przedziałowi $[a,b]$:

- `max_psoms`: suma psoms ciągu $t'_a, t'_{a+1}, \ldots, t'_b$;
- `max_pref`: maksymalna z sum prefiksowych postaci $t'_a + t'_{a+1} + \ldots + t'_i$ dla $i \leqslant b$;
- `max_suf`: maksymalna z sum sufiksowych postaci $t'_i + t'_{i+1} + \ldots + t'_b$ dla $i \geqslant a$;
- `suma`: $t'_a + t'_{a+1} + \ldots + t'_b$;

oraz odpowiadającymi im wzorami rekurencyjnymi:

$$\begin{aligned}
\texttt{max_psoms}(p) &= \max(\texttt{max_psoms}(l),\ \texttt{max_psoms}(r), \\
&\qquad \texttt{max_suf}(l) + \texttt{max_pref}(r)) && (3)\\
\texttt{max_pref}(p) &= \max(\texttt{max_pref}(l),\ \texttt{suma}(l) + \texttt{max_pref}(r)) && (4)\\
\texttt{max_suf}(p) &= \max(\texttt{max_suf}(r),\ \texttt{max_suf}(l) + \texttt{suma}(r)) && (5)\\
\texttt{suma}(p) &= \texttt{suma}(l) + \texttt{suma}(r) && (6)
\end{aligned}$$

Dodajmy, że w liściu l odpowiadającemu elementowi t'_i mamy:

$$\texttt{max\_psoms}(l) = \texttt{max\_pref}(l) = \texttt{max\_suf}(l) = \max(t'_i, 0) \tag{7}$$

$$\texttt{suma}(l) = t'_i \tag{8}$$

Za pomocą podanych wzorów możemy już zaimplementować żądane operacje:

- `init`: za pomocą (7) i (8) inicjujemy wartości dla liści, po czym za pomocą wzorów (3)–(6) dla kolejnych węzłów wewnętrznych (warstwami od dołu). Złożoność czasowa $O(n)$.
- `modyfikuj`(i, x): korzystając ze wzorów (7) i (8), ustawiamy wartości w liściu l odpowiadającemu t'_i, po czym aktualizujemy wszystkie wartości na ścieżce od l do korzenia (wzory (3)–(6)). Złożoność czasowa pojedynczej operacji: $O(\log n)$.
- `zapytanie`: zwracamy `max_psoms` dla korzenia (czas $O(1)$).

Całkowity koszt czasowy inicjalizacji i wykonania m zapytań to $O(n + m \log n)$. Opisane rozwiązanie wzorcowe zostało zaimplementowane w plikach `lyz.cpp`, `lyz1.pas` oraz `lyz2.java`.

Testy

Rozwiązania zawodników były sprawdzane na 10 zestawach danych testowych. Sposób ich generowania był jednolity i w znacznej mierze losowy. Polegał on na powtarzaniu następujących czynności:

- wylosowanie przedziału $[a,b]$ rozmiarów stopy;
- dodawanie do klubu łyżwiarzy, których rozmiar stopy należy do przedziału $[a,b]$, aż do momentu, gdy nie można przyporządkować łyżew wszystkim łyżwiarzom (liczby dodawanych łyżwiarzy były tak dobierane, aby Bajtazar musiał korzystać z możliwie szerokiego zakresu rozmiarów łyżew przy rozdawaniu ich łyżwiarzom);
- usuwanie z klubu łyżwiarzy z przedziału rozmiarów stopy, który powoduje niezachodzenie kryterium (2);

do czasu, aż wykorzystany zostanie limit operacji. Poniższa tabelka zawiera parametry kolejnych testów, w kolejności: liczbę rozmiarów stopy, liczbę operacji, zapas łyżew każdego rozmiaru oraz przedział tolerancji stóp łyżwiarzy.

Nazwa	n	m	k	d	Opis
lyz1.in	500	1 000	12	10	mały test losowy, małe d
lyz2.in	5 000	10 000	54	2 000	mały test losowy, duże d
lyz3.in	5 000	10 000	72	2 000	mały test losowy, duże d
lyz4.in	15 000	30 000	6 740	70	średni test losowy

Nazwa	n	m	k	d	Opis
lyz5.in	100 000	250 000	57 434	30	duży test losowy
lyz6.in	100 000	250 000	23 522	40	duży test losowy
lyz7.in	200 000	500 000	1 012	50	duży test losowy
lyz8.in	200 000	500 000	512 300 012	21 212	duży test losowy
lyz9.in	200 000	500 000	999 999 997	23 500	duży test losowy
lyz10a.in	200 000	500 000	1 000 000 000	50 000	duży test losowy
lyz10b.in	200 000	500 000	1 000 000 000	0	duży test losowy z zerową tolerancją

Zadanie dodatkowe

Jeżeli spodobało Ci się, drogi Czytelniku, zadanie o łyżwiarzach i łyżwach, to możesz zechcieć zmierzyć się z innym, nieco podobnym problemem. Dane są w nim górne ograniczenia $a_1, a_2, \ldots, a_n$ na wartości elementów permutacji zbioru $\{1, 2, \ldots, n\}$ i należy odpowiadać na zapytania o to, czy istnieje jakakolwiek permutacja spełniająca te ograniczenia. Dodatkowo, ograniczenia te zmieniają się (pojedynczo) w czasie. Zainteresowanych Czytelników odsyłamy do zadania *Permutacja* z piątej rundy Potyczek Algorytmicznych 2009.

Zawody III stopnia

opracowania zadań

Michał Włodarczyk
Treść zadania, Opracowanie

Mirosław Michalski
Program

Dostępna pamięć: 32 MB.

OI, Etap III, dzień próbny, 31.03.2009

Wiedźmak

Bajtazar został wiedźmakiem — pogromcą potworów. Przyszło mu wracać do swojego rodzimego Bajtogrodu przez krainę pełną potworów. Szczęśliwie, ludzie zamieszkujący tę krainę, walczący od pokoleń z potworami, opanowali sztukę wytwarzania specjalistycznych mieczy pomocnych w konfrontacji z nimi. W krainie, przez którą wędruje Bajtazar, znajduje się wiele miejscowości i łączących je dróg. Drogi nie krzyżują się poza miejscowościami, choć niektóre z nich prowadzą nie tylko po ziemi, ale i przez podziemia.

Bajtazar zebrał informacje o tej krainie (jako wiedźmak lubi wiedzieć różne rzeczy). Wie, jakiego rodzaju potwory można spotkać na każdej drodze i ile czasu potrzeba na jej przebycie. Wie też, w których miejscowościach można znaleźć kowali i na jakie rodzaje potworów potrafią oni wykuwać miecze. W tej chwili nie posiada żadnych mieczy i chce jak najszybciej dostać się do Bajtogrodu. Wstyd się przyznać, ale choć jest wiedźmakiem, to nie wie, jak to zrobić. Pomóż mu i znajdź taką trasę, która jak najszybciej doprowadzi go do Bajtogrodu i na której, zawsze gdy może spotkać potwory danego rodzaju, ma wcześniej możliwość zaopatrzenia się w miecz odpowiedni do walki z nimi. Bajtazar jest bardzo silny i może nosić ze sobą dowolnie wiele mieczy.

Wejście

Pierwszy wiersz standardowego wejścia zawiera cztery liczby całkowite: n, m, p i k ($1 \leqslant n \leqslant 200$, $0 \leqslant m \leqslant 3\,000$, $1 \leqslant p \leqslant 13$, $0 \leqslant k \leqslant n$) pooddzielane pojedynczymi odstępami, oznaczające odpowiednio: liczbę miejscowości, liczbę łączących je dróg, liczbę gatunków potworów i liczbę kowali. Miejscowości są ponumerowane od 1 do n, przy czym Bajtogród ma numer n, a miejscowość, z której wyrusza Bajtazar — numer 1. Rodzaje potworów są ponumerowane od 1 do p.

W kolejnych k wierszach opisani są kowale, po jednym w każdym z tych wierszy. Wiersz $(i+1)$-szy zawiera liczby całkowite w_i, q_i, $r_{i,1} < r_{i,2} < \ldots < r_{i,q_i}$ ($1 \leqslant w_i \leqslant n$, $1 \leqslant q_i \leqslant p$, $1 \leqslant r_{i,j} \leqslant p$) pooddzielane pojedynczymi odstępami, oznaczające odpowiednio: numer miejscowości, w której mieszka kowal, liczbę rodzajów potworów, przeciwko którym potrafi on wykuć miecze, oraz rodzaje tych potworów (w kolejności rosnącej). Kowale mogą mieszkać w tych samych miejscowościach.

W kolejnych m wierszach opisane są drogi. Wiersz $(k+i+1)$-szy zawiera liczby całkowite v_i, w_i, t_i, s_i, $u_{i,1} < u_{i,2} < \ldots < u_{i,s_i}$ ($1 \leqslant v_i < w_i \leqslant n$, $1 \leqslant t_i \leqslant 500$, $0 \leqslant s_i \leqslant p$, $1 \leqslant u_{i,j} \leqslant p$) pooddzielane pojedynczymi odstępami, oznaczające odpowiednio: miejscowości, które łączy droga, czas potrzebny na przebycie drogi (jest on taki sam w każdym kierunku), liczbę rodzajów potworów, które można spotkać na danej drodze, oraz rodzaje tych potworów (w kolejności rosnącej). Żadne dwie drogi nie łączą tej samej pary miejscowości.

Wyjście

Twój program powinien wypisać na standardowe wyjście jedną liczbę całkowitą — minimalny łączny czas potrzebny na dotarcie do Bajtogrodu. W przypadku, gdy dotarcie do Bajtogrodu jest niemożliwe, należy wypisać −1.

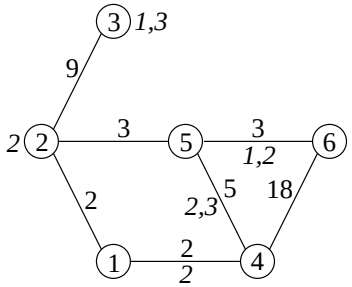
Przykład

Dla danych wejściowych:

```
6 7 4 2
2 1 2
3 2 1 3
1 2 2 0
2 3 9 0
1 4 2 1 2
2 5 3 0
4 5 5 2 2 3
4 6 18 0
5 6 3 2 1 2
```

poprawnym wynikiem jest:

```
24
```



Natomiast dla danych:

```
2 1 1 1
2 1 1
1 2 1 1 1
```

poprawnym wynikiem jest:

```
-1
```

Komentarz do przykładu

W pierwszym przykładzie na rysunku kółka reprezentują miejscowości, a liczby w ich wnętrzu to ich numery. Krawędzie reprezentują drogi, a liczby umieszczone nad nimi — czas potrzebny na ich przebycie. Liczby znajdujące się obok kółek (pisane kursywą) oznaczają rodzaje potworów, przeciwko którym kowal z danej miejscowości potrafi wykuć miecze. Liczby pod krawędziami (pisane kursywą) oznaczają rodzaje potworów, które można spotkać na danej drodze.

Wiedźmak Bajtazar powinien najpierw pójść do miejscowości nr 2, zdobyć miecz przeciwko potworom nr 2, wrócić do miejscowości nr 1, potem do miejscowości nr 4 i w końcu do Bajtogrodu.

W drugim przykładzie Bajtazar nie jest w stanie zdobyć miecza na potwora nr 1, więc nie może też dojść do Bajtogrodu.

Rozwiązanie

Analiza problemu

Jak każdy Czytelnik już z pewnością zauważył, nasze zadanie ma naturę grafową. Rozważmy graf nieskierowany G, w którym miejscowości są wierzchołkami, a krawędzie to drogi pomiędzy nimi, przy czym jako wagę krawędzi przyjmujemy czas potrzebny na przebycie odpowiadającej jej drogi. W takim grafie mamy znaleźć najkrótszą bezpieczną ścieżkę między pewnymi dwoma wierzchołkami. Ścieżka będzie *bezpieczna*, jeśli zawsze przed wejściem na drogę zbierzemy komplet mieczy pozwalający rozprawić się z występującymi na niej potworami.

Na początek spróbujmy uprościć, co tylko się da. Zauważmy, że:

- nie warto pamiętać, którzy kowale wykuwają które miecze — w zupełności wystarczy nam informacja, jakie rodzaje mieczy można zdobyć w poszczególnych miejscowościach;
- jeżeli Bajtazar znajduje się w jakiejś miejscowości, to na pewno dobrym pomysłem jest dla niego zebranie od razu wszystkich dostępnych w niej rodzajów mieczy;
- jako że wszystkie miecze przeznaczone do walki z danym rodzajem potworów są dla Bajtazara zupełnie jednakowe, w dowolnym momencie wędrówki liczy się dla niego tylko informacja, jakie rodzaje potworów może on pokonać za pomocą posiadanych mieczy.

Dalej, oznaczmy przez $M(v)$ zbiór mieczy dostępnych w miejscowości v, natomiast przez $P(e)$ — zbiór mieczy potrzebnych do bezpiecznego pokonania drogi e.

Rozwiązanie wzorcowe

Wejdźmy na chwilę w rolę Bajtazara — może pomoże nam to wymyślić dobrą strategię rozwiązania problemu. Załóżmy, że doszliśmy właśnie do miejscowości v i mamy w plecaku zbiór mieczy S. Zgodnie z dotychczasowymi spostrzeżeniami, w tym momencie opłaca nam się zebrać wszystkie miecze dostępne w v. Jeśli teraz e jest pewną drogą wychodzącą z v, to możemy nią wyruszyć, jeżeli

$$P(e) \subseteq S \cup M(v).$$

Oznaczmy przez $t(v,S)$ minimalny czas potrzebny, by znaleźć się w miejscowości v, posiadając zbiór mieczy S — łącznie z mieczami wykuwanymi w v. Jeżeli sytuacja określana przez parę (v,S) nie jest możliwa, to przyjmujemy, że $t(v,S) = \infty$. Oczywiście $t(1,\ M(1)) = 0$, bo na samym początku wędrówki Bajtazar może od razu zdobyć wszystkie miecze z miejscowości nr 1.

Zauważmy teraz, że jeśli $t(v,S) = c \neq \infty$ i w grafie G istnieje krawędź (v,w), to wiedźmakowi wystarczy informacja, czy $P\big((v,w)\big) \subseteq S$. Jeśli tak, to może przejść do miejscowości w i od razu zebrać wszystkie nowe miecze, jakie tam znajdzie. Niech c' oznacza czas potrzebny na przebycie drogi (v,w). Wtedy wiedźmak może znaleźć się w miejscowości

w z mieczami $S \cup M(w)$ w czasie $c + c'$. Nie wiemy jednak, czy jest to najszybszy sposób, więc otrzymujemy nierówność

$$t(w,\ S \cup M(w)) \leqslant c + c'.$$

Bardzo podobną nierówność otrzymalibyśmy, próbując wymyślić algorytm znajdowania najkrótszej ścieżki w grafie G. Okazuje się, że zbieżność ta jest nieprzypadkowa, tyle że nasza nierówność odpowiada szukaniu najkrótszej ścieżki w nieco innym grafie...

Parę $[v,S]$ nazwijmy *stanem*. Skonstruujmy teraz nowy graf H, którego wierzchołkami są stany, zaś zbiór krawędzi definiujemy jak wyżej. Formalnie,

$$\big([v,A],[w,B]\big) \in E(H) \quad \Longleftrightarrow \quad (v,w) \in E(G) \ \wedge \ P\big((v,w)\big) \subseteq A \ \wedge \ B = A \cup M(w), \quad (1)$$

przy czym $E(H)$ i $E(G)$ oznaczają zbiory krawędzi odpowiednio grafów H i G, natomiast wagi pozostają takie jak oryginalnych krawędzi. Zauważmy, że H jest grafem skierowanym. Zadanie sprowadza się teraz do znalezienia najkrótszej ścieżki w grafie H z wierzchołka $[1,M(1)]$ do dowolnego wierzchołka postaci $[n,X]$, bo nieważne jest, z jakimi mieczami dotrzemy do Bajtogrodu. Powszechnie znanym i efektywnym algorytmem rozwiązującym ten problem dla grafu o nieujemnych wagach jest algorytm Dijkstry. Czytelników niezaznajomionych z tym algorytmem odsyłamy do [18] czy [20], a praktycznie do dowolnej książki o algorytmach. Algorytm Dijkstry korzysta ze struktury danych zwanej kolejką priorytetową lub kopcem, której opis również można znaleźć w każdej z wymienionych książek.

Miecze i maski

Dotychczas pisaliśmy dużo o zbiorach i relacjach między nimi, nie zastanawiając się wcale nad implementacją rozwiązania na komputerze. Chcemy reprezentować zbiory w taki sposób, by można było efektywnie wykonywać na nich następujące operacje:

- badanie równości zbiorów,
- obliczanie sumy zbiorów,
- stwierdzanie, czy zbiór jest podzbiorem drugiego.

Okazuje się, że jeśli umiemy wykonać dwie pierwsze operacje, to umiemy też trzecią. Wynika to z poniższej tożsamości, której dowód pozostawiamy jako proste ćwiczenie:

$$A \subseteq B \Leftrightarrow A \cup B = B.$$

W tym miejscu w istotny sposób skorzystamy z ograniczenia $p \leqslant 13$. Otóż skojarzymy miecze z liczbami w systemie dwójkowym w następujący sposób:

$$\begin{aligned}
\text{pierwszy miecz} &\equiv [0000000000001]_2 &&= 1\\
\text{drugi miecz} &\equiv [0000000000010]_2 &&= 2\\
&\dots\\
\text{piąty miecz} &\equiv [0000000010000]_2 &&= 16\\
&\dots
\end{aligned}$$

Formalnie, i-temu mieczowi przyporządkowujemy liczbę 2^{i-1}. Zbiór traktujemy jako sumę reprezentacji jego elementów. Na przykład, zbiór złożony z mieczy $1, 2, 5$ to $[0000000010011]_2 = 19$. Jak widać, reprezentacja zbioru o zadanych elementach jest jednoznaczna, dzięki czemu możemy zbiory bez problemu porównywać. Taka reprezentacja nazywana jest *maską bitową* zbioru. Jak łatwo zauważyć, maska bitowa żadnego zbioru mieczy nie przekroczy

$$\sum_{k=0}^{p-1} 2^k = 2^p - 1 \leqslant 8\,191, \tag{2}$$

co umożliwia wykorzystanie do ich przechowywania (z solidnym zapasem) chociażby liczb całkowitych 32-bitowych. Dzięki temu do obliczania sumy zbiorów można użyć operacji sumy logicznej na bitach, w Pascalu zapisywanej jako **or**, a w C/C++ jako „|”. Oznaczenia te nie są przypadkowe — suma na pojedynczych bitach to nic innego jak alternatywa logiczna. Przykładowo:

$$19 \mid 48 = [0000000010011]_2 \mid [0000000110000]_2 = [0000000110011]_2 = 51.$$

Ostatnie szlify

Jedyną rzeczą dzielącą nas od rozwiązania wzorcowego jest konstrukcja grafu H. Ustaliliśmy już, że zbiory będziemy reprezentować jako ich maski bitowe. Na mocy (2), liczba wierzchołków wyniesie wówczas $2^p n \approx 1{,}6 \cdot 10^6$, co nie sprawi nam większego kłopotu. Jednak liczba krawędzi może osiągnąć $2^p m \approx 2{,}5 \cdot 10^7$, a taka ich ilość prawie na pewno nie zmieści się w skromnym limicie pamięciowym wynoszącym 32 MB. Aby temu zaradzić, możemy konstruować krawędzie dynamicznie w oparciu o krawędzie G i zależność (1). Oto pseudokod rozwiązania (przyjmujemy, że na początku $t[v][S] = \infty$ dla wszystkich v, S):

```
 1: t[1][M(1)] := 0;
 2: queue.push((1, M(1)));
 3: while not queue.empty() do
 4: begin
 5:   (v,S) := queue.top();
 6:   queue.pop();
 7:   for w : (v,w) ∈ E(G) do
 8:     if P((v,w)) | S = S then
 9:       if t[v][S] + time((v,w)) < t[w][S | M(w)] then
10:       begin
11:         t[w][S | M(w)] := t[v][S] + time((v,w));
12:         queue.update((w, S | M(w)));
13:       end
14: end
```

Zakładamy, że operacja $queue.top()$ zwraca taką parę (v,S) znajdującą się w kolejce priorytetowej, dla której wartość $t[v][S]$ jest minimalna, $queue.pop()$ usuwa taką parę z kolejki, zaś $queue.update(x)$ wstawia element x do kolejki lub aktualizuje jego położenie w kolejce, jeśli x już się tam znajduje. Algorytm Dijkstry z użyciem kolejki priorytetowej działa w czasie $O(|E| \log |V|)$, co w naszym przypadku daje złożoność $O(2^p m \cdot p \log n)$.

Jest to złożoność pesymistyczna i w przypadku losowych testów omijamy większość wierzchołków, więc program działa szybciej, niż można by się spodziewać z analizy złożoności czasowej. Natomiast złożoność pamięciowa jest rzędu $O(2^p n)$, ponieważ dominującym czynnikiem jest rozmiar tablicy t. Rozwiązanie to zostało zaimplementowane w plikach `wie.cpp` i `wie3.cpp`.

Inne rozwiązania

Jak zrobiłby to wiedźmak i dlaczego zrobiłby to źle

Znając wiedźmakowe podejście, wybierałby on pierwszą drogę, którą może przejść, i szedłby tak naprzód, aż napotka przeszkodę. Wtedy zawróciłby w poszukiwaniu odpowiedniego miecza i powtarzałby całą tę procedurę do skutku. Innymi słowy, próbowałby rozwiązać problem metodą prób i błędów. Nadzieje na skonstruowanie w ten sposób najkrótszej ścieżki są jednak bardzo mizerne — takiemu wiedźmakowi cała wędrówka zapewne zajęłaby z 7 tomów powieści.

Zniecierpliwiony wiedźmak zapewne szukałby najkrótszej ścieżki do Bajtogrodu w grafie G, ignorując potwory i licząc na łut szczęścia. Oczywiste jest, że takie podejście, choć efektowne, praktycznie nie ma szans zadziałać. Amatorzy mocnych wrażeń znajdą je w pliku `wieb1.cpp`.

Zapominalski wiedźmak zapomniałby kupić wszystkie dostępne miecze przed wyruszeniem w podróż i rozpoczęłby wyszukiwanie z wierzchołka $[1,0]$, a nie $[1,M(1)]$. Błąd ten popełnić bardzo łatwo, a takie rozwiązanie zdobywało jedynie 40 punktów. Można je znaleźć w pliku `wieb2.cpp`.

Rozwiązania wolniejsze

Wszystkie przygotowane przez jurorów rozwiązania wolniejsze opierają się w mniejszym lub większym stopniu na schemacie wykorzystanym w rozwiązaniu wzorcowym. Ze względu na specyficzną strukturę grafu H, część z nich, mimo teoretycznie gorszej złożoności, uzyskiwała maksymalną punktację (rozwiązania `wie1.cpp` i `wie2.pas`, korzystające ze zwykłej kolejki z wielokrotnym wstawianiem elementów zamiast kopca), a część nieco niższą (rozwiązanie `wies1.cpp`, w którym dla rozważanego wierzchołka za każdym razem zostają przetworzone wszystkie maski bitowe — 60 punktów — i rozwiązanie `wies2.cpp`, implementujące siłową wersję algorytmu Dijkstry niewykorzystującą kopca — 80 punktów).

Testy

Do oceny rozwiązań przygotowano 10 zestawów testów. Pierwsze trzy zestawy zawierają nieduże testy poprawnościowe wygenerowane ręcznie (w każdym z tych zestawów zadbano o to, żeby w miejscowości numer 1 występowały jakieś potrzebne miecze). Wszystkie pozostałe zestawy składają się wyłącznie z testów losowych: w każdym z nich obierano losowy graf, po czym w części testów wybierano pewną ścieżkę o zadanej długości i rozmieszczano miecze w grafie w taki sposób, aby ta właśnie ścieżka była najkrótszą ścieżką dla Bajtazara, natomiast w pozostałych testach miecze rozmieszczano w grafie zupełnie losowo.

Oznaczenia w poniższej tabelce są zgodne z tymi w treści zadania, tzn. reprezentują odpowiednio: liczbę miejscowości, liczbę dróg, liczbę rodzajów potworów (a zarazem mieczy) oraz liczbę kowali. Niniejsze zadanie nie sprawiło zawodnikom większych problemów i stanowiło rozgrzewkę przed właściwymi zawodami III stopnia.

Nazwa	n	m	p	k	Opis
wie1a.in	1	0	4	1	mały test poprawnościowy
wie1b.in	4	4	4	4	mały test poprawnościowy
wie1c.in	1	0	1	0	mały test poprawnościowy
wie1d.in	2	1	13	2	mały test poprawnościowy
wie2.in	12	14	4	8	mały test poprawnościowy
wie3.in	70	115	7	9	średni test poprawnościowy
wie4.in	101	166	10	10	test losowy
wie5.in	140	573	10	20	test losowy
wie6.in	150	684	11	40	test losowy
wie7a.in	150	2624	12	9	test losowy, miejscowość nr 1 zawiera potrzebne miecze
wie7b.in	150	2624	12	9	test losowy, miejscowość nr 1 zawiera potrzebne miecze
wie8a.in	180	2718	13	30	test losowy, miejscowość nr 1 zawiera potrzebne miecze
wie8b.in	180	2705	13	30	test losowy
wie9a.in	200	778	13	60	test losowy
wie9b.in	200	2796	13	60	test losowy
wie10a.in	200	2779	13	30	test losowy
wie10b.in	200	2793	13	200	test losowy, miejscowość nr 1 zawiera potrzebne miecze
wie10c.in	200	2778	13	200	test losowy, miejscowość nr 1 zawiera potrzebne miecze
wie10d.in	200	2781	13	1	test losowy

Marian M. Kędzierski — Treść zadania
Marian M. Kędzierski, Błażej Osiński — Opracowanie
Błażej Osiński — Program

Dostępna pamięć: 32 MB. *OI, Etap III, dzień pierwszy, 01.04.2009*

Tablice

Rozważmy tablicę o wymiarach $n \times m$ wypełnioną **różnymi** *liczbami całkowitymi. Na tej tablicy możemy wykonywać następujące operacje:*

1. *zamiany dwóch wierszy*
2. *zamiany dwóch kolumn.*

Powiemy, że dwie tablice są **podobne**, *jeżeli przy pomocy pewnej sekwencji powyższych operacji wykonanych na pierwszej tablicy możemy z niej otrzymać drugą.*

Napisz program, który dla danego zestawu par tablic stwierdzi, które pary zawierają tablice podobne.

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita t ($1 \leqslant t \leqslant 10$) oznaczająca liczbę par tablic. W następnych liniach znajdują się opisy kolejnych par tablic.

Opis pary tablic zaczyna się od wiersza zawierającego dwie liczby całkowite n oraz m ($1 \leqslant n, m \leqslant 1000$) oddzielone pojedynczym odstępem, oznaczające odpowiednio liczbę wierszy oraz liczbę kolumn obu tablic.

W następnych n wierszach znajduje się opis pierwszej tablicy. W i-tym spośród tych wierszy znajduje się m liczb całkowitych a_{ij} ($-1\,000\,000 \leqslant a_{ij} \leqslant 1\,000\,000$) pooddzielanych pojedynczymi odstępami, oznaczających kolejne liczby w i-tym wierszu pierwszej tablicy.

W następnych n wierszach znajduje się opis drugiej tablicy. W i-tym spośród tych wierszy znajduje się m liczb całkowitych b_{ij} ($-1\,000\,000 \leqslant b_{ij} \leqslant 1\,000\,000$) pooddzielanych pojedynczymi odstępami, oznaczających kolejne liczby w i-tym wierszu drugiej tablicy.

Wszystkie liczby występujące w jednej tablicy są parami różne.

Wyjście

Twój program powinien wypisać na standardowe wyjście t wierszy. W k-tym z nich powinno znaleźć się jedno słowo „`TAK`*”, jeżeli tablice w k-tej wczytanej parze są podobne, zaś słowo „*`NIE`*” w przeciwnym przypadku.*

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`2`	`TAK`
`4 3`	`NIE`
`1 2 3`	
`4 5 6`	
`7 8 9`	
`10 11 12`	
`11 10 12`	
`8 7 9`	
`5 4 6`	
`2 1 3`	
`2 2`	
`1 2`	
`3 4`	
`5 6`	
`7 8`	

Wyjaśnienie do przykładu: *Pierwsza para zawiera tablice podobne. Aby przetworzyć pierwszą tablicę na drugą, wystarczy zamienić ze sobą pierwsze dwie kolumny, a następnie pierwszy wiersz z ostatnim i drugi wiersz z trzecim.*

Druga para zawiera tablice, które nie są podobne. Aby to stwierdzić, wystarczy zauważyć, że zbiory wartości w ich komórkach są różne.

Rozwiązanie

Rozwiązanie siłowe

Dla uproszczenia opisu nazwijmy wymienione w treści zadania operacje na tablicy *operacjami elementarnymi*, zaś tablice, na których operujemy — *macierzami*[1].

Najprostsze rozwiązanie naszego zadania polega na generowaniu wszystkich możliwych macierzy, jakie można utworzyć z pierwszej macierzy przy użyciu operacji elementarnych, i sprawdzaniu, dla każdej z nich, czy otrzymaliśmy drugą macierz. Niestety takie rozwiązanie siłowe jest bardzo powolne, ponieważ liczba możliwych wyników ciągów operacji to iloczyn liczb permutacji wierszy i kolumn, a zatem złożoność czasowa takiego rozwiązania wyniosłaby $\Omega(n! \cdot m!)$. Wobec ograniczeń z zadania, oczywiście nie możemy sobie na to pozwolić.

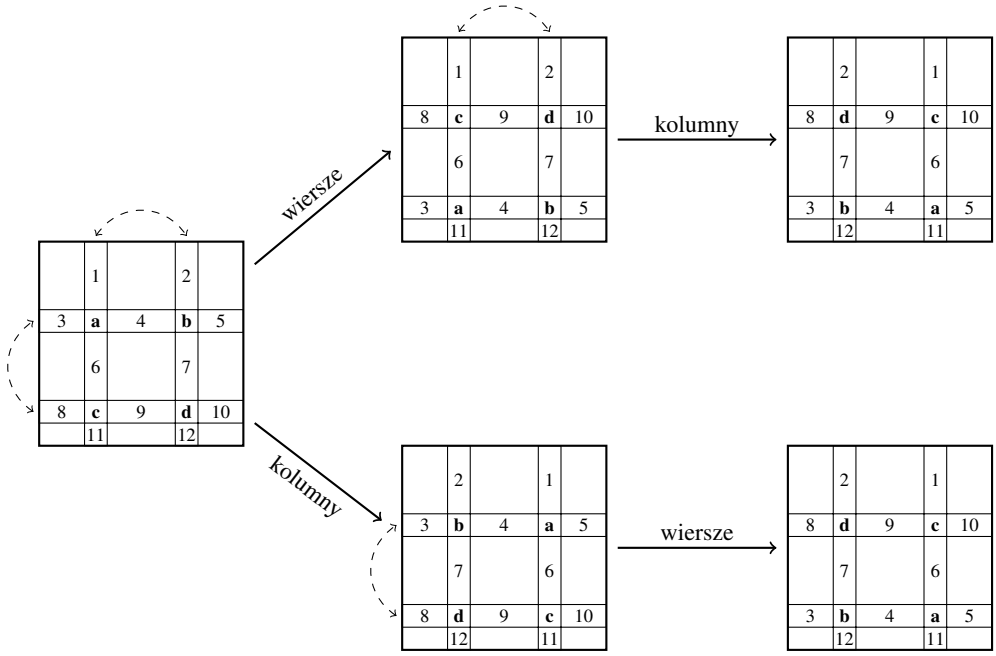
Kluczowe spostrzeżenia

Kluczowym spostrzeżeniem prowadzącym do rozwiązania zadania jest to, że operacje elementarne nie mieszają zawartości wierszy między sobą. Innymi słowy, jeżeli dwie licz-

[1] W rzeczywistości istnieje w algebrze liniowej pojęcie *operacji elementarnej* na *macierzy* i obejmuje ono nieco szerszą klasę operacji niż tutaj rozważane. Od nich także wzięła się inspiracja do niniejszego zadania.

by a i b znajdują się w tym samym wierszu, to po wykonaniu dowolnej liczby operacji elementarnych będą one nadal w jednym wierszu, potencjalnie o innym numerze. Można to łatwo udowodnić, zauważając, że operacja zamiany wierszy nie zmienia zawartości poszczególnych wierszy, a jedynie ich kolejność, natomiast operacja zamiany kolumn jedynie przestawia elementy w ramach każdego z wierszy. Podobnie rzecz ma się z kolumnami.

Drugim kluczowym spostrzeżeniem jest to, że jeżeli chcemy wykonać kolejno dwie operacje elementarne, jedną na wierszach, a drugą na kolumnach, to kolejność ich wykonania nie ma znaczenia — w obu przypadkach otrzymamy to samo (patrz rys. 1).



Rys. 1: Rysunkowy dowód faktu, że operacje zamiany miejscami dwóch wierszy i dwóch kolumn są przemienne.

Z powyższego spostrzeżenia wynika w szczególności, że jeżeli istnieje ciąg operacji elementarnych przekształcający jedną macierz w drugą, to istnieje też ciąg złożony z tych samych operacji i dający taki sam wynik, na którego początku są tylko operacje na wierszach, a na końcu na kolumnach.

Rozwiązanie wzorcowe

Jak wykorzystać poczynione spostrzeżenia? Otóż nadal będziemy próbowali przetworzyć pierwszą macierz tak, aby stała się identyczna z drugą, jednak tym razem nie musimy już badać wszystkich permutacji, gdyż wiemy, że wiele z nich będzie prowadziło do tego samego wyniku.

Spróbujmy uszeregować wiersze pierwszej macierzy tak, aby zgadzały się z wierszami drugiej macierzy. Oczywiście problemem jest to, że odpowiadające sobie wiersze w obu macierzach nie są identyczne (mogą różnić się kolejnością liczb). I tutaj przychodzi nam z pomocą istotna własność, wyraźnie wspomniana w treści zadania: liczby w komórkach każdej macierzy są parami różne. Dzięki temu możemy identyfikować poszczególne wiersze na przykład z najmniejszymi liczbami, które się w nich znajdują. Jeżeli bowiem macierze mają być podobne, to najmniejsze wyrazy odpowiadających sobie wierszy muszą być równe.

Aby więc dobrać odpowiednią kolejność wierszy pierwszej macierzy, wystarczy w każdej z macierzy posortować tablice złożone z par: numer wiersza i jego najmniejszy wyraz (sortujemy według najmniejszego wyrazu). W ten sposób albo od razu dowiadujemy się, że macierze nie są podobne (jeżeli zbiory najmniejszych wyrazów nie zgadzają się), albo jesteśmy w stanie przyporządkować każdemu wierszowi pierwszej macierzy odpowiedni wiersz drugiej macierzy. Jeżeli się to udało, to zmieniamy odpowiednio kolejność wierszy pierwszej macierzy, zaś drugą pozostawiamy bez zmian. W ten sposób otrzymujemy takie dwie macierze, że dla każdego $i \in \{1,2,\ldots,n\}$ najmniejsza liczba w i-tym wierszu każdej z nich jest taka sama oraz wyjściowe macierze są podobne wtedy i tylko wtedy, gdy pierwszą z nowo powstałych macierzy można przekształcić w drugą jedynie za pomocą zamian kolumn miejscami. Następnie w identyczny sposób możemy uszeregować kolumny (niezależnie od wyniku przyporządkowania wierszy).

Po wykonaniu tych dwóch kroków (uszeregowania wierszy i kolumn) albo przekonamy się, że dwie rozważane macierze nie mogą być podobne, albo otrzymamy jedyne permutacje kolumn i wierszy pierwszej macierzy reprezentujące zestaw operacji elementarnych, który może przeprowadzać ją na drugą macierz. Aby jednak upewnić się co do podobieństwa, musimy jeszcze faktycznie wykonać te operacje na pierwszej macierzy, a następnie sprawdzić, czy rzeczywiście otrzymaliśmy w ten sposób drugą macierz.

Złożoność czasowa tego algorytmu to $O(n \cdot m + n \cdot \log n + m \cdot \log m)$, jeżeli użyjemy sortowania działającego w czasie liniowo-logarytmicznym (np. sortowania przez scalanie). Jednak dzięki stosunkowo małemu zakresowi liczb w komórkach macierzy (od $-d$ do d dla $d \leqslant 1\,000\,000$) każde z sortowań możemy wykonać w czasie liniowym względem d (sortowanie przez zliczanie), co prowadzi do złożoności czasowej $O(n \cdot m + d)$.

Algorytm wzorcowy działa podobnie, tyle że w drugiej fazie (czyli przy znajdowaniu przyporządkowania kolumn) korzysta z wyniku uszeregowania wierszy i porównuje ze sobą jedynie dowolne dwa odpowiadające sobie wiersze. Na końcu, podobnie jak wyżej, stosuje uzyskane permutacje do pierwszej macierzy i sprawdza, czy otrzymano drugą. Jego implementacje znajdują się w następujących plikach: `tab.cpp`, `tab1.c`, `tab2.pas`, `tab3.java`.

Inne rozwiązania

Powyższe rozwiązanie nie jest jedynym, za które przewidziana była maksymalna liczba punktów. Można także podać wiele innych, równie dobrych metod. Jedną z nich jest podejście oparte na sprowadzeniu macierzy do pewnej *postaci kanonicznej*.

Jest to dosyć częsta metoda sprawdzania, czy dwa elementy jakiegoś zbioru są w relacji równoważności[2]. Nie jest tu istotne, co rozumiemy przez „postać kanoniczną" — ważne jest tylko, żeby był to jakiś ściśle określony reprezentant każdej klasy równoważności[3], którego jesteśmy w stanie łatwo wygenerować z każdego elementu zbioru. W tym przypadku chodzi nam o znalezienie jakiejś funkcji, która każdą macierz przekształci na podobną do niej, przy czym dwie macierze podobne do siebie zawsze przekształci na taką samą.

Inne, klasyczne przykłady takiego podejścia to:

- Badanie, czy dwa słowa są anagramami, które polega na zliczeniu wystąpień każdej litery w obu słowach i porównaniu otrzymanych statystyk.
- Badanie, czy zbiory wyrazów w dwóch ciągach są równe, które polega na posortowaniu każdego z ciągów i porównaniu ich posortowanych postaci.

Znalezienie „postaci kanonicznej" często polega na jakimś rodzaju sortowania. W tym wypadku wystarczy, jeżeli posortujemy według pewnego kryterium najpierw wiersze, a potem kolumny obu macierzy. Możemy np. sortować wiersze względem minimalnych elementów, a następnie kolumny względem elementów w pierwszym wierszu. Rozwiązanie takie można zaimplementować w takiej samej złożoności jak wzorcowe.

Jeżeli dobrze przyjrzymy się temu algorytmowi, to odkryjemy, że jest on bardzo podobny do poprzedniego, z tą różnicą, że w algorytmie wzorcowym sprowadzaliśmy pierwszą macierz do drugiej (permutując kolumny i wiersze pierwszej macierzy tak, aby zgadzały się z drugą), natomiast w niniejszym rozwiązaniu permutujemy wiersze i kolumny obu macierzy naraz. Implementacja tego rozwiązania znajduje się w pliku `tab4.cpp`.

Można wymienić wiele podobnych algorytmów, w których konstruowane postaci kanoniczne są nieco inne, nie zawsze jednak otrzymamy w ten sposób efektywne rozwiązanie. Wszystko zależy od tego, jaki rodzaj postaci kanonicznej wybierzemy i jak szybko jesteśmy w stanie ją otrzymywać z dowolnej macierzy.

Haszowanie

Można także posłużyć się zupełnie inną, ciekawą metodą, opartą na haszowaniu. Daje ona prostą heurystykę, która z dużym prawdopodobieństwem działa poprawnie, mimo iż nie można zagwarantować jej poprawności w ogólnym przypadku.

Podejście to polega na skonstruowaniu dla każdej z macierzy swoistego „zbioru kolumn", w którym każda kolumna jest reprezentowana przez swoje posortowane elementy zakodowane w jednej liczbie całkowitej, oraz porównaniu tych zbiorów dla obu macierzy. Następnie operację tę należy powtórzyć dla wierszy.

W jaki sposób zakodować ciąg wyrazów z kolumny w jednej liczbie całkowitej? Niech $a_1 < a_2 < \ldots < a_n$ będą elementami pewnej kolumny posortowanymi rosnąco i niech Q będzie pewną liczbą całkowitą (najlepiej pierwszą i większą niż wszystkie a_i). Wtedy liczba

$$Z = a_1 \cdot Q^{n-1} + a_2 \cdot Q^{n-2} + \ldots + a_{n-1} \cdot Q + a_n$$

[2] *Relacja równoważności* to taka relacja pomiędzy elementami pewnego zbioru, która jest *zwrotna* (czyli każdy element jest sam ze sobą w relacji), *symetryczna* (jeżeli a jest w relacji z b, to b jest w relacji z a) oraz *przechodnia* (jeżeli a jest w relacji z b oraz b jest w relacji z c, to również a jest w relacji z c).

[3] *Klasa równoważności* (inaczej klasa abstrakcji) to maksymalny ze względu na zawieranie podzbiór elementów, którego każde dwa elementy są ze sobą w danej relacji równoważności.

reprezentuje cały ciąg (a_i). Nazwijmy ją *charakterystyką* rozważanej kolumny. W przypadku większych testów (lub dużej liczby Q) obliczenie Z często będzie powodowało przepełnienie zakresu liczby. W takiej sytuacji operacje są wykonywane modulo 2^{32} dla typu całkowitego 32-bitowego i modulo 2^{64} dla typu 64-bitowego. Z tego powodu może się zdarzyć, że dwie różne kolumny (odpowiednio dwa wiersze) zostaną odwzorowane na tę samą liczbę Z i w efekcie macierze zostaną uznane za podobne, mimo iż podobne nie są (taką sytuację nazywamy *konfliktem* w haszowaniu).

Zauważmy, że operacje elementarne nie zmieniają zbioru charakterystyk kolumn i wierszy macierzy. Jeżeli więc okaże się, że zbiór charakterystyk kolumn (wierszy) jest inny w przypadku pierwszej macierzy niż w przypadku drugiej, to nie jest możliwe, żeby macierze te były do siebie podobne. Jeżeli jednak zbiory te będą identyczne i nie wystąpią żadne konflikty haszowania, to można pokazać, że macierze rzeczywiście są podobne. W tym celu należy dowieść, że ze zbioru charakterystyk wierszy i kolumn macierzy można odtworzyć całą macierz z dokładnością do podobieństwa, czyli zrekonstruować pewną postać kanoniczną macierzy. Uzasadnienie tego faktu pozostawiamy Czytelnikowi.

Powyższa metoda nie daje wprawdzie gwarancji poprawności, jest jednak łatwa do zapisania i trudno jest przygotować testy, dla których działa ona niepoprawnie (dlatego że nie wiadomo, jaka stała Q została użyta w programie). Rozwiązanie takie odpowiednio zaimplementowane z bardzo dużym prawdopodobieństwem uzyskiwało komplet punktów. Gdybyśmy chcieli to prawdopodobieństwo zwiększyć, moglibyśmy powtórzyć obliczenie kilkukrotnie z użyciem różnych wartości Q. Rozwiązanie to, podobnie jak większość rozwiązań opartych na haszowaniu, jest godne polecenia, ponieważ prawdopodobieństwo błędnej odpowiedzi spowodowanej użyciem haszowania jest wielokrotnie mniejsze niż prawdopodobieństwo błędu implementacyjnego w przypadku wybrania bardziej skomplikowanego algorytmu. Jeżeli jednak chcemy mieć pewność, że zastosowany algorytm jest poprawny, to niestety nie jest to podejście satysfakcjonujące.

Przykład implementacji powyższej heurystyki znajduje się w pliku `tab5.cpp`. Dodajmy na koniec, że analogiczne rozwiązania, które jako charakterystyki wierszy i kolumn obierają jakieś prostsze wartości, np. sumy, minima czy maksima elementów, już poprawne nie są — znalezienie odpowiednich kontrprzykładów na takie rozwiązania pozostawiamy Czytelnikowi jako ciekawe ćwiczenie.

Testy

Rozwiązania zawodników były sprawdzane na 10 zestawach danych testowych:

Nazwa	max **n**	max **m**	**Opis**
tab1.in	10	10	bardzo małe i proste testy
tab2.in	10	10	małe testy, kwadratowe macierze
tab3.in	20	20	małe testy, prostokątne macierze
tab4.in	130	130	średniej wielkości testy, kwadratowe macierze
tab5-6.in	131	140	średniej wielkości testy, prostokątne macierze
tab7-10.in	1000	1000	duże testy

Wojciech Rytter, Bartosz Walczak — Treść zadania, Opracowanie
Jakub Radoszewski — Program

Dostępna pamięć: 64 MB. *OI, Etap III, dzień pierwszy, 01.04.2009*

Słowa

Niech h będzie funkcją określoną na napisach złożonych z cyfr `0` *i* `1`*. Funkcja h przekształca napis w, zastępując (niezależnie i równocześnie) każdą cyfrę* `0` *przez* `1` *i każdą cyfrę* `1` *przez napis „*`10`*”. Na przykład $h($„`1001`”$)=$ „`101110`”, $h($„”$)=$ „” (tzn. funkcja h zastosowana do pustego napisu jest pustym napisem). Zauważmy, że h jest różnowartościowa. Przez h^k oznaczmy k-krotne złożenie funkcji h ze sobą. W szczególności, h^0 to funkcja identycznościowa $h^0(w)=w$.*

Interesują nas napisy postaci $h^k($„`0`”$)$ dla $k=0,1,2,3,\ldots$ Oto kilka pierwszych takich napisów:

„`0`”, „`1`”, „`10`”, „`101`”, „`10110`”, „`10110101`”.

Mówimy, że napis x jest **podsłowem** *napisu y, jeżeli występuje w nim jako spójny (tj. jednokawałkowy) podciąg. Mamy dany ciąg liczb naturalnych $k_1,k_2,\ldots,k_n$. Celem zadania jest sprawdzenie, czy napis postaci*

$$h^{k_1}(\text{„0”})\cdot h^{k_2}(\text{„0”})\cdot\ldots\cdot h^{k_n}(\text{„0”})$$

jest podsłowem $h^m($„`0`”$)$ dla pewnego m. Przy tym operacja „$\cdot$” oznacza sklejenie (konkatenację) napisów.

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą t, $1\leqslant t\leqslant 13$, oznaczającą liczbę przypadków testowych do rozważenia. Pierwszy wiersz opisu każdego przypadku zawiera jedną liczbę całkowitą n, $1\leqslant n\leqslant 100\,000$. W drugim wierszu opisu znajduje się n nieujemnych liczb całkowitych $k_1,k_2,\ldots,k_n$ pooddzielanych pojedynczymi odstępami. Suma liczb z drugiego wiersza każdego przypadku jest nie większa niż 10 000 000.

Wyjście

Twój program powinien wypisać na standardowe wyjście t wierszy, po jednym dla każdego przypadku testowego. Wiersz odpowiadający danemu przypadkowi testowemu powinien zawierać jedno słowo: `TAK` *— jeśli w tym przypadku $h^{k_1}($„`0`”$)\cdot h^{k_2}($„`0`”$)\cdot\ldots\cdot h^{k_n}($„`0`”$)$ jest podsłowem $h^m($„`0`”$)$ dla pewnego m, lub* `NIE` *w przeciwnym razie.*

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`2`	`TAK`
`2`	`NIE`
`1 2`	
`2`	
`2 0`	

Wyjaśnienie do przykładu: *Słowo z pierwszego przypadku testowego to „`110`" — jest ono podsłowem na przykład słowa $h^4($„`0`"$)$ = „`10110`". W drugim przypadku testowym występuje słowo „`100`", które nie jest podsłowem żadnego słowa postaci $h^m($„`0`"$)$.*

Rozwiązanie

Dla wygody wprowadźmy oznaczenie $h_k := h^k(0)$. Słowa h_k są to tzw. słowa Fibonacciego. Zazwyczaj definiuje się je rekurencyjnie w następujący sposób:

$$h_0 = 0, \quad h_1 = 1, \quad h_k = h_{k-1} \cdot h_{k-2} \quad \text{dla } k \geqslant 2. \tag{$*$}$$

Równoważność definicji $(*)$ i tej podanej w treści zadania łatwo pokazać przez indukcję. Rozwiązanie wzorcowe bazuje na definicji przez podstawienie, którą podano w treści zadania. Postać $(*)$ może być jednak pomocna w zrozumieniu niektórych własności, z których będziemy korzystać.

Powiemy, że ciąg $(k_1, \ldots, k_n)$ jest *dobry*, jeżeli $h_{k_1} \ldots h_{k_n}$ jest podsłowem h_m dla pewnego m, oraz że jest *zły* w przeciwnym przypadku. Zadanie polega więc na sprawdzeniu, czy dany na wejściu ciąg $(k_1, \ldots, k_n)$ jest dobry.

Pierwsze podejście do rozwiązania tego zadania może wyglądać następująco: generujemy całe słowo $u = h_{k_1} \ldots h_{k_n}$, po czym sprawdzamy za pomocą algorytmu wyszukiwania wzorca (np. KMP), czy występuje ono jako podsłowo w h_m dla odpowiednio dobranego m. Okazuje się, że jeżeli h_m jest co najmniej 8 razy dłuższe niż wyszukiwane słowo, to rozwiązanie takie działa poprawnie. Uzasadnienie tego spostrzeżenia polega na tym, że jeżeli u jest nie dłuższe niż h_{m-3} dla pewnego m, to jeśli jest ono podsłowem h_{m+1}, to jest też podsłowem h_m — równoważnie, jeżeli nie uda nam się znaleźć u w h_m, to nie ma potrzeby szukania tego słowa w h_{m+1} i dalszych. Faktycznie, jeżeli przy podziale $h_{m+1} = h_m \cdot h_{m-1}$ słowo u leży w ramach h_m lub h_{m-1}, to jest to jasne, a w przeciwnym przypadku wystarczy skorzystać z zapisu $h_{m+1} = h_{m-1}h_{m-2} \cdot h_{m-3}h_{m-4}h_{m-3}$, jako że $h_{m-2} \cdot h_{m-3} = h_{m-1}$.

Już pobieżna analiza pozwala stwierdzić, że przy ograniczeniach na $k_1, \ldots, k_n$ podanych w treści zadania takie rozwiązanie nie ma szans powodzenia na dużych danych wejściowych, gdyż rozmiar słowa h_k rośnie wykładniczo względem k. Dlatego rozwiązanie, którego szukamy, musi operować tylko na ciągu $(k_1, \ldots, k_n)$, bez generowania słowa, które ten ciąg reprezentuje.

Rozwiązanie wzorcowe

Idea rozwiązania wzorcowego polega na przekształcaniu danego ciągu $w = (k_1, \ldots, k_n)$ przez coś w rodzaju funkcji odwrotnej do h. Okazuje się, że dla dużych elementów ciągu w cofanie

funkcji h polega na zwykłym ich zmniejszaniu, a dla małych (nie większych niż 3) trzeba czasem rozpatrywać pewne przypadki szczególne. Operacje wykonywane w algorytmie mają tę własność, że ciąg dobry przekształcają w ciąg dobry, a zły — w zły. Proces ten doprowadza w końcu do ciągu jednoelementowego, który oczywiście jest dobry (wtedy ciąg początkowy też był dobry), lub do ciągu, o którym potrafimy stwierdzić, że na pewno jest zły (wtedy początkowy był zły).

```
1:  w := (k_1, ..., k_n)
2:  while |w| > 1 do begin
3:    if w zawiera fragment k,0 dla k ∉ {1,3} then return false
4:    Wykonaj kolejno następujące operacje na ciągu w:
5:      zamień, jeżeli występuje, pierwszy element 0 → 2
6:      zamień, jeżeli występuje, ostatni element 3 → 2
7:      usuń, jeżeli występuje, ostatni element równy 1
8:      zamień wszystkie fragmenty 1,0 → 2
9:      zamień wszystkie fragmenty 3,0 → 2,2
10:     zmniejsz wszystkie elementy o 1
11: end
12: return true
```

Pełny kod rozwiązania znajduje się w plikach `slo.cpp` oraz `slo1.pas`.

Przykład działania algorytmu

Zobaczmy, jak zadziała rozwiązanie wzorcowe dla ciągu $w = (1,2,4,1,2,5)$. Poniższa tabela przedstawia, co dzieje się z ciągiem w w kolejnych iteracjach pętli **while** pod wpływem instrukcji w liniach 5–10 (numer instrukcji nad strzałką).

nr	przebieg iteracji
1	$(1,2,4,1,2,5) \xrightarrow{10} (0,1,3,0,1,4)$
2	$(0,1,3,0,1,4) \xrightarrow{5} (2,1,3,0,1,4) \xrightarrow{9} (2,1,2,2,1,4) \xrightarrow{10} (1,0,1,1,0,3)$
3	$(1,0,1,1,0,3) \xrightarrow{6} (1,0,1,1,0,2) \xrightarrow{8} (2,1,2,2) \xrightarrow{10} (1,0,1,1)$
4	$(1,0,1,1) \xrightarrow{7} (1,0,1) \xrightarrow{8} (2,1) \xrightarrow{10} (1,0)$
5	$(1,0) \xrightarrow{8} (2) \xrightarrow{10} (1)$

Iteracja 6 już się nie wykona, gdyż otrzymany w wyniku iteracji 5 ciąg w jest jednoelementowy. Jako wynik algorytmu otrzymujemy zatem, że ciąg $(1,2,4,1,2,5)$ jest dobry.

Złożoność rozwiązania wzorcowego

Zanim przystąpimy do dowodu poprawności przedstawionego rozwiązania, zastanówmy się nad czasem jego działania. W szczególności upewnijmy się, że algorytm ten w ogóle się zatrzymuje.

Pokażemy, że czas działania algorytmu na ciągu $(k_1, \ldots, k_n)$ wynosi $O(k_1 + \ldots + k_n + n)$. Przyjrzyjmy się pojedynczej iteracji pętli **while**. Niech n oznacza bieżącą długość ciągu w, a s — bieżącą sumę elementów ciągu w z pominięciem pierwszego, jeżeli jest mniejszy od 2. Pokażemy, że po wykonaniu instrukcji 5–10 wartość $s+n$ maleje o co najmniej $\lfloor \frac{n}{2} \rfloor$. Faktycznie, wykonanie instrukcji 5 (w połączeniu z instrukcją 10) nie zmienia wartości $s+n$, gdyż wówczas początkowe 0 lub 1 w ciągu nie jest uwzględnione w s. Inne zmiany, czyli:

- końcowe 3 przechodzi w 1 (dwukrokowo, wskutek instrukcji 6 i 10),
- końcowe 1 znika,
- fragment $1,0$ przechodzi w 1 (dwukrokowo),
- fragment $3,0$ przechodzi w $1,1$ (dwukrokowo),
- pozostałe elementy zmniejszają się o 1,

powodują zmniejszenie wartości $s+n$ o co najmniej 1 dla każdego elementu lub każdej pary kolejnych elementów ciągu w. Wynika stąd, że $s+n$ maleje łącznie o co najmniej $\lfloor \frac{n}{2} \rfloor$ (zmiana ta równałaby się $\lceil \frac{n}{2} \rceil$, gdyby nie specjalne traktowanie początkowego elementu w). Oczywiście jeżeli $s+n \leqslant 1$, algorytm się zatrzymuje. Jako że czas wykonania pojedynczej iteracji pętli **while** wynosi $O(n)$, jest on w pełni amortyzowany przez zmianę $s+n$. Zatem całkowity czas działania algorytmu wynosi $O(s+n)$.

Kilka prostych własności słów h_k

W przekształceniu $h(h_{k-1}) = h_k$ każda cyfra `0` w h_k powstaje w wyniku podstawienia `1` $\to$ `10`, a każda cyfra `1`, po której nie występuje `0` — w wyniku podstawienia `0` $\to$ `1`. Wynika stąd w szczególności, że:

(a) h_k zaczyna się od `1` dla $k \geqslant 1$,

(b) h_k kończy się na `0` dla k parzystych, a na `1` dla k nieparzystych,

(c) w h_k nie występuje podsłowo `00`,

(d) w h_k nie występuje podsłowo `111` (jest to wniosek z (c) dla słowa h_{k-1}),

(e) w h_k nie występuje podsłowo `101010` (jest to wniosek z (d) dla słowa h_{k-1}).

Poprawność rozwiązania wzorcowego

Zaczniemy od uzasadnienia, że kryterium w linii 3, które odrzuca ciąg w jako zły, jest poprawne.

Lemat 1. Jeżeli $k \notin \{1,3\}$, to h_k`0` nie jest podsłowem żadnego h_m.

Dowód: Jeżeli k jest parzyste, h_k kończy się cyfrą `0`. Wtedy h_k`0` ma sufiks `00`, który nie może być podsłowem żadnego h_m (własność (c)). Jeżeli k jest nieparzyste i $k \geqslant 5$, to można udowodnić (np. korzystając z postaci $(*)$), że słowo h_k ma sufiks $h_5 =$ `10110101`, a więc także `10101`, czyli `101010` jest sufiksem słowa h_k`0`. Ale zgodnie z własnością (e) powyżej, słowo `101010` nie może wystąpić w żadnym h_m. ■

Teraz uzasadnimy, że każda z operacji wykonywanych w liniach 5–10 jest poprawna, tzn. że ciąg w jest dobry po wykonaniu tej operacji wtedy i tylko wtedy, gdy był dobry przed jej wykonaniem. Zauważmy przy tym, że operacje w liniach 5–9 są od siebie niezależne i mogą zostać wykonane w dowolnej kolejności. Dla ciągu $w = (k_1, \ldots, k_n)$ wprowadzamy oznaczenie $h_w = h_{k_1} \ldots h_{k_n}$.

Jeżeli w ciągu w, poza pierwszym elementem, występuje jakiekolwiek 0, to musi być ono poprzedzone przez 1 albo 3, gdyż w przeciwnym przypadku ciąg w zostałby odrzucony w linii 3. Ponieważ $h_1h_0 =$ `10` $= h_2$ i $h_3h_0 =$ `1010` $= h_2h_2$, instrukcje 8 i 9 przekształcają ciąg w w równoważny ciąg w', taki że $h_w = h_{w'}$. To dowodzi, że operacje te są poprawne, ponadto eliminują one wszystkie zera występujące w ramach w poza być może pierwszym.

Aby pozbyć się początkowego zera, zauważmy, że jeżeli h_w występuje jako podsłowo w h_m, to początkowe zero musi być drugą cyfrą fragmentu `10` $= h_2$. Oznacza to, że jeżeli na początku ciągu w zamienimy 0 na 2, to dla tak otrzymanego w' słowo $h_{w'}$ również występuje jako podsłowo w h_m. To dowodzi poprawności operacji w linii 5.

Uzasadnimy teraz poprawność instrukcji 6, 7 i 10. Załóżmy, że w ciągu w nie występuje żadne zero. Oznaczmy przez w' ciąg, który powstaje z w po zastosowaniu operacji w liniach 6 i 7. Można zauważyć, że każda z tych operacji odcina ostatnią cyfrę `1` z h_w, jeżeli ciąg w kończy się na 1 lub 3. Zatem $h_w = h_{w'}$ lub $h_w = h_{w'}$`1`, przy czym w tym pierwszym przypadku ciąg w' nie kończy się na 1 ani na 3. Oznaczmy przez w'' ciąg, który powstaje z w' przez zmniejszenie wszystkich elementów o 1 (instrukcja w linii 10).

Udowodnimy najpierw, że jeżeli w jest dobry, to w'' jest dobry. Załóżmy, że h_w jest podsłowem h_m, i zastanówmy się, z czego powstaje to podsłowo w przekształceniu $h(h_{m-1}) = h_m$. Niech $w' = (k_1, \ldots, k_n)$. Pokażemy, że dla każdego i odpowiedni człon h_{k_i} w $h_{w'} = h_{k_1} \ldots h_{k_n}$ powstaje dokładnie z h_{k_i-1}. Funkcja h jest różnowartościowa, a zatem nie istnieje żadne inne słowo x, takie że $h(x) = h_{k_i}$. Jeśli więc h_{k_i} nie powstaje z h_{k_i-1}, to przekształcenie $h(h_{m-1}) = h_m$ zamienia pewną cyfrę `1` w blok `10`, który występuje na granicy wystąpienia h_{k_i} w h_m, tzn. prawe `0` jest pierwszą cyfrą h_{k_i} lub lewe `1` jest ostatnią cyfrą h_{k_i}. Ten pierwszy przypadek jest niemożliwy, bo h_{k_i} zaczyna się od `1` (jako że $k_i \geqslant 1$). Drugi przypadek może zajść jedynie wtedy, gdy po h_{k_i} występuje `0`. Tak nie jest dla $i < n$ (wówczas po h_{k_i} występuje $h_{k_{i+1}}$) ani dla $i = n$ w przypadku, gdy $h_w = h_{w'}$`1`. Zatem musi być $i = n$ oraz $h_w = h_{w'}$, ale wtedy $k_i \notin \{1,3\}$, więc otrzymujemy sprzeczność z Lematem 1. Tym samym pokazaliśmy, że fragment $h_{k_1} \ldots h_{k_n}$ w h_w jako podsłowie h_m powstaje w wyniku zastosowania przekształcenia h do $h_{k_1-1} \ldots h_{k_n-1} = h_{w''}$. Zatem $h_{w''}$ jest podsłowem h_{m-1}, czyli w'' jest dobry.

Pozostała do wykazania implikacja w drugą stronę. Jeżeli w'' jest dobry, to istnieje takie m, że $h_{w''}$`0` lub $h_{w''}$`1` jest podsłowem h_m (wynika to z $(*)$). Stąd $h(h_{w''}$`0`$) = h_{w'}$`1` lub $h(h_{w''}$`1`$) = h_{w'}$`10` jest podsłowem h_{m+1}, a jedno i drugie zaczyna się od h_w. Zatem w jest dobry. To kończy dowód poprawności rozwiązania wzorcowego.

Rozwiązanie alternatywne

Inne rozwiązanie tego zadania bazuje na następującej własności: słowo x jest prefiksem jakiegoś h_m ($m \geqslant 1$) wtedy i tylko wtedy, gdy x można złożyć ze słów Fibonacciego w następujący sposób (uwaga na odwróconą kolejność indeksów):

$$x = h_{\ell_r} \ldots h_{\ell_1}, \quad \text{gdzie} \quad \ell_1 \in \{1,2\}, \quad \ell_i \in \{\ell_{i-1}+1, \ell_{i-1}+2\} \quad \text{dla } i = 2,\ldots,r. \qquad (**)$$

Na przykład słowo `1011010110110101`, które jest prefiksem h_7, ma rozkład $h_5h_4h_2h_1$. Dowód indukcyjny tego faktu korzysta z rekurencyjnej definicji słów Fibonacciego $(*)$.

Każde podsłowo x słowa h_m rozszerza się z lewej do prefiksu h_m. Powyższa własność pozwala łatwo uzyskać minimalne takie rozszerzenie. Wybieramy $\ell_1 = 1$ lub $\ell_1 = 2$ na podstawie ostatniej cyfry słowa x i obcinamy x z prawej o h_{ℓ_1}. To samo robimy kolejno dla $i = 2,\ldots,r$: wybieramy $\ell_i = \ell_{i-1}+1$ lub $\ell_i = \ell_{i-1}+2$ na podstawie ostatniej cyfry i obcinamy x z prawej o h_{ℓ_i}. Ponieważ dwa kolejne słowa Fibonacciego różnią się ostatnią cyfrą, wybór w każdym kroku jest jednoznaczny. Jeżeli w pewnym kroku końcówka słowa x nie zgadza się z wybranym h_{ℓ_i}, to słowo, od którego zaczęliśmy, nie może być podsłowem żadnego h_m. W przeciwnym razie otrzymujemy prefiks h_m, którego sufiksem (a więc podsłowem h_m) jest początkowe słowo x.

Oczywiście, aby to rozwiązanie mogło zadziałać efektywnie, należy konstruować rozkład $(**)$ słowa $h_{k_1} \ldots h_{k_n}$, posługując się jedynie indeksami $k_1,\ldots,k_n$ oraz $\ell_1,\ldots,\ell_r$. Nie jest to jednak trudne, o czym Czytelnik może się przekonać, zaglądając do pliku `sloa.cpp`.

Testy

Zadanie zostało ocenione na zestawie 10 testów, z których każdy zawierał 13 przypadków testowych.

Nazwa	n	s	k z przedziału	Opis
slo1.in	[1, 1457]	[0, 899]	[0, 26]	test ręcznie generowany
slo2.in	[4,20000]	[27,61803]	[0, 28]	test ręcznie generowany
slo3.in	2000	100000	[100, 400]	test losowy
slo4.in	6000	200000	[150, 700]	test losowy
slo5.in	15000	1000000	[200,2000]	test losowy
slo6.in	50000	10000000	[300,2000]	test losowy
slo7.in	70000	6000000	[300, 800]	test losowy
slo8.in	80000	10000000	[300,1200]	test losowy
slo9.in	100000	10000000	[600,1000]	test losowy
slo10.in	100000	10000000	[300, 600]	test losowy

Testy 1 i 2 zostały ułożone w całości ze specyficznych przypadków testowych, między innymi takich, które odrzucają wybrane rozwiązania błędne. Tylko te testy zalicza rozwiązanie nieoptymalne oparte na wyszukiwaniu wzorca (jego implementacje można znaleźć w plikach `slos1.cpp` i `slos3.pas`).

Każdy z testów 3–10 składa się z trzech przypadków specyficznych oraz 10 głównych, wśród których są przypadki z odpowiedzią pozytywną i negatywną. Testy główne były generowane następująco: generowano losowo ciąg o długości kilkakrotnie większej niż n reprezentujący słowo h_k, po czym wybierano z niego losowy fragment o długości bliskiej n i sumie bliskiej s. W przypadku żądania odpowiedzi pozytywnej po prostu wypisywano wybrany fragment, a w przypadku odpowiedzi negatywnej zmieniano wartość losowego elementu ciągu o ± 2, co prawie gwarantuje przejście do odpowiedzi negatywnej.

Jakub Wojtaszczyk
Treść zadania, Opracowanie

Paweł Parys
Program

Dostępna pamięć: 32 MB.

OI, Etap III, dzień pierwszy, 01.04.2009

Wyspa

Bajtazar jest królem Bajtocji, wyspy na Oceanie Szczęśliwości. Bajtocja ma kształt figury wypukłej, a wszystkie miasta w niej są położone nad brzegiem oceanu. Jedno z tych miast to Bajtogród, stolica Bajtocji. Każde dwa miasta są połączone drogą biegnącą przez wyspę po prostej łączącej oba miasta. Drogi łączące różne pary miast przecinają się, tworząc skrzyżowania.

Bitocy, konkurent Bajtazara do tronu, zaplanował nikczemny spisek. W trakcie podróży Bajtazara ze stolicy do **sąsiedniego** *miasta opanował Bajtogród. Bajtazar musi jak najszybciej powrócić do Bajtogrodu, by odzyskać władzę. Niestety niektóre z dróg są patrolowane przez zbrojne bojówki Bitocego. Bajtazar nie może poruszać się takimi drogami, choć może przekraczać je na skrzyżowaniach. Na trasie może on skręcać wyłącznie na skrzyżowaniach.*

Bajtazar wie, które drogi są patrolowane przez bojówki Bitocego. Poprosił Cię o wyznaczenie najkrótszej bezpiecznej trasy z miasta, w którym się aktualnie znajduje, do Bajtogrodu.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite n i m, oddzielone pojedynczym odstępem ($3 \leqslant n \leqslant 100\,000$, $1 \leqslant m \leqslant 1\,000\,000$), oznaczające odpowiednio: liczbę miast oraz liczbę dróg patrolowanych przez bojówki Bitocego. Ponumerujmy miasta od 1 do n, zaczynając od Bajtogrodu i idąc wzdłuż brzegu zgodnie z ruchem wskazówek zegara. Bajtazar znajduje się w mieście nr n. W każdym z kolejnych n wierszy znajduje się para liczb x_i i y_i ($-1\,000\,000 \leqslant x_i, y_i \leqslant 1\,000\,000$), oddzielonych pojedynczym odstępem, oznaczających współrzędne miasta nr i na wyspie.

W każdym z kolejnych m wierszy znajduje się para liczb a_j i b_j ($1 \leqslant a_j < b_j \leqslant n$). Para taka oznacza, że droga łącząca miasta a_j i b_j jest patrolowana przez bojówki Bitocego. Pary liczb opisujące patrolowane drogi nie powtarzają się. Możesz założyć, że dla każdych danych testowych Bajtazar może dostać się do Bajtogrodu.

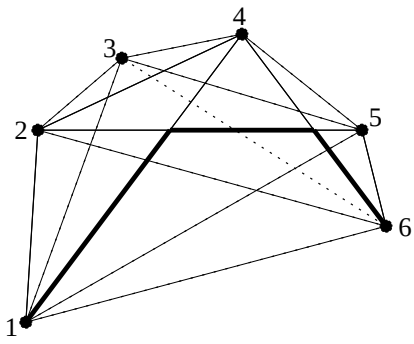
Wyjście

Twój program powinien wypisać na standardowe wyjście jedną liczbę zmiennopozycyjną: długość najkrótszej bezpiecznej trasy prowadzącej z miasta n do Bajtogrodu. Wynik Twojego programu może różnić się od poprawnego o co najwyżej 10^{-5}.

Przykład

Dla danych wejściowych:

```
6 9
-12 -10
-11 6
-4 12
6 14
16 6
18 -2
3 4
1 5
2 6
2 3
4 5
3 5
1 3
3 6
1 6
```



poprawnym wynikiem jest:

```
42.0000000000
```

Trasa, którą powinien podążać Bajtazar, wychodzi z miasta 6 w kierunku miasta 4, następnie biegnie drogą łączącą miasta 2 i 5, a na koniec drogą łączącą Bajtogród z miastem 4.

Rozwiązanie

Wprowadzenie

Przez M_i będziemy oznaczać miasto o numerze i. Jako że wszystkie drogi są dwukierunkowe, znalezienie najkrótszej trasy z M_n do Bajtogrodu (czyli M_1) jest równoważne znalezieniu najkrótszej drogi z M_1 do M_n. Drogę, na której nie ma bojówek Bitocego (czyli taką, którą może poruszać się Bajtazar), będziemy nazywać *przejezdną*.

Zaczniemy od zrozumienia, jak wygląda szukana najkrótsza trasa.

Lemat 1. Najkrótsza bezpieczna trasa z M_1 do M_n wraz z odcinkiem $[M_n, M_1]$ stanowi brzeg wielokąta wypukłego (być może zdegenerowanego do odcinka, jeśli droga $(1,n)$ jest przejezdna), którego wnętrza nie przecina żadna przejezdna droga.

Dowód: Najkrótsza bezpieczna trasa oczywiście nie może mieć samoprzecięć, a zatem wraz z odcinkiem $[M_1, M_n]$ ogranicza pewien wielokąt prosty. Żadna droga nie przecina odcinka $[M_1, M_n]$ poza końcami (bo pomiędzy M_1 a M_n na brzegu wyspy nie ma miast), zatem jeśli jakakolwiek przejezdna droga D przecina wnętrze tego wielokąta, to przynajmniej dwukrotnie przecina najkrótszą trasę. Wobec tego, zastępując odcinek trasy pomiędzy pierwszym i ostatnim punktem przecięcia odpowiednim fragmentem drogi D, otrzymamy drogę krótszą. To pokazuje, że żadna przejezdna droga nie przecina naszego wielokąta. Gdyby ten wielokąt nie był wypukły, to wierzchołek z kątem wklęsłym znajdowałby się na

pewnym skrzyżowaniu dróg (bo wszystkie drogi wychodzące z jednego miasta są zawarte w kącie wypukłym). Ale wtedy pozostałe fragmenty dwóch dróg krzyżujących się na tym skrzyżowaniu znajdowałyby się wewnątrz wielokąta, a więc przecinałyby go — wbrew temu, co już udowodniliśmy. ■

Lemat 1 charakteryzuje najkrótszą trasę w sposób jednoznaczny, bowiem dla dwóch różnych wielokątów wypukłych, które mają wspólny odcinek brzegowy i leżą po tej samej stronie prostej zawierającej ten odcinek, jeden z nich musi zawierać we wnętrzu fragment brzegu drugiego (rysunkowe uzasadnienie tego stwierdzenia pozostawiamy Czytelnikowi).

Rozwiązania nieoptymalne

Zacznijmy od rozważenia najprostszego rozwiązania. W zadaniu występuje problem znalezienia najkrótszej ścieżki między dwoma wierzchołkami w grafie, co przywodzi na myśl algorytm Dijkstry (patrz np. [20]). Zastanówmy się jednak, jaki graf musielibyśmy rozważyć. Otóż wierzchołkami naszego grafu musiałyby być nie tylko miasta, ale także skrzyżowania dróg (bo tam również możemy skręcać). Jednak skrzyżowań przekątnych w wielokącie jest $O(n^4)$ (każda czwórka wierzchołków wielokąta wyznacza jedno przecięcie przekątnych), czyli stanowczo za dużo, by stosować taki algorytm — niezależnie od złożoności czasowej, brak nam pamięci komputera, by choćby przechowywać wierzchołki takiego grafu.

W następnych podejściach spróbujemy skorzystać z udowodnionego wyżej lematu. Jeżeli droga $(1,n)$ jest przejezdna, to oczywiście jest najkrótszą trasą pomiędzy miastami 1 i n; dalej będziemy zakładać, że Bitocy (całkiem rozsądnie) postanowił jednak patrolować tę drogę. Rozważmy dowolną inną przejezdną drogę D. Prosta k zawierająca D nie przecina wnętrza odcinka (M_1,M_n), bo pomiędzy M_1 a M_n nie leży żadne miasto. Zatem miasta M_1 i M_n leżą po tej samej stronie k i przynajmniej jedno z nich nie leży na k (bo D jest różna od $(1,n)$). Zauważmy, że z lematu wynika, iż optymalna trasa cała przebiega przez tę półpłaszczyznę ograniczoną przez k, w której leżą miasta M_1 i M_n — optymalna trasa nie może przecinać (tzn. przechodzić na drugą stronę) odcinka D na mocy lematu oraz nie może przecinać k poza D, gdyż nie może wyjść poza wyspę. Z drugiej strony, jeżeli rozważymy przecięcie wszystkich półpłaszczyzn wyznaczonych przez wszystkie przejezdne drogi D, to otrzymamy wielokąt wypukły (bo przecinamy zbiory wypukłe), którego wnętrza nie przecina żadna przejezdna droga (bo każda przejezdna droga leży w brzegu pewnej półpłaszczyzny), a zatem jego brzeg będzie szukaną optymalną trasą.

Zadanie znalezienia obszaru będącego przecięciem zbioru zadanych półpłaszczyzn jest bardzo zbliżone do problemu znalezienia otoczki wypukłej zbioru punktów (o tym problemie można poczytać w większości książek o algorytmach, w szczególności w [20]). Jeśli przez q oznaczymy liczbę półpłaszczyzn, to algorytm rozwiązujący ten problem (opisany dokładniej dalej) działa w złożoności czasowej $O(q \log q)$ — czyli w naszym wypadku $O(n^2 \log n)$, tzn. dużo lepszej, ale ciągle niesatysfakcjonującej. Aby poprawić złożoność, będziemy musieli podjąć wysiłek rozważania tylko części dróg i odrzucenia tych, o których umiemy zagwarantować, że nie przebiega nimi optymalna trasa.

Rozwiązanie wzorcowe

Stoi przed nami zadanie efektywnego wybrania części przejezdnych dróg tak, by nie rozpatrywać ich wszystkich. Zauważmy jednak, że jeśli dla ustalonej przejezdnej drogi (a,b), przy czym $a < b$, istnieje taka przejezdna droga (a,b'), że $b < b'$, to dowolna trasa idąca kawałek drogą (a,b) przecina drogę (a,b'), a zatem na mocy lematu nie może być najkrótsza. Wystarczy zatem rozważać takie przejezdne drogi (a,b), że dla dowolnego $b' > b$ droga (a,b') jest patrolowana. Drogi o tej własności nazwiemy *potrzebnymi*. Oczywiście z każdego miasta wychodzi co najwyżej jedna droga potrzebna, a zatem dróg potrzebnych jest w sumie $O(n)$.

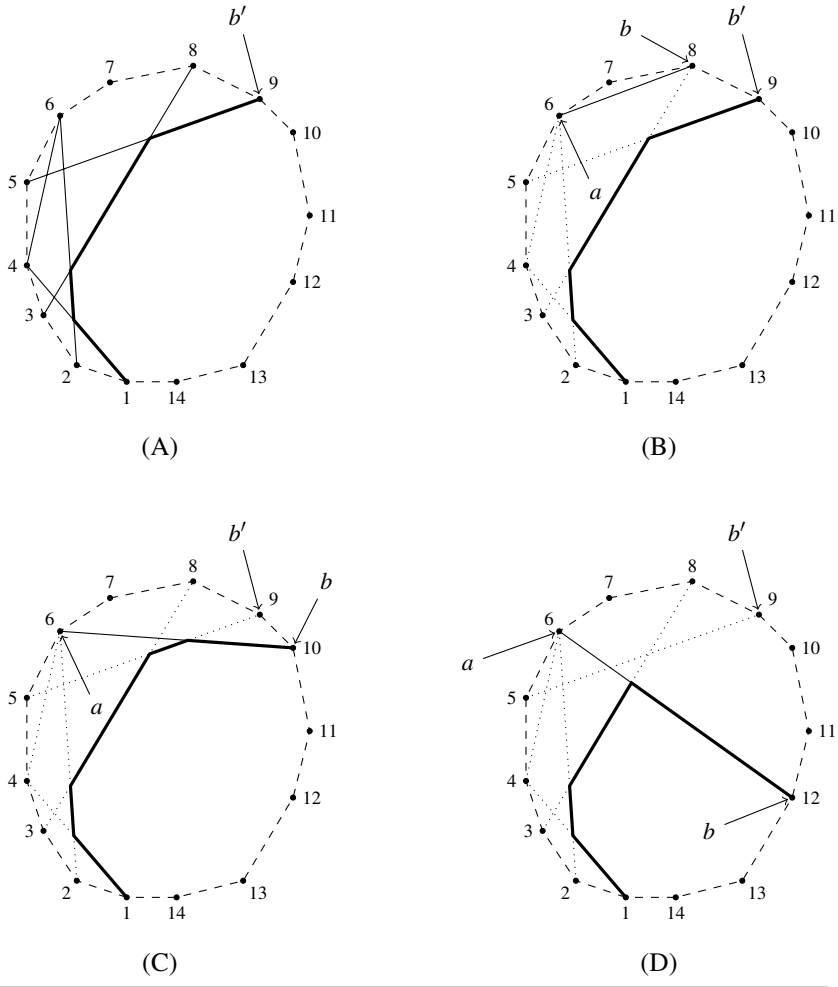
Zastanówmy się nad efektywną implementacją wyszukiwania dróg potrzebnych. Musimy stworzyć tablicę, w której dla każdego miasta a będzie podany numer miasta b, do którego prowadzi potrzebna droga z a (a jeżeli z a nie prowadzi żadna przejezdna droga, to droga potrzebna z a nie istnieje). Chcemy dla każdego miasta a przeglądać wszystkie drogi (a,b) w kolejności od największego b i znaleźć pierwszą, która jest przejezdna. Wystarczy więc wrzucić wszystkie drogi patrolowane wychodzące z miasta a do miast o większym numerze do tablicy, posortować w porządku malejącym numerów miast docelowych i przeglądać w poszukiwaniu pierwszej drogi, której brakuje. Wszystkich dróg patrolowanych jest co najwyżej m, zatem łączny czas wyszukiwania dróg potrzebnych to $O((m+n)\log n)$.

Należy tu zwrócić uwagę na sposób przechowywania dróg patrolowanych. Jako że z każdego konkretnego miasta może wychodzić $O(n)$ dróg patrolowanych, to tablica rozmiaru $O(n^2)$ nie zmieści się w pamięci. Można użyć list (które się trudniej sortuje) albo, w Javie i w C++, tablic dynamicznych o zmiennym rozmiarze (vectorów). Można też po prostu wszystkie drogi patrolowane wczytać do jednej tablicy rozmiaru $O(m)$, ewentualnie zamieniając końce tak, by numer miasta startowego był mniejszy niż numer docelowego, a następnie posortować tę tablicę leksykograficznie — w kolejności rosnącej numerów miast startowych, rozstrzygając remisy na korzyść większych numerów miast docelowych. Wtedy przeglądając tę tablicę, wyznaczymy drogi potrzebne dla wszystkich miast po kolei.

Mając tablicę dróg potrzebnych, będziemy konstruować trasę, której nie przecinają żadne inne drogi. Opiszemy tu precyzyjnie stosowany algorytm. Będziemy rozważać drogi potrzebne w kolejności wyznaczonej przez ich miasta początkowe: najpierw zaczynająca się w mieście 1, potem w 2, itd. Tuż po przetworzeniu drogi (a,b) chcemy zawsze mieć obliczoną trasę z miasta 1 do pewnego miasta $b' \geqslant b$, której nie przecina żadna z dotychczas przetworzonych potrzebnych dróg i która jest wypukła (formalnie: po dołożeniu drogi $(b',1)$ stanowi brzeg wielokąta wypukłego). Przynajmniej jedna potrzebna droga musi kończyć się w mieście n (w przeciwnym wypadku nie da się w ogóle dojść do miasta n), zatem po przetworzeniu wszystkich potrzebnych dróg dostajemy rozwiązanie. Pierwszą potrzebną drogę bardzo łatwo przetworzyć, musi to być droga $(1,b)$ (taka musi istnieć, bo jeśli nie istnieje droga potrzebna z miasta 1, to wszystkie drogi z niego są patrolowane i wbrew założeniom zadania nie istnieje trasa, którą może przejść Bajtazar), więc na początku trasa składa się tylko z niej.

Przypuśćmy, że mamy skonstruowaną trasę do miasta b' i chcemy przetworzyć kolejną potrzebną drogę (a,b). Wiemy, że $a > a'$, przy czym (a',b') jest drogą, której fragment został wykorzystany, by dojść do b'. Jeśli $b \leqslant b'$, to drogę (a,b) możemy odrzucić (gdyż nie przecina drogi (a',b'), a zatem z wypukłości nie przecina też dotychczas skonstruowanej trasy). W przeciwnym razie szukamy punktu jej przecięcia C z dotychczasową trasą. Do

punktu przecięcia C idziemy dotychczasową trasą, a dalej drogą (a,b). Otrzymany wielokąt jest wypukły (jest przecięciem figur wypukłych) i nie przecina go żadna z dotychczas skonstruowanych dróg (wykorzystany fragment drogi (a,b) przebiega wewnątrz starego wielokąta, a więc żadna z uprzednio przejrzanych dróg go nie przecina). Punkt przecięcia efektywnie znajdujemy poprzez umieszczenie fragmentów dróg, z których składa się dotychczasowa trasa, na stosie. Gdy przetwarzamy nową drogę (a,b), zdejmujemy od wierzchołka stosu odcinki do momentu trafienia na taki odcinek $[P,Q]$, który przecina się z $[M_a,M_b]$ w pewnym punkcie C, a następnie umieszczamy na stosie odcinki $[P,C]$ i $[C,M_b]$.



Rys. 1: Kilka początkowych kroków algorytmu konstrukcji trasy Bajtazara. (A) Ciągłe kreski reprezentują drogi potrzebne wychodzące z miast od 1 do 5, natomiast skonstruowany fragment trasy, prowadzący do miasta $b' = 9$, jest oznaczony pogrubioną linią. (B) Jeżeli droga potrzebna z miasta $a = 6$ prowadzi do miasta $b = 8$, to trasa nie ulega zmianie (jako że $b \leqslant b'$). (C) Jeżeli drogą potrzebną jest (6,10), to przecina ona dotychczasową trasę w jej ostatnim odcinku. (D) Jeżeli natomiast jest nią (6,12), to przecięcie następuje na przedostatnim odcinku dotychczasowej trasy, więc ostatni odcinek trasy zostaje zdjęty ze stosu.

Żeby sprawdzić, czy dane dwa odcinki $[P,Q]$ i $[M_a,M_b]$ przecinają się, a jeżeli tak, wyznaczyć ich punkt przecięcia, można np. zapisać je w postaci parametrycznej:

$$\{(1-s)\cdot P+s\cdot Q:\ s\in[0,1]\}\quad\text{oraz}\quad\{(1-t)\cdot M_a+t\cdot M_b:\ t\in[0,1]\}$$

i rozwiązać układ dwóch równań liniowych (osobno po współrzędnej x i y) z niewiadomymi s i t:

$$(1-s)\cdot P+s\cdot Q=(1-t)\cdot M_a+t\cdot M_b.$$

Przy przetwarzaniu pojedynczej drogi włożymy na stos co najwyżej jeden nowy odcinek, więc w czasie działania całego algorytmu włożymy $O(n)$ odcinków, a zatem i zdejmiemy najwyżej $O(n)$ odcinków. Zatem ta część algorytmu działa w czasie $O(n)$. Rozwiązanie takie zostało zaimplementowane w plikach `wys.cpp`, `wys1.c`, `wys5.pas`, `wys6.java`, `wys7.ml` (działające na liczbach zmiennoprzecinkowych) oraz w pliku `wys3.cpp` (implementacja używająca wyłącznie liczb całkowitych, z własną arytmetyką ułamków).

Rozwiązanie alternatywne

Rozwiązanie to opiera się na spostrzeżeniu, że możemy jeszcze bardziej ograniczyć liczbę rozważanych dróg. Zauważmy, że jeśli potrzebna droga (a,b) ma zostać wykorzystana w trasie docelowej, to w szczególności wszystkie drogi (a',b) dla $a'<a$ muszą być patrolowane — czyli wykorzystamy co najwyżej jedną drogę wiodącą do danego b (tę z miasta a o najmniejszym numerze). Możemy zatem, konstruując drogi potrzebne z każdego a, trzymać dodatkowo tablicę indeksowaną miastami docelowymi, wskazującą, czy już jakaś potrzebna droga prowadzi do tego miasta. Jeżeli tak, to droga, którą właśnie skonstruowaliśmy, na pewno nie będzie nigdy wykorzystana (bo przeglądamy miasta wyjściowe w kolejności rosnącej), czyli nie trzeba jej dodawać.

Zauważmy, że żeby została skonstruowana droga potrzebna z miasta a do miasta b, na wejściu muszą być podane (patrolowane) drogi (a,b') dla wszystkich $b'>b$. Co więcej, każda skonstruowana droga potrzebna prowadzi do innego miasta b. Wobec tego, aby skonstruować k dróg potrzebnych, na wejściu musi znaleźć się co najmniej $0+1+\ldots+(k-1)=k\cdot(k-1)/2=\Omega(k^2)$ dróg. Wobec tego skonstruujemy co najwyżej $O(\sqrt{m})$ dróg potrzebnych.

W takim razie przy konstrukcji najkrótszej trasy nie musimy działać sprytnie, wystarczy stworzyć ją w złożoności czasowej $O(k^2)$. Dla każdej pary skonstruowanych dróg wyznaczamy ich punkt przecięcia (o ile się przecinają). Następnie dla aktualnej drogi wybieramy punkt przecięcia z tą z kolejnych skonstruowanych dróg, który znajduje się najbliżej jej początku, i w tę drogę potrzebną skręcamy. Rozwiązanie to zostało zaimplementowane w pliku `wys2.cpp`.

Testy

Poprawność rozwiązań badało 14 testów połączonych w 10 zestawów.

Pierwsze pięć testów to testy pseudolosowe. Zawierają stosunkowo małe wielokąty (tak by Bitocy mógł patrolować nawet wszystkie drogi, mieszcząc się w górnym limicie). W tych testach drogi potrzebne są stosunkowo krótkie, a wynikowa trasa zawiera stosunkowo wiele

dróg. W tym celu jako przejezdne wybrane zostały losowe krótkie drogi, wszystkie długie drogi są patrolowane.

Druga grupa testów, również pseudolosowych, zawiera duże wielokąty, w których patrolowane są losowe drogi wychodzące z miast o stosunkowo małych numerach i prowadzące do miast o stosunkowo dużych numerach (ze względu na ograniczenia z zadania, siłą rzeczy, procent dróg patrolowanych musi tu być niewielki).

W zestawie znajdują się dwa testy w pełni losowe (tzn. takie, w których każda droga jest patrolowana z równym prawdopodobieństwem), co dla dużych wielokątów powoduje, że najkrótsza trasa jest zdecydowanie nieskomplikowana.

Do tego jest jeden test badający dokładność obliczeń, w którym występują dwie drogi prawie równoległe (a więc stosunkowo niewielki kątowo błąd w wyznaczaniu punktu przecięcia istotnie zmienia długość trasy), oraz test skrajny, w którym dozwolone jest poruszanie się wyłącznie po brzegu wielokąta.

Poniższa tabelka zawiera statystyki poszczególnych testów, w których k to liczba dróg potrzebnych skonstruowanych przez algorytm alternatywny, a l to liczba dróg na najkrótszej trasie.

Nazwa	n	m	k	l	Opis
wys1.in	100	4 772	41	26	mały test pseudolosowy
wys2.in	200	18 472	120	49	mały test pseudolosowy
wys3.in	500	120 793	289	99	mały test pseudolosowy
wys4.in	1 000	494 995	920	518	mały test pseudolosowy
wys5.in	1 412	982 600	420	108	mały test pseudolosowy
wys6.in	10 000	1 000 000	1 185	26	duży test pseudolosowy
wys7a.in	20 000	1 000 000	1 299	23	duży test pseudolosowy
wys7b.in	4	4	2	2	test dokładnościowy
wys8a.in	30 000	1 000 000	1 287	22	duży test pseudolosowy
wys8b.in	1 416	1 000 000	6	4	test w pełni losowy
wys9a.in	40 000	1 000 000	1 235	14	duży test pseudolosowy
wys9b.in	55 944	1 000 000	1	1	test w pełni losowy
wys10a.in	50 000	1 000 000	584	2	duży test pseudolosowy
wys10b.in	1 412	994 755	1 411	1 411	test skrajny

Marek Biskup
Treść zadania, Opracowanie

Piotr Niedźwiedź
Program

Dostępna pamięć: 64 MB.

OI, Etap III, dzień drugi, 02.04.2009

Kod

Bajtocki Instytut Telekomunikacyjny (BIT) zajmuje się ustalaniem standardów przesyłania danych w sieciach telekomunikacyjnych Bajtocji. Bajtazar, jeden z informatyków pracujących w BIT, zajmuje się **kodami prefiksowymi** *— specjalnym sposobem reprezentowania znaków. Każdemu znakowi bajtockiego alfabetu odpowiada pewien ciąg bitów, nazywany kodem tego znaku. Kody znaków mają następujące własności:*

- *Kod żadnego znaku nie jest prefiksem (tj. początkowym fragmentem) kodu żadnego innego znaku. Na przykład, jeśli* `010010` *jest kodem litery* `A`*, to żaden z ciągów bitów:* `0`*,* `01`*,* `010`*,* `0100` *ani* `01001` *nie może być kodem żadnego znaku. Podobnie,* `0100100`*,* `0100101` *oraz dłuższe ciągi zaczynające się od* `010010` *nie mogą być kodami znaków.*
- *Jeśli dany ciąg bitów* w *jest prefiksem kodu pewnego znaku, ale nie jest całym kodem, to ciągi bitów postaci* w`0` *i* w`1` *(czyli* w *z dopisanym na końcu zerem lub jedynką) są prefiksami kodów (lub całymi kodami) pewnych znaków. Na przykład, jeśli* `0100` *jest prefiksem kodu litery* `A`*, to* `01000` *oraz* `01001` *muszą być prefiksami kodów (lub kodami) pewnych znaków.*

Rozważmy następujący, przykładowy kod prefiksowy dla alfabetu złożonego ze znaków `A`*,* `B`*,* `C`*,* `D` *i* `E`*:*

znak	kod znaku
A	00
B	10
C	11
D	010
E	011

Kodowanie ciągu znaków za pomocą kodu prefiksowego polega na połączeniu kodów jego kolejnych znaków. Na przykład zakodowany ciąg `BACAEBABAE` *ma postać* `10001100011100010000011`.

Bajtazar zauważył, że jeśli pewna liczba początkowych bitów zostanie utracona, to zakodowana informacja może być odkodowana niepoprawnie albo wcale. Na przykład, jeśli usuniemy pięć pierwszych bitów z ciągu podanego powyżej, to powstały ciąg `10001110001000011` *zostanie odkodowany jako* `BACBABAE`*. Pięć ostatnich liter (*`BABAE`*) jest poprawnych, jednak trzy pierwsze (*`BAC`*) nie. Bajtazar zauważył, że wszystkie litery po pierwszym* `E` *zostały odkodowane poprawnie. Doszedł do wniosku, że zawsze, jeśli żadne bity kodu* `E` *nie zostaną utracone, to wszystkie kolejne znaki za* `E` *zostaną odkodowane poprawnie. Tak będzie dla dowolnego kodowanego ciągu znaków, w którym występuje* `E`*. Zauważył on, że litera* `D` *też ma tę własność, ale litery* `A`*,* `B` *i* `C` *tej własności nie mają.*

Opisaną własność kodów znaków `E` *i* `D` *Bajtazar określił jako bycie* **kodem synchronizującym**. *Bajtazar powierzył Ci zadanie napisania programu znajdującego, dla danego kodu prefiksowego, wszystkich kodów synchronizujących. Aby zaoszczędzić czas, wymyślił sobie, że pokaże Ci wszystkie kody znaków na swoim binarnym monitorze. Monitor ma cztery przyciski:*

- `0` — *dopisz 0*
- `1` — *dopisz 1*
- `B` — *backspace, usuwa ostatnio dopisaną cyfrę*
- `X` — *po naciśnięciu tego przycisku monitor wydaje charakterystyczny dźwięk „beep”.*

Na początku wyświetlacz jest pusty; Bajtazar kolejno naciska przyciski i gdy na monitorze pojawia się kod kolejnego znaku, Bajtazar naciska przycisk `X`*. Po pokazaniu ostatniego znaku Bajtazar czyści ekran (naciskając odpowiednią liczbę razy przycisk* `B`*). Wiesz, że Bajtazar pokaże Ci cały kod prefiksowy, naciskając najmniejszą możliwą dla tego kodu liczbę przycisków.*

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita n *(*$6 \leqslant n \leqslant 3\,000\,000$*) oznaczająca liczbę przycisków naciśniętych przez Bajtazara. W kolejnym wierszu znajduje się* n*-literowy napis złożony ze znaków '*`0`*', '*`1`*', '*`B`*' oraz '*`X`*', oznaczających poszczególne przyciski. Każde naciśnięcie przycisku* `X` *oznacza kolejny znak (znaki numerujemy od 1). Suma długości wszystkich kodów nie przekroczy* 10^8*.*

Wyjście

W pierwszym wierszu standardowego wyjścia należy wypisać liczbę k *— liczbę kodów synchronizujących. W kolejnych* k *wierszach należy wypisać, w porządku rosnącym, numery znaków, które są kodami synchronizującymi, po jednym w wierszu. Jeżeli dla danego kodu prefiksowego nie ma żadnych kodów synchronizujących, to należy wypisać tylko jeden wiersz zawierający liczbę 0.*

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`21`	`2`
`11XB0XBB00XB11XB0XBBB`	`4`
	`5`

Wyjaśnienie przykładu

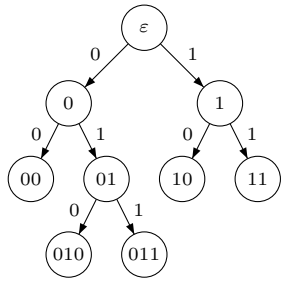
Oto kolejne kody, które pojawiają się na monitorze Bajtazara: `11`, `10`, `00`, `011`, `010`*. Kody* `011` *i* `010` *są kodami synchronizującymi.*

Rozwiązanie

Wczytanie kodu i jego reprezentacja

Na początku zajmiemy się wczytywaniem kodu do pamięci komputera. Jak się za chwilę przekonamy, sposób pokazywania kodu przez Bajtazara jest prosty i wręcz ułatwia jego wczytywanie.

Naturalnym sposobem reprezentacji kodu prefiksowego jest *właściwe* drzewo binarne, czyli takie, w którym każdy wierzchołek w drzewie ma dwóch synów lub jest liściem. Krawędzie do lewego syna są etykietowane wartością `0`, a do prawego syna — wartością `1`. Każdemu wierzchołkowi w takim drzewie odpowiada ciąg binarny złożony z etykiet na drodze od korzenia do tego wierzchołka. Ciąg ten będzie *etykietą* wierzchołka.



Rys. 1: Właściwe drzewo binarne.

Przykładowe drzewo binarne przedstawione jest na rysunku 1. Etykiety liści odpowiadają kodom znaków (czyli tzw. słowom kodowym) z przykładu z treści zadania. Etykiety wierzchołków wewnętrznych odpowiadają prefiksom właściwym (tzn. początkowym fragmentom krótszym niż całe słowo) słów kodowych. Etykieta korzenia to słowo puste, które oznaczone jest przez ε. Okazuje się, że takie drzewo można stworzyć dla każdego kodu prefiksowego, co wynika z dwóch własności kodów prefiksowych opisanych w treści zadania:

- Własność pierwsza mówi, że kod żadnego znaku nie jest prefiksem kodu innego znaku. W reprezentacji drzewowej znaczy to tyle, że wierzchołek wewnętrzny nie może być słowem kodowym.
- Własność druga mówi, że jeśli w jest właściwym prefiksem słowa kodowego to $w\texttt{0}$ oraz $w\texttt{1}$ są prefiksami (właściwymi lub nie) słowa kodowego. Dla drzewa oznacza to, że wierzchołek wewnętrzny ma zawsze dwóch synów, czyli że drzewo jest właściwe.

W tym miejscu warto przytoczyć jedną ważną własność właściwych drzew binarnych, która będzie nam potrzebna w dalszej części opisu, a której prosty dowód (np. przez indukcję względem rozmiaru drzewa) pozostawiamy Czytelnikowi:

Obserwacja 1. Właściwe drzewo binarne o ℓ liściach ma $2\ell - 1$ wierzchołków.

Reprezentacja danych wejściowych umożliwia łatwą konstrukcję drzewa. W tym celu potrzebny nam będzie licznik m przechowujący liczbę odczytanych już słów kodowych. Zaczynamy z pojedynczym wierzchołkiem — korzeniem drzewa — i inicjalizujemy licznik na

0. Następnie czytamy kolejne znaki z wejścia i konstruujemy drzewo, wykonując operacje zależne od przeczytanego znaku.

`0`: Tworzymy nowy wierzchołek, łączymy go z aktualnym wierzchołkiem krawędzią o etykiecie `0` (lewa krawędź) i przechodzimy do nowego wierzchołka.

`1`: Tworzymy nowy wierzchołek, łączymy go z aktualnym wierzchołkiem krawędzią o etykiecie `1` (prawa krawędź) i przechodzimy do nowego wierzchołka.

`B`: Przechodzimy do ojca aktualnego wierzchołka.

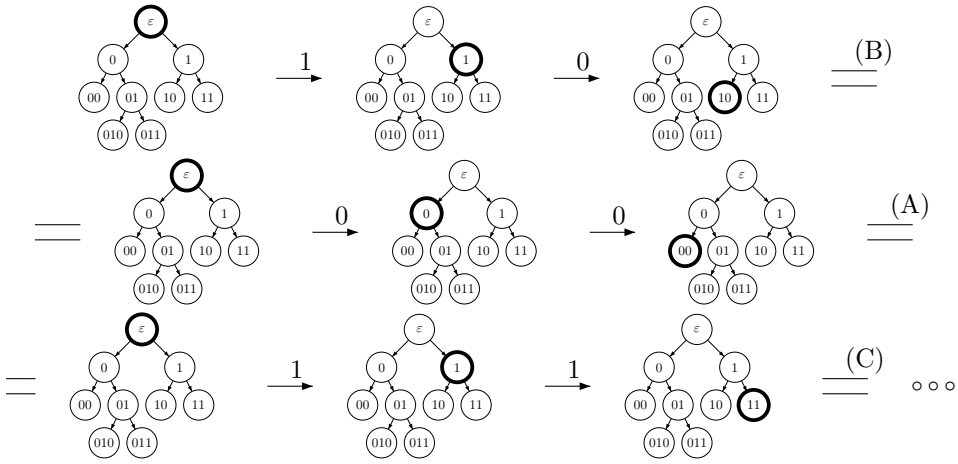
`X`: Zwiększamy m o jeden i zapisujemy nową wartość m w aktualnym wierzchołku.

W każdym momencie etykieta aktualnego wierzchołka odpowiada ciągowi aktualnie wyświetlanemu na monitorze. Algorytm ten można zrealizować rekurencyjnie:

```
1:  m := 0;
2:  procedure KONSTRUUJ(q)
3:  begin
4:    while true do
5:      c := KOLEJNYZNAKNAWEJŚCIU();
6:      if c = 0 then
7:        q.left := NOWYWIERZCHOŁEK();
8:        KONSTRUUJ(q.left);
9:      else if c = 1 then
10:       q.right := NOWYWIERZCHOŁEK();
11:       KONSTRUUJ(q.right);
12:     else if c = X then
13:       m := m + 1;
14:       q.number := m;
15:     else { (c = B) lub koniec wejścia }
16:       return;
17: end
18: { Wczytanie drzewa: }
19: Pomiń pierwszy wiersz wejścia;
20: r := NOWYWIERZCHOŁEK();
21: KONSTRUUJ(r);
22: return r;
```

Pierwszy wiersz jest pomijany, gdyż informacja o liczbie znaków w opisie drzewa nie jest nam potrzebna. Z warunku, że Bajtazar czyści ekran, wynika, że program wróci z wywołań rekurencyjnych funkcji KONSTRUUJ po przeczytaniu literek `B`. Powrót z głównego wywołania funkcji KONSTRUUJ będzie rezultatem braku kolejnych znaków na wejściu.

Drzewa binarnego można łatwo użyć do dekodowania zakodowanego ciągu. Na starcie „umieszczamy” dekoder w korzeniu drzewa. Następnie czytamy kolejne bity zakodowanego ciągu, schodząc w dół drzewa po krawędziach odpowiadających przeczytanym bitom. Po dotarciu do liścia wypisujemy literę odpowiadającą etykiecie tego liścia i wracamy do korzenia.



Rys. 2: Dekodowanie ciągu bitów. Etykiety nad strzałkami oznaczają kolejne czytane bity, a symbole w nawiasach — odkodowane znaki.

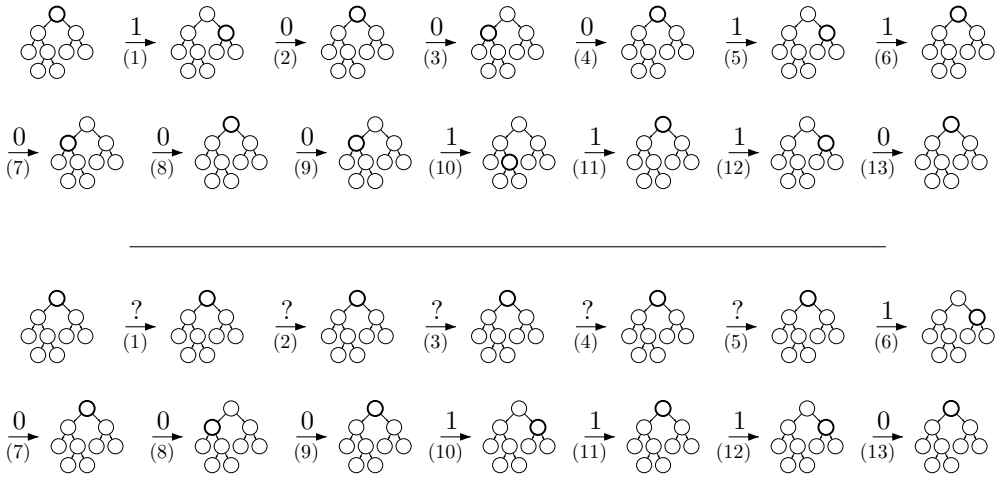
Dla zakodowanego ciągu z treści zadania — `10001100011110001000011` — dekodowanie przebiega następująco (rysunek 2):

1. Startujemy w korzeniu: $q = \varepsilon$ (q oznaczać będzie aktualną pozycję dekodera, czyli jego *stan*, a ε to korzeń drzewa).
2. Czytamy `1` i schodzimy w prawo ($q = $ `1`).
3. Czytamy `0` i schodzimy w lewo ($q = $ `10`). Dotarliśmy do liścia `10`, więc odkodowujemy literę `B`. Następnie przenosimy dekoder do korzenia ($q = \varepsilon$) i kontynuujemy.
4. Czytamy `0` i schodzimy w lewo ($q = $ `0`).
5. Czytamy `0` i schodzimy w lewo ($q = $ `00`). Odkodowujemy `A` i kontynuujemy z korzenia.
6. ...

Rozważmy dekoder umieszczony w pewnym wierzchołku q_1. Po przeczytaniu słowa w trafi on do pewnego wierzchołka q_2. Zakładamy przy tym, że q_2 nie jest liściem, bo — jak pamiętamy — dekoder z liścia trafia natychmiastowo do korzenia (czyli przyjmujemy wtedy $q_2 = \varepsilon$). W tym przypadku powiemy, że dekoder *przechodzi* z q_1 do q_2 po słowie w. Oznaczmy takie przejście $\delta(q_1, w) = q_2$. Zauważmy, że δ jest funkcją ze zbioru par (wierzchołek wewnętrzny, słowo) w zbiór wierzchołków wewnętrznych. Dla każdego słowa kodowego w mamy $\delta(\varepsilon, w) = \varepsilon$, tzn. słowa kodowe zawsze prowadzą z korzenia do korzenia.

Odkodowanie ciągu z pominięciem prefiksu

Jak zachowa się dekoder, jeśli pewien początkowy fragment zakodowanej wiadomości zostanie utracony? Aby odpowiedzieć na to pytanie, porównajmy pracę dekodera na pełnych



Rys. 3: Przejścia dla dekodera zaczynającego od pierwszego bitu (na górze) i od szóstego bitu (na dole). Na strzałkach zaznaczona jest wartość aktualnie czytanego bitu (nad strzałką) oraz jego numer (pod strzałką). Pierwsze pięć bitów dolnego dekodera jest oznaczone znakiem zapytania, bo dekoder te bity pomija. Z rysunku wynika, że synchronizacja dolnego dekodera następuje po przeczytaniu jedenastego bitu.

danych (oznaczany jako dekoder D_0) oraz na danych bez pięciu początkowych bitów (dekoder D_5), używając przykładu z treści zadania.

Rysunek 3 pokazuje działanie dekodera D_0 (na górze) oraz D_5 (na dole). Dekodery w liściach zostały na tym rysunku pominięte, gdyż trafiają one natychmiast do korzenia drzewa (porównaj z rysunkiem 2). Pominięte zostały też odkodowywane znaki. Dekoder D_5 opuszcza pięć początkowych bitów, więc bity te zostały na rysunku oznaczone znakiem zapytania.

Widzimy, że po bitach 5-10 stany dekoderów są różne. Po jedenastym bicie stany dekoderów stają się identyczne. Dekodery odkodowały jednak w tym momencie różne litery, ponieważ ich poprzednie stany były różne. Następnie dekodery czytają taki sam zakodowany ciąg. Biorąc pod uwagę, że stan dekoderów po 11. bicie jest taki sam oraz że dekodery podlegają takim samym regułom przejścia, wnioskujemy, że ich kolejne stany muszą być identyczne. Odkodują one zatem identyczne kolejne litery.

W treści zadania podano, że synchronizacja wiąże się w tym przypadku z wystąpieniem litery `E`. W dodatku każde wystąpienie litery `E` synchronizuje dekoder, niezależnie od tego, jaka wiadomość jest przesyłana oraz jaki prefiks tej wiadomości został pominięty przy dekodowaniu. Synchronizacja polega na tym, że po odczytaniu kodu `E` stan dekodera D_k, czyli dekodera pomijającego pierwszych k bitów, staje się identyczny ze stanem dekodera D_0, zaczynającego wiadomość od początku. Tuż po odczytaniu `E` dekoder D_0 będzie w stanie ε, gdyż po przeczytaniu pełnego słowa kodowego dekoder zawsze trafia do tego stanu (porównaj z rysunkiem 2). Dekoder D_k musi więc po przeczytaniu `E` trafić także do stanu ε.

Popatrzmy, w jakim stanie może znaleźć się dekoder D_k *tuż przed* przeczytaniem litery `E`. Fragment poprzedzający literę `E` może być dowolnym ciągiem słów kodowych. (Nie znaczy to, że jest to dowolny ciąg bitów, np. `000` nie jest ciągiem słów kodowych!) Dowolny prefiks (początkowy fragment) tego ciągu może zostać utracony. Dekoder zawsze zaczyna w stanie

ε. Zatem dekoder tuż przed przeczytaniem `E` może być w dowolnym stanie, do którego da się przejść z korzenia po sufiksie (końcowym fragmencie) ciągu słów kodowych. Na potrzeby tego zadania nazwijmy zbiór takich wierzchołków zbiorem wierzchołków *ważnych*. Reasumując:

Definicja 1. Zbiór wierzchołków *ważnych* dla drzewa kodu prefiksowego to zbiór wierzchołków wewnętrznych drzewa, do których da się dojść z korzenia po pewnym sufiksie pewnego ciągu słów kodowych.

Nasze zadanie polega na znalezieniu kodów synchronizujących, czyli takich, że dekoder po napotkaniu takiego kodu zawsze dobrze odczyta kolejne znaki. W świetle analizy przedstawionej powyżej, równoważnym sformułowaniem zadania jest:

Znaleźć wszystkie słowa kodowe, synchronizujące każdy ważny wierzchołek.

Synchronizacja oznacza sprowadzenie wierzchołka do ε, bo taki jest stan dekodera D_0 po zakończeniu czytania pełnego słowa kodowego. Rozwiązanie w naturalny sposób dzieli się na dwie części:

1. znalezienie zbioru wierzchołków ważnych,
2. znalezienie słów kodowych synchronizujących wszystkie wierzchołki ważne.

Zadanie wyszukania wierzchołków ważnych da się również rozbić następująco:

1. wyznaczenie wierzchołków wewnętrznych, do których da się dojść z korzenia po sufiksach pojedynczych słów kodowych (nazwijmy je wierzchołkami *bardzo ważnymi*),
2. wyznaczenie wierzchołków wewnętrznych, do których da się dojść z wierzchołków bardzo ważnych po ciągach słów kodowych.

Aby nigdzie nam nie zaginęła w tekście, poniżej ponawiamy dokładną definicję wierzchołków bardzo ważnych:

Definicja 2. Zbiór wierzchołków *bardzo ważnych* dla drzewa kodu prefiksowego to zbiór wierzchołków wewnętrznych drzewa, do których da się dojść z korzenia po pewnym sufiksie pewnego słowa kodowego.

Na rysunku 1 wierzchołki bardzo ważne to ε, `0` oraz `1`. Rzeczywiście, etykiety tych wierzchołków to sufiksy słów kodowych. Są to również jedyne wierzchołki ważne w tym drzewie, bo startując z dowolnego wierzchołka bardzo ważnego i przechodząc po dowolnym słowie kodowym, trafiamy z powrotem do jednego z wierzchołków bardzo ważnych.

Poszczególne etapy rozwiązania są przedstawione w kolejnych podrozdziałach.

Wyznaczenie zbioru wierzchołków bardzo ważnych

Podamy kilka algorytmów dla problemu wyszukiwania wierzchołków bardzo ważnych. Najprostszy z nich jest naturalny, choć nieefektywny. Drugi algorytm stanowi jego optymalizację. Trzeci algorytm jest najefektywniejszy, jednak działa na innej zasadzie oraz nie poprawia całkowitej złożoności rozwiązania.

Naiwny algorytm dla zbioru wierzchołków bardzo ważnych

Naiwny algorytm znajdowania wierzchołków bardzo ważnych bazuje na definicji tych wierzchołków. Wierzchołek q jest bardzo ważny, jeśli $q = \delta(\varepsilon, u)$ dla pewnego sufiksu u słowa kodowego. W naiwnym algorytmie sprawdzamy wszystkie słowa kodowe po kolei. Dla każdego słowa w rozważamy wszystkie jego sufiksy u i dodajemy do zbioru wierzchołków bardzo ważnych wierzchołek docelowy przejścia z korzenia po rozważanym sufiksie, czyli $\delta(\varepsilon, u)$.

Przyjmijmy następujące oznaczenia:

- K — zbiór wszystkich słów kodowych rozważanego kodu,
- suffixes(w) — zbiór wszystkich sufiksów słowa w; zbiór ten zawiera w szczególności słowo puste i w,
- N — liczba słów kodowych w kodzie,
- $w_1, w_2, \ldots, w_N$ — kolejne słowa kodowe kodu.
- $L = |w_1| + |w_2| + \ldots + |w_N|$ — suma długości wszystkich słów kodowych kodu.

Oto pseudokod naiwnego algorytmu dla zbioru wierzchołków bardzo ważnych:

```
1: W := ∅;
2: forall w in K do
3:    forall u in suffixes(w) do
4:       W := W ∪ {δ(ε, u)};
5: return W;
```

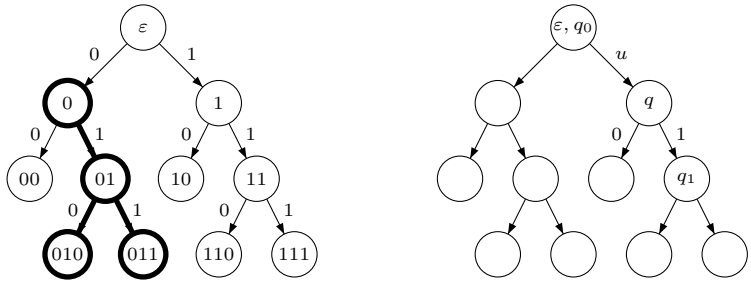
Dla każdego słowa w wykonujemy $|w|+1$ przejść po sufiksie tego słowa. Sufiksy te mają długość $0, 1, 2, \ldots, |w|$, co daje w sumie $O(|w|^2)$ operacji. Złożoność algorytmu naiwnego wynosi więc $O(|w_1|^2 + |w_2|^2 + \ldots + |w_N|^2)$. W pesymistycznym przypadku, jeśli słowa kodowe mają długość rzędu N, czyli dla drzewa niezrównoważonego, złożoność algorytmu to $O(N^3)$. Z kolei gdy drzewo kodu jest zrównoważone, czyli gdy słowa kodowe mają długości $O(\log N)$, złożoność algorytmu wyniesie $O(N \log^2 N)$. Jakkolwiek w optymistycznym przypadku złożoność jest niezła, przypadek pesymistyczny jest wysoce niezadowalający.

Poprawiony algorytm dla zbioru wierzchołków bardzo ważnych

Optymalizacja naiwnego algorytmu będzie opierała się na spostrzeżeniu, że wiele sufiksów słów kodowych ma wspólne fragmenty. Jeśli słowa kodowe różnią się tylko na ostatnim bicie, np. `1010100` oraz `1010101`, to odpowiadające sobie sufiksy tych słów też różnią się tylko na ostatnim bicie. Zamiast więc wykonywać obliczenia dla obu sufiksów oddzielnie, możemy większość obliczeń wykonać wspólnie.

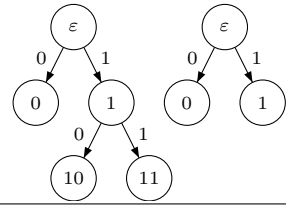
Rozważmy u_0 i u_1 — sufiksy słów kodowych różniące się tylko na ostatnim bicie. Możemy napisać $u_0 = u\mathtt{0}$ oraz $u_1 = u\mathtt{1}$, gdzie u jest wspólną częścią słów. W algorytmie naiwnym obliczaliśmy oddzielnie $q_0 = \delta(\varepsilon, u_0)$ oraz $q_1 = \delta(\varepsilon, u_1)$ i dodawaliśmy oba wierzchołki do zbioru wierzchołków bardzo ważnych. Wartość $\delta(\varepsilon, u_0)$, czyli wierzchołek

docelowy przejścia od korzenia po słowie u_0, możemy znaleźć w dwóch krokach. Najpierw obliczymy przejście po słowie u: $q = \delta(\varepsilon, u)$, a następnie z wierzchołka q przejdziemy po bicie 0: $q_0 = \delta(q, 0)$. Podobnie wyznaczymy $\delta(\varepsilon, u_1)$, jednak w tym przypadku nie ma potrzeby obliczania q ponownie — wystarczy obliczyć $q_1 = \delta(q, 1)$. Przykład opisanych w tym akapicie obliczeń jest przedstawiony na rysunku 4. (Uwaga, rysunek przedstawia inne drzewo niż w treści zadania.)



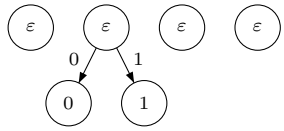
Rys. 4: Przykład wykorzystania wspólnych prefiksów słów do przyspieszenia naiwnego algorytmu znajdowania wierzchołków bardzo ważnych. Rozważamy sufiksy długości 2 kodów 010 i 011, czyli 10 i 11. Sufiksy te są wyznaczone przez etykiety pogrubionych krawędzi na rysunku z lewej strony. Przejścia z korzenia po tych sufiksach ilustruje rysunek z prawej. Wspólną część sufiksów stanowi słowo $u = 1$ i po tym słowie trafiamy do wierzchołka q. Następnie po bitach 0 i 1 trafiamy z q do q_0 i q_1. Zauważmy, że q_0 jest korzeniem, gdyż z wierzchołka 10 przechodzimy bezpośrednio do ε.

Popatrzmy teraz na wszystkie sufiksy słów kodowych powstałe przez obcięcie pierwszego bitu. Sufiksy te są opisane przez drzewa binarne, które są zaczepione w lewym i prawym synu korzenia drzewa kodu. Dla kodu z rysunku 1, poddrzewa odpowiadające tym sufiksom przedstawione są na rysunku 5. Zauważmy, że etykiety w węzłach drzewa zostały poprawione (względem etykiet z rysunku 1), aby odzwierciedlały etykiety na ścieżce od nowego korzenia do węzła.



Rys. 5: Poddrzewa drzewa z rysunku 1 odpowiadające sufiksom słów kodowych powstałych po obcięciu pierwszego bitu.

Tak samo sufiksy powstałe przez obcięcie pierwszych dwóch bitów można opisać za pomocą poddrzew zaczepionych w wierzchołkach w odległości 2 od korzenia (rysunek 6). W ogólności, sufiksy powstałe przez obcięcie pierwszych k bitów można opisać przez poddrzewa zaczepione w wierzchołkach w odległości k od korzenia. Zatem zbiór wszystkich sufiksów słów kodowych jest opisany przez wszystkie poddrzewa zaczepione w wierzchołkach drzewa kodu.



Rys. 6: Poddrzewa drzewa z rysunku 1 odpowiadające sufiksom słów kodowych powstałych po obcięciu pierwszych dwóch bitów.

W nowym algorytmie skorzystamy z tego, że sufiksy tworzą drzewa binarne, aby wykonywać operacje na wielu sufiksach jednocześnie. Dla każdego drzewa sufiksów będziemy obliczać funkcję $\delta(\varepsilon, u')$ dla u' odpowiadających etykietom *wszystkich* wierzchołków tego drzewa, także wierzchołków wewnętrznych (docelowo interesuje nas wartość $\delta(\varepsilon, u)$ dla u będących sufiksami słów kodowych, czyli odpowiadających liściom drzewa sufiksów). Dzięki temu, mając wartość $\delta(\varepsilon, u')$, będziemy mogli w jednym kroku obliczyć wartość $\delta(\varepsilon, x)$ dla x będącego synem wierzchołka u'. W ten sposób każde poddrzewo sufiksów zostanie przetworzone w czasie proporcjonalnym do jego rozmiaru.

Na procedurę przetwarzania drzewa sufiksów możemy spojrzeć jak na synchroniczne przechodzenie po dwóch drzewach. W każdym momencie trzymamy po jednym wierzchołku z każdego drzewa — drzewa kodu (x) i poddrzewa sufiksów (y). Przy przechodzeniu po etykiecie `0`, każdy wierzchołek przechodzi na swojego lewego syna (dla etykiety `1` — na prawego). Wierzchołek x trzyma aktualną wartość funkcji δ, więc jeśli dojdziemy z nim do liścia, musimy przejść do korzenia. Wierzchołek y trzyma aktualny wierzchołek w drzewie sufiksów. Naszym celem jest przetworzenie wszystkich sufiksów z tego drzewa, więc liście tego drzewa ograniczają głębokość przechodzenia.

Pseudokod przetwarzania drzewa sufiksów (pojedynczego!) w celu uzupełnienia zbioru wierzchołków bardzo ważnych B wygląda następująco:

```
1: procedure PRZESZUKIWANIESYNCHRONICZNE1A(x, y)
2: begin
3:   if x jest liściem then
4:     x := ε;
5:   if y jest liściem then
6:     B := B ∪ {x};
7:   else
8:     PRZESZUKIWANIESYNCHRONICZNE1A(x.left, y.left);
9:     PRZESZUKIWANIESYNCHRONICZNE1A(x.right, y.right);
10: end
```

Aby przetworzyć wszystkie sufiksy naszego kodu, musimy wywołać procedurę PRZESZUKIWANIESYNCHRONICZNE1A(ε, y) dla każdego wierzchołka y. Możemy jednak dokonać drobnej optymalizacji. Jeśli w czasie przechodzenia x stanie się liściem, to zostanie ustawione z powrotem na korzeń. Wartości zmiennych x i y są więc wówczas identyczne jak dla bezpośredniego wywołania PRZESZUKIWANIESYNCHRONICZNE1A(ε, y) dla aktualnej wartości wierzchołka y. Takie wywołanie zostanie (lub zostało) wykonane, więc w tym momencie można je pominąć.

Ostateczna postać naszego algorytmu wyznaczającego wierzchołki bardzo ważne to:

```
 1: procedure PRZESZUKIWANIESYNCHRONICZNE1B(x, y)
 2: begin
 3:   if x jest liściem then
 4:     return; { omawiana optymalizacja }
 5:   if y jest liściem then
 6:     B := B ∪ {x};
 7:   else
 8:     PRZESZUKIWANIESYNCHRONICZNE1B(x.left, y.left);
 9:     PRZESZUKIWANIESYNCHRONICZNE1B(x.right, y.right);
10: end
11: { Kod wyznaczający wierzchołki bardzo ważne: }
12: B := ∅;
13: forall q in T do { T — zbiór wszystkich wierzchołków }
14:   PRZESZUKIWANIESYNCHRONICZNE1B(ε, q);
15: return B;
```

Koszt przetworzenia pojedynczego poddrzewa o korzeniu w wierzchołku q jest proporcjonalny do liczby wierzchołków w tym poddrzewie, którą oznaczymy przez S_q. Ten koszt należy przesumować po wszystkich wierzchołkach, aby dostać złożoność całego algorytmu: $O\left(\sum_{q \in T} S_q\right)$.

Obliczenie tej sumy zostawimy na koniec. Zapowiemy teraz jedynie, że złożoność ta wynosi $O(|w_1| + |w_2| + \ldots + |w_N|) = O(L)$, czyli jest znacznie lepsza niż dla algorytmu naiwnego; dodatkowo, warunek z treści zadania gwarantuje, że $L \leqslant 10^8$, czyli że jest to wartość stosunkowo nieduża.

Efektywny algorytm wyznaczania wierzchołków bardzo ważnych

Algorytm poprawiony ma tę zaletę, że zastosowana w nim technika jest identyczna z techniką używaną w pozostałych krokach rozwiązania. Nie jest on jednak ciągle optymalny. Jako ciekawostkę podamy algorytm o pesymistycznej złożoności $O(N)$, używający techniki podobnej do przetwarzania wstępnego w algorytmie Aho–Corasicka ([35]). Ta poprawka nie zmienia jednak pesymistycznej złożoności całego rozwiązania zadania.

Dla każdego wierzchołka q drzewa T wyznaczymy następujące wierzchołki:

- $L[q]$ — najniższy liść odpowiadający pewnemu właściwemu sufiksowi etykiety q,
- $I[q]$ — najniższy wierzchołek wewnętrzny odpowiadający pewnemu właściwemu sufiksowi etykiety q (I pochodzi od angielskiego *internal node*).

Właściwy sufiks słowa to końcowy fragment tego słowa krótszy niż całe słowo, a *najniższy* znaczy będący najdalej od korzenia, czyli mający najdłuższą etykietę. Jeśli obliczenia będziemy wykonywać, poczynając od wierzchołków położonych najwyżej (czyli od wierzchołków o mniejszej odległości od korzenia), wartości L i I dla aktualnego wierzchołka będziemy mogli wyznaczyć w oparciu o wartości L i I ojca.

Faktycznie, niech p będzie ojcem wierzchołka q, a b bitem na krawędzi łączącej te wierzchołki. $I[p]$ daje najniższy wierzchołek wewnętrzny odpowiadający właściwemu sufiksowi etykiety p. Z $I[p]$ można przejść po bicie b do pewnego wierzchołka x. Nietrudno

się przekonać, że etykieta x jest sufiksem etykiety q. W dodatku x jest najniższym wierzchołkiem o tej własności — jeśli istniałby niższy, to jego rodzic odpowiadałby sufiksowi etykiety p, a byłby niżej niż $I[p]$, co prowadzi do sprzeczności. Mamy dwa przypadki:

- Jeśli x jest liściem, to jest najniższym liściem odpowiadającym właściwemu sufiksowi etykiety q. Wtedy sufiks etykiety q odpowiadający najniższemu wierzchołkowi wewnętrznemu musi być też sufiksem etykiety x.
- Jeśli x jest wierzchołkiem wewnętrznym, to jest najniższym wierzchołkiem wewnętrznym odpowiadającym właściwemu sufiksowi etykiety q. Wtedy sufiks etykiety q odpowiadający najniższemu liściowi, jeśli istnieje, musi być też sufiksem etykiety x.

Obliczenia przedstawia poniższy algorytm. Zwróćmy uwagę na szczególny przypadek korzenia drzewa, dla którego każda z wartości $I[\varepsilon]$ oraz $L[\varepsilon]$ jest pusta.

```
1:  L[ε] := nil;
2:  I[ε] := nil;
3:  forall q in T w kolejności BFS[1], oprócz korzenia do
4:      p := q.parent;
5:      b := etykieta krawędzi między p i q;
6:      if I[p] = nil then
7:          x := ε;
8:      else
9:          x := δ(I[p], b);
10:     if x jest liściem then
11:         L[q] := x;
12:         I[q] := I[x];
13:     else
14:         I[q] := x;
15:         L[q] := L[x];
```

Mając obliczone wartości $I[q]$ (wartości $L[q]$ służyły wyłącznie do wyznaczenia $I[q]$), możemy znaleźć wszystkie wierzchołki odpowiadające sufiksom słów kodowych. Są to wszystkie wierzchołki $I[q]$ dla liści, wszystkie wierzchołki $I[I[q]]$ dla liści itd.

```
1:  B := ∅;
2:  forall q in T w odwróconej kolejności BFS do
3:      if q ∈ B or q jest liściem then
4:          B := B ∪ I[q];
5:  return B;
```

Koszt czasowy tego algorytmu jest ewidentnie proporcjonalny do rozmiaru drzewa, a zatem, na mocy Obserwacji 1, wynosi $O(N)$.

[1] tzn. w kolejności rosnących odległości od korzenia

Wyznaczenie zbioru wierzchołków ważnych

Zakładamy w tym momencie, że zbiór wierzchołków bardzo ważnych B mamy dany. Jak zostało napisane wcześniej, wierzchołki ważne to wierzchołki, do których da się dojść z wierzchołków bardzo ważnych po ciągach słów kodowych. Takie wierzchołki łatwo wyznaczyć, inicjalizując zbiór W zbiorem wierzchołków bardzo ważnych i rozszerzając go kolejno o wierzchołki, do których da się dojść z pewnego wierzchołka z W po pojedynczym słowie kodowym. Procedurę kończymy, gdy przejścia z wierzchołków ze zbioru W po słowach kodowych trafiają zawsze do W. Otrzymany zbiór to zbiór wierzchołków ważnych.

W algorytmie będziemy trzymać kolejkę wierzchołków do przetworzenia, zainicjalizowaną zbiorem wierzchołków bardzo ważnych, oraz zbiór wierzchołków, do którego będziemy wrzucać wierzchołki ważne, również zawierający na początku wszystkie wierzchołki bardzo ważne. Przetworzenie wierzchołka będzie polegać na sprawdzeniu przejść z tego wierzchołka po wszystkich słowach kodowych. Każdy wierzchołek docelowy takiego przejścia jest wierzchołkiem ważnym. Jeśli natrafimy pierwszy raz na ten wierzchołek, czyli jeśli nie ma go w naszym zbiorze, to dodajemy go do zbioru oraz do kolejki przetwarzania. Obliczenia kończą się po opróżnieniu kolejki.

```
1:  W := zbiór wierzchołków bardzo ważnych;
2:  Q.insertAll(W);
3:  while Q ≠ ∅ do
4:    q := Q.popFront();
5:    forall w in K do
6:      q' := δ(q, w);
7:      if q' ∉ W then
8:        W := W ∪ {q'};
9:        Q.insert(q');
10: return W;
```

W tej postaci algorytm ma złożoność czasową $O(N \cdot L)$.

Podobnie jak poprzednio, możemy skorzystać z tego, że wiele słów kodowych posiada wspólne prefiksy. Znowu wykorzystamy synchroniczne przechodzenie po dwóch drzewach. W pierwszym drzewie (zmienna x) startujemy z wierzchołka q, który przetwarzamy. Drugie drzewo (zmienna y) opisuje wszystkie słowa kodowe, po których przechodzimy z wierzchołka q. Zaczynamy w nim więc z korzenia.

Jeśli przy takim przechodzeniu wierzchołek x stanie się liściem, to musi on przeskoczyć od korzenia. Jeśli z kolei wierzchołek y dojdzie do liścia, to skończyło się słowo kodowe i do aktualnego wierzchołka x da się dojść z wierzchołka q po całym słowie kodowym. Zatem wierzchołek x dodajemy do zbioru wierzchołków ważnych, jeśli go tam jeszcze nie ma, i dodajemy do przyszłego przetwarzania. W tej postaci złożoność algorytmu wynosi $O(N^2)$, gdyż jest maksymalnie $O(N)$ wierzchołków ważnych (patrz Obserwacja 1) i dla każdego z nich wykonujemy $O(N)$ operacji.

Aby przyspieszyć działanie algorytmu (jak się wkrótce okaże, poprawia to również jego złożoność), zapamiętujemy, jakie pary (ε, y) były już przetwarzane. Nie ma sensu przetwarzać jakiejś pary powtórnie, więc po napotkaniu takiej pary kolejny raz, nie kontynuujemy rekurencyjnych wywołań.

```
1:  procedure PRZESZUKIWANIESYNCHRONICZNE2(x, y)
2:  begin
3:    if x jest liściem then
4:      PRZESZUKIWANIESYNCHRONICZNE2(ε, y);
5:    else
6:      if x = ε then
7:        if y.visited then return;
8:        else y.visited := true;
9:      if y jest liściem then
10:       if x ∉ W then
11:         Q.insert(x);
12:         W := W ∪ {x};
13:     else
14:       PRZESZUKIWANIESYNCHRONICZNE2(x.left, y.left);
15:       PRZESZUKIWANIESYNCHRONICZNE2(x.right, y.right);
16: end
17: { Kod wyznaczający wierzchołki ważne: }
18: W := zbiór wierzchołków bardzo ważnych;
19: Q.insertAll(W);
20: forall x in T do
21:   x.visited := false;
22: while Q ≠ ∅ do
23:   PRZESZUKIWANIESYNCHRONICZNE2(Q.popFront(), ε);
24: return W;
```

Aby oszacować złożoność tego algorytmu, zauważmy, że każda para (x,ε) oraz (ε,y) występuje w przeszukiwaniu co najwyżej raz. Jest to zapewnione odpowiednio przez to, że każdy wierzchołek pojawia się na kolejce co najwyżej raz, oraz przez flagę *visited*. Liczba operacji potrzebna do przetworzenia pojedynczej pary (x,ε) lub (ε,y), z pominięciem operacji wykonanych w ramach rekurencyjnego przetwarzania innych par (x,ε) oraz (ε,y), jest proporcjonalna do rozmiaru poddrzewa zaczepionego w x lub, odpowiednio, w y. Istotnie, wartość pierwszego elementu z pary (x,ε) podczas takiego przetwarzania nie może wyjść poza poddrzewo zaczepione w x. Analogicznie, wartość drugiego elementu pary (ε,y) nie może wyjść poza poddrzewo zaczepione w y.

Złożoność algorytmu wyznaczania wierzchołków ważnych z wierzchołków bardzo ważnych wynosi zatem $O\left(\sum_{q\in T} S_q\right)$.

Znalezienie kodów synchronizujących

Znaleźliśmy już wierzchołki ważne. Słowo w jest synchronizujące, jeśli dla każdego wierzchołka ważnego q zachodzi $\delta(q,w)=\varepsilon$. Najprostszy algorytm znajdowania słów synchronizujących wygląda następująco.

```
1: W := zbiór wierzchołków ważnych;
2: S := ∅;
3: forall w in K do
```

```
4:    b := true;
5:    forall q in W do
6:       if δ(q,w) ≠ ε then
7:          b := false;
8:          break;
9:    if b = true then
10:      S := S ∪ {w};
11: return S;
```

Ten algorytm ma złożoność $O(N \cdot L)$. Metoda jego przyspieszenia jest podobna jak poprzednio. W głównej pętli przeglądamy wszystkie wierzchołki ważne. Dla każdego wierzchołka analizujemy od razu wszystkie słowa kodowe, wykorzystując synchroniczne przechodzenie drzewa, i patrzymy, które z nich *nie* synchronizują danego wierzchołka. Te słowa kodowe nie mogą być synchronizujące.

Podobnie jak w algorytmie znajdowania wierzchołków ważnych, w pierwszym drzewie (zmienna x) startujemy z wierzchołka q, który przetwarzamy. Drugie drzewo (zmienna y) opisuje wszystkie słowa kodowe.

Taki poprawiony algorytm będzie miał złożoność $O(N^2)$ (dla każdego wierzchołka ważnego wykonane będzie co najwyżej N operacji). Dzięki wykorzystaniu optymalizacji obcinającej pary (ε, y), które uprzednio były przetwarzane, złożoność algorytmu można obniżyć do $O\left(\sum_{q \in T} S_q\right)$. Szczegółowa analiza złożoności tego algorytmu jest analogiczna do analizy złożoności algorytmu wyznaczania wierzchołków ważnych, więc ją pomijamy.

```
1:  procedure PRZESZUKIWANIESYNCHRONICZNE3(x, y)
2:  begin
3:     if y jest liściem then
4:        if x nie jest liściem then
5:           y.synchronizing := false;
6:     else
7:        if x jest liściem then
8:           if not y.visited then
9:              y.visited := true;
10:             PRZESZUKIWANIESYNCHRONICZNE3(ε, y);
11:       else
12:          PRZESZUKIWANIESYNCHRONICZNE3(x.left, y.left);
13:          PRZESZUKIWANIESYNCHRONICZNE3(x.right, y.right);
14: end
15: { Wyznaczanie kodów synchronizujących: }
16: W := zbiór wierzchołków ważnych;
17: forall x in liście T do
18:    x.synchronizing := true;
19: forall q in wierzchołki wewnętrzne T do
20:    q.visited := false;
21: forall q in W do
22:    PRZESZUKIWANIESYNCHRONICZNE3(q, ε);
```

Analiza złożoności

We wszystkich trzech krokach algorytmu złożoność wynosiła $O\left(\sum_{q\in T} S_q\right)$. Sugerowaliśmy, że jest ona równa $O(|w_1|+|w_2|+\ldots+|w_N|)$.

Udowodnimy tę własność, pokazując, że

$$\sum_{q\in T} S_q = 1+2\sum_{w\in K}|w|.$$

Dowód przeprowadzimy przez indukcję względem rozmiaru drzewa. Dla drzewa składającego się z samego korzenia równość jest prawdziwa: $1 = 1+2\cdot 0$. Załóżmy, że równość jest prawdziwa dla drzew T_1 i T_2 odpowiadających kodom odpowiednio K_1 oraz K_2. Pokażemy, że jest ona prawdziwa również dla drzewa T odpowiadającemu kodowi K, składającego się z drzew T_1 i T_2 połączonych wspólnym korzeniem.

$$\begin{aligned}
\sum_{q\in T} S_q &= S_\varepsilon + \sum_{q\in T_1} S_q + \sum_{q\in T_2} S_q = && (1)\\
&= (2\cdot|K|-1) + \left(1+2\sum_{w\in K_1}|w|\right) + \left(1+2\sum_{w\in K_2}|w|\right) = && (2)\\
&= 1+2\cdot|K_1|+2\cdot|K_2|+2\sum_{w\in K_1}|w|+2\sum_{w\in K_2}|w| = && (3)\\
&= 1+2\left(|K_1|+\sum_{w\in K_1}|w|\right)+2\left(|K_2|+\sum_{w\in K_2}|w|\right) = && (4)\\
&= 1+2\sum_{w\in K_1}(|w|+1)+2\sum_{w\in K_2}(|w|+1) = && (5)\\
&= 1+2\sum_{w\in K}|w|. && (6)
\end{aligned}$$

Zaznaczmy, że w przejściu (1)–(2) korzystamy z Obserwacji 1.

Każdy z trzech kroków algorytmu ma złożoność $O(|w_1|+|w_2|+\ldots+|w_N|) = O(L)$, więc cały algorytm ma taką złożoność. Pierwszy krok da się zoptymalizować do złożoności $O(N)$, jednak nie wpływa to na złożoność całego rozwiązania.

Implementacje rozwiązania wzorcowego można znaleźć w plikach `kod.cpp`, `kod1.pas` i `kod2.java`.

Testy

Zadanie było sprawdzane na 10 zestawach danych testowych. Zestawy zawierały od jednego do pięciu testów. Testowe kody zostały przygotowane za pomocą losowego generatora drzew sterowanego kilkoma parametrami. Generator decydował, czy dzieci aktualnego wierzchołka mają być liśćmi, czy wierzchołkami wewnętrznymi. Dla tych ostatnich obliczał nowy zestaw parametrów i działał rekurencyjnie.

Parametry generatora, opisane poniżej, wpływały na kształt drzew, czyli ich wielkość, wysokość oraz stopień zrównoważenia.

- I — liczba wierzchołków wewnętrznych drzewa.

- Z — *zrównoważenie* drzewa — liczba od 0 do 100, przy czym 50 oznacza drzewo zrównoważone, 0 drzewo z dużą liczbą wierzchołków w lewym poddrzewie a małą w prawym, a 100 odwrotnie. Z steruje podziałem liczby I pomiędzy synów danego wierzchołka.

- LZ — losowość zrównoważenia — liczba od 0 (mała losowość) do 100 (duża losowość). LZ opisuje losowość podziału I między synów danego wierzchołka.

- ZZ — zmiana zrównoważenia — prawdopodobieństwo (w procentach), że w danym wierzchołku zrównoważenie zmieni się na $100 - Z_p$, gdzie Z_p to zrównoważenie ojca wierzchołka. Dzięki ZZ drzewa niezrównoważone nie są skrzywione w jedną stronę.

- L — limit — jeśli liczba wierzchołków wewnętrznych spada poniżej tego limitu, to jeden z synów danego wierzchołka musi być liściem. Dzięki parametrowi L można było wygenerować drzewo zrównoważone, w którym długości ścieżek od liści do korzenia (czyli długości kodów) mają znaczną wariancję.

Parametry użyte do wygenerowania testów opisuje poniższa tabelka.

Nazwa	**I**	**Z**	**LZ**	**ZZ**	**L**
kod1.in	50	90	10	20	16
kod2.in	100	50	10	90	60
kod3a.in	200	30	30	80	40
kod3b.in	200	50	0	50	4
kod4a.in	1 000	5	6	20	20
kod4b.in	1 000	99	20	90	30
kod4c.in	2 000	10	6	13	4
kod4d.in	2 000	5	5	54	14
kod4e.in	2 020	3	60	20	13
kod5a.in	20 000	70	20	10	3 000
kod5b.in	30 000	10	20	10	200
kod6.in	100 000	45	30	20	300
kod7.in	500 000	65	30	20	600
kod8.in	500 000	50	20	50	300
kod9a.in	600 000	30	10	30	450
kod9b.in	560 000	50	0	0	10
kod10a.in	20 000	50	20	0	7 000
kod10b.in	600 000	35	30	60	800

Kontekst zadania

Zadanie związane jest z rzeczywistym problemem odporności kodów prefiksowych na błędy transmisji. W zadaniu rozważaliśmy błędy powstałe przez pominięcie prefiksu kodowanego ciągu. Podobnie można rozważać błędy zamiany bitów na przeciwne lub wycięcia środkowego fragmentu wiadomości. Kody prefiksowe, ze względu na właściwości synchronizujące, wykazują wysoką odporność na błędy. Po utracie synchronizacji dekoder spontanicznie wraca do poprawnego odkodowywania. Dzieje się to średnio już po niewielkiej liczbie bitów.

Właściwości synchronizujące kodów poprawia istnienie synchronizujących słów kodowych — tych rozważanych w zadaniu, synchronizujących wierzchołki ważne, a także bardziej ogólnych, synchronizujących wszystkie wierzchołki. Nawet jeśli żadne pojedyncze słowo kodowe nie synchronizuje wszystkich wierzchołków, często można znaleźć ciąg słów kodowych posiadający taką własność. Więcej na temat synchronizacji kodów prefiksowych Czytelnik znajdzie w [34].

Tomasz Kulczyński — Treść zadania
Tomasz Kulczyński, Juliusz Sompolski — Opracowanie
Juliusz Sompolski — Program

Dostępna pamięć: 32 MB. *OI, Etap III, dzień drugi, 02.04.2009*

Poszukiwania

Młodzi i szaleńczo w sobie zakochani Bajtek i Bajtyna wpadli w niezłe tarapaty. Zły czarownik Bitocy porwał Bajtynę, mając nadzieję na sowity okup. Bajtek nie poddaje się jednak — nie jest zbyt majętny, więc postanowił odbić swoją wybrankę. Bitocy nie lubi walki wręcz, zaproponował więc inne rozwiązanie całej sytuacji. Jeśli młodzieniec zgadnie, na którym piętrze wieży znajduje się uwięziona, czarodziej puści ją wolno.

Pięter jest bardzo wiele — są ponumerowane od 1 do n. Jedyną pomocą dla Bajtka mogą być pytania zadawane czarodziejowi. Pytania muszą być postaci „Czy Bajtyna jest wyżej/niżej niż piętro x?". Oczywiście, Bajtek może wybrać dokładnie jeden z wyrazów „wyżej", „niżej", a także dowolnie ustalić liczbę x. Bitocy zawsze odpowiada na tak postawione pytanie zgodnie z prawdą, ale każe sobie zapłacić a bajtalarów, jeśli odpowiedź brzmi „tak", lub b bajtalarów, gdy odpowiedź brzmi „nie". Cóż, jeśli Bajtek zbytnio zubożeje, a Bitocy nadmiernie się wzbogaci, to Bajtyna może zechcieć zostać u czarodzieja...

Bajtek zastanawia się, jakie pytania zadawać. Niestety Bajtyna słyszy całą rozmowę, tzn. kolejne pytania Bajtka i odpowiedzi Bitocego, a jest osobą bardzo oszczędną. Jeśli tylko Bajtek wyda choćby o jednego bajtalara więcej, niż jest to (w najgorszym przypadku) niezbędne do ustalenia jej położenia, to obrazi się na niego śmiertelnie i odejdzie z Bitocym. Dokładniej, jeżeli w pewnym momencie rozmowy da się wywnioskować, że od tego momentu, niezależnie od dalszych odpowiedzi Bitocego, Bajtek może odgadnąć położenie Bajtyny, wydając przy tym nie więcej niż K bajtalarów, a od tego momentu Bajtek wyda kwotę większą niż K, to jego szanse u Bajtyny spadną do zera (punktów za dany test). Pomóż Bajtkowi!

Komunikacja

Powinieneś zaimplementować program, który rozwiąże problem Bajtka, korzystając z dostarczonej biblioteki (symulującej czarodzieja Bitocego). Aby użyć biblioteki, należy wpisać w swoim programie:

- *C/C++:* `#include "poslib.h"`
- *Pascal:* `uses poslib;`

Biblioteka udostępnia trzy procedury i funkcje:

- `inicjuj` — *podaje liczbę pięter n oraz koszty a i b. Powinna zostać użyta dokładnie raz, na samym początku działania programu.*
  - *C/C++:* `void inicjuj(int *n, int *a, int *b);`
  - *Pascal:* `procedure inicjuj(var n, a, b: longint);`
- `pytaj` — *znak c oznacza rodzaj pytania (`'W'` dla wyżej lub `'N'` dla niżej), a x to numer piętra. Wynikiem funkcji jest wartość logiczna odpowiedzi czarodzieja. Twój program może użyć tej funkcji dowolną liczbę razy.*

  - *C/C++:* `int pytaj(char c, int x);` *(0 oznacza fałsz, a 1 prawdę),*
  - *Pascal:* `function pytaj(c: char; x: longint): boolean;`

- `odpowiedz` — *za pomocą tej procedury/funkcji podajesz numer piętra, na którym jest Bajtyna. Powinna zostać użyta dokładnie raz. Jej wykonanie zakończy działanie Twojego programu.*
  - *C/C++:* `void odpowiedz(int wynik);`
  - *Pascal:* `procedure odpowiedz(wynik: longint);`

Twój program nie może otwierać żadnych plików ani używać standardowego wejścia i wyjścia. Rozwiązanie będzie kompilowane wraz z biblioteką następującymi poleceniami:

- *C:* `gcc -O2 -static poslib.c pos.c -lm`
- *C++:* `g++ -O2 -static poslib.c pos.cpp -lm`
- *Pascal:* `ppc386 -O2 -XS -Xt pos.pas`

W katalogu `/home/zawodnik/rozw/lib` *możesz znaleźć przykładowe pliki bibliotek i nieoptymalne rozwiązania ilustrujące sposób ich użycia. Aby podane powyżej polecenia kompilacji działały, pliki bibliotek powinny znajdować się w bieżącym katalogu.*

Limity

Możesz założyć, że $1 \leqslant n \leqslant 10^9$ *oraz* $1 \leqslant a,b \leqslant 10\,000$.

Przykładowy przebieg programu

Wywołanie funkcji	**Zwracane wartości i wyjaśnienia**
Pascal: `inicjuj(n,a,b);` C lub C++: `inicjuj(&n,&a,&b);`	Od tego momentu $n = 5$, $a = 1$, $b = 2$.
`pytaj('W',3);`	Wynik równa się 0/`false`. Pytasz, czy Bajtyna jest powyżej trzeciego piętra. Uzyskujesz odpowiedź, że nie. Płacisz 2 bajtalary.
`pytaj('N',2);`	Wynik równa się 0/`false`. Pytasz, czy jest poniżej drugiego piętra. Uzyskujesz odpowiedź „nie”, za którą płacisz kolejne 2 bajtalary.
`pytaj('W',2);`	Wynik równa się 1/`true`. Pytasz, czy jest powyżej drugiego piętra. Uzyskujesz odpowiedź „tak”, za którą płacisz bajtalara.
`odpowiedz(3);`	Odpowiadasz, że Bajtyna znajduje się na trzecim piętrze. Jest to poprawna odpowiedź. Wydałeś łącznie 5 bajtalarów.

Powyższy przebieg interakcji jest poprawny, ale nieoptymalny, a więc program nie uzyskałby punktów za taki test. W szczególności, dobrze napisany program potrafi dla danych $n = 5$, $a = 1$, $b = 2$ *tak zadawać pytania, żeby w każdym przypadku wydać co najwyżej 4 bajtalary.*

Rozwiązanie

Pierwsze podejście

Pierwsze rozwiązanie podchodzi do problemu wprost, obliczając dla każdego (całkowitego dodatniego) k, jakim minimalnym kosztem można znaleźć Bajtynę w przedziale zawierającym k pięter, w przypadku pesymistycznym. Zauważmy, że wspomniany minimalny koszt nie zależy od umiejscowienia tych k pięter w wieży, a jedynie od ich liczby, tzn. k. Możemy zatem oznaczyć ten koszt przez $t[k]$.

Wartości $t[k]$ będziemy obliczać w kolejności rosnących k, używając programowania dynamicznego. Oczywiście $t[1] = 0$. Jeżeli $k > 1$, to do zidentyfikowania położenia Bajtyny konieczne jest zadanie co najmniej jednego pytania; załóżmy, że jest to pytanie, czy Bajtyna znajduje się niżej niż $(p+1)$-sze od dołu piętro ($1 \leqslant p < k$). Jeżeli Bitocy odpowie, że tak, to za to pytanie Bajtek zapłaci a bajtalarów, a w dalszych pytaniach będzie mógł ograniczyć się do fragmentu wieży zawierającego tylko p pięter. W przeciwnym przypadku Bajtek poniesie koszt b bajtalarów, a przedział pięter zawęzi się do długości $k - p$. Zauważmy, że w obu przypadkach udało nam się w prosty sposób wyrazić *cały* zasób informacji, jakie Bajtek może wywnioskować z jednej odpowiedzi Bitocego.

Zakładając, że w pozostałej części konwersacji Bajtek będzie zadawał pytania w sposób minimalizujący dalszy koszt, otrzymujemy, że w przypadku odpowiedzi „tak" (na pierwsze pytanie) Bajtek wyda $a + t[p]$ bajtalarów, a w przypadku odpowiedzi „nie" — $b + t[k - p]$ bajtalarów. W takim razie pesymistyczny koszt w przypadku tego pytania to większa z tych dwóch liczb.

Zauważmy teraz, że pytania:

- „Czy Bajtyna jest wyżej niż piętro p?"
- „Czy Bajtyna jest niżej niż piętro $k - p + 1$?"

nie różnią się wcale pod względem długości przedziałów pięter uzyskanych po odpowiedzi (tj. p oraz $k - p$) oraz odpowiadających im kosztów pierwszego pytania (odpowiednio a oraz b). To pokazuje, że pytań typu „wyżej" nie musimy wcale rozważać. Aby zatem obliczyć $t[k]$, należy znaleźć takie p, by koszt zadania pytania typu „niżej" dla piętra numer $p + 1$ był minimalny. Mamy więc równość:

$$t[k] = \min_{1 \leqslant p < k} \max(\, a + t[p],\, b + t[k - p]\,).$$

Ponadto przez $p[k]$ oznaczmy wartość p realizującą to minimum, czyli spełniającą:

$$t[k] = \max(\, a + t[p[k]],\, b + t[k - p[k]]\,).$$

W ten sposób w złożoności czasowej $O(n^2)$ i pamięciowej $O(n)$ możemy wyznaczyć wszystkie wartości $t[k]$ oraz $p[k]$ dla $k = 1, 2, \ldots, n$. Po tych wstępnych obliczeniach schemat zadawania pytań jest już bardzo prosty do skonstruowania: jeżeli w danym momencie rozwiązania ograniczyliśmy rozważania do k-piętrowego fragmentu wieży ($k > 1$), to kolejnym pytaniem powinno być „Czy Bajtyna jest niżej niż piętro $p[k] + 1$?". Wówczas faza zadawania pytań będzie miała już tylko złożoność $O(n)$.

Takie rozwiązanie zostało zaimplementowane w plikach `poss1.cpp` oraz `poss1p.pas`; na zawodach przechodziło ono pierwsze trzy testy.

Poprawki

Okazuje się, że powyższe rozwiązanie można jeszcze trochę usprawnić. W tym celu należy zauważyć, że funkcja

$$f(p) = \max(\, a+t[p], b+t[k-p]\,)$$

ze wzoru

$$t[k] = \min_{1 \leqslant p < k} f(p)$$

jest najpierw nierosnąca, a potem niemalejąca. Jest tak, ponieważ $f(p)$ równa się maksimum funkcji nierosnącej $b+t[k-p]$ i funkcji niemalejącej $a+t[p]$.

Możemy więc próbować szukać jej minimum szybciej niż przez sekwencyjne sprawdzanie wartości p, a mianowicie za pomocą tzw. *wyszukiwania ternarnego*. Załóżmy, że przedział poszukiwań minimum funkcji $f(p)$ zawęziliśmy już do $p \in [l,r]$. Podzielmy przedział $[l,r]$ na trzy mniej więcej równe części w punktach $p_1 = \lfloor \frac{2l+r}{3} \rfloor$ i $p_2 = \lfloor \frac{l+2r}{3} \rfloor$. Zachodzi jeden z trzech przypadków:

- $f(p_1) < f(p_2)$ — wiemy wówczas, że p_2 jest położone w przedziale, w którym f jest niemalejąca, więc p_2 znajduje się za minimum funkcji. Możemy zatem zawęzić przedział poszukiwań do $p \in [l, p_2 - 1]$.
- $f(p_1) > f(p_2)$ — analogicznie, wówczas p_1 znajduje się przed minimum f, a więc możemy zawęzić przedział poszukiwań do $p \in [p_1 + 1, r]$.
- $f(p_1) = f(p_2) \stackrel{\text{def}}{=} m$ — w tej sytuacji nie jesteśmy w stanie łatwo stwierdzić, czy p_1 i p_2 są po lewej, po prawej, czy po przeciwnych stronach minimum. Możemy jednak próbować szukać minimum rekurencyjnie w przedziałach $[l, p_1]$, $[p_1+1, p_2-1]$ i $[p_2, r]$. Jeżeli w którymkolwiek z tych przedziałów znajdziemy minimum mniejsze niż m, to jest to na pewno minimum całej funkcji i nie musimy sprawdzać pozostałych przedziałów. Pesymistycznie jednak może zajść konieczność sprawdzenia wszystkich przedziałów.

Gdyby nie trzeci przypadek, rozwiązanie to miałoby złożoność czasową $O(n \log n)$, gdyż minimum f znajdowalibyśmy w czasie $O(\log n)$, za każdym podziałem redukując długość przedziału poszukiwań minimum o czynnik $\frac{2}{3}$. Ostatni przypadek sprawia jednak, że rozwiązanie to może działać w złożoności $O(n^2)$, choć w praktyce działa dużo szybciej niż pierwsze.

Implementacje tego rozwiązania można znaleźć w plikach `poss2.cpp` i `poss2p.pas`; przechodziły one na zawodach 5 pierwszych grup testów. W rozwiązaniach tych zastosowano pewne drobne, odkryte eksperymentalnie usprawnienia, na przykład to, że dla małych wartości parametru k szybszy w praktyce okazuje się wcześniejszy algorytm z wyszukiwaniem liniowym.

Rozwiązanie wzorcowe

Odwróćmy stawiany w zadaniu problem. Zamiast wyznaczać koszt znalezienia odpowiedzi dla danego przedziału pięter, dla każdego kosztu c obliczmy, ile co najwyżej pięter może

występować w zakresie poszukiwań, żeby koszt optymalnego wyszukania wśród tych pięter wyniósł nie więcej niż c. Oznaczmy tę szerokość przedziału (liczbę pięter) przez $Q[c]$.

Na początek zauważmy, że $Q[0] = 1$. Przyjmijmy dla wygody, że dla $c < 0$ mamy $Q[c] = -\infty$. Niech teraz $c > 0$. Załóżmy, że poszukiwania Bajtyny ograniczyliśmy już do przedziału złożonego z $Q[c]$ pięter i chcemy zadać czarownikowi następne pytanie. Po odpowiedzi przedział podzieli się na dwa krótsze, w których musimy umieć wyznaczyć rozwiązanie odpowiednio za $c-a$ i $c-b$ bajtalarów (nie wiemy, na który z nich wskaże Bitocy). Przedziały te muszą więc mieć długości nie większe niż $Q[c-a]$ i $Q[c-b]$. Stąd już prosto wnioskujemy, że:

$$Q[c] = \max(Q[c-a] + Q[c-b], 1)$$

i że do wyznaczania wartości $Q[c]$ możemy, podobnie jak w poprzednim rozwiązaniu, zastosować programowanie dynamiczne.

Aby teraz zadać odpowiednie pytanie, wystarczy znaleźć najmniejszy taki koszt w, że $Q[w-1] < n \leqslant Q[w]$. Jest to koszt, który pesymistycznie będziemy musieli ponieść. Przyjmijmy $n_0 = n$, $w_0 = w$. Kolejne nasze pytania będą dzielić przedział długości

$$n_i \leqslant Q[w_i] = Q[w_i - a] + Q[w_i - b]$$

na przedziały długości $Q[w_i - a]$ (do którego powinno zawężać się poszukiwanie po otrzymaniu odpowiedzi „tak") oraz $n_i - Q[w_i - a] \leqslant Q[w_i - b]$ (do którego powinno zawężać się poszukiwanie po otrzymaniu odpowiedzi „nie"). Wówczas po odpowiedzi Bitocego albo zapłacimy a bajtalarów i będziemy wyszukiwali w przedziale długości $n_{i+1} = Q[w_i - a]$, albo zapłacimy b bajtalarów i będziemy wyszukiwali w przedziale długości $n_{i+1} \leqslant Q[w_i - b]$. Możemy więc utrzymać się w pesymistycznym koszcie.

Zauważmy jednak, że w przypadku odpowiedzi „nie", niekoniecznie $w' = w_i - b$ jest najmniejszym takim w', że $Q[w'] \geqslant n_{i+1}$. Być może można pesymistyczny koszt w_{i+1} poprawić, próbując kolejnych, mniejszych wartości:

$$w_{i+1} = \min\left\{ j : \;\; j \leqslant w' \;\wedge\; Q[j] \geqslant n_{i+1} \right\}. \tag{1}$$

Konieczność wykonania tego kroku wynika z fragmentu treści zadania, w którym jest zdefiniowane to, jakie dokładnie wymagania odnośnie do strategii zadawania pytań stawia Bajtyna.

Rozwiązanie to ma złożoność $O(w)$ — musimy obliczyć $Q[i]$ dla wszystkich $i \leqslant w$ i przejść po nich w kierunku rosnących wartości i w trakcie obliczania w, a później — w kierunku malejących wartości podczas zadawania pytań.

```
1: inicjuj(n,a,b);
2: Q[0] := 1;
3: w := 0;
4: while Q[w] < n do begin
5:    w := w + 1;
6:    if w < a or w < b then
7:       Q[w] := 1;
8:    else
9:       Q[w] := Q[w - a] + Q[w - b];
```

```
10: end
11: dol := 1;
12: gora := n;
13: while dol < gora do begin
14:   while Q[w-1] >= gora-dol+1 do
15:     w := w-1;
16:   if pytaj('N', dol+Q[w-a]) then begin
17:     gora := dol+Q[w-a]-1;
18:     w := w-a;
19:   end
20:   else begin
21:     dol := dol+Q[w-a];
22:     w := w-b;
23:   end
24: end
25: odpowiedz(dol);
```

Aby oszacować złożoność powyższego rozwiązania względem n, należy ograniczyć z góry wartość w. Zauważmy, że ciąg $Q[i]$ jest niemalejący. Stąd dla $i \geqslant \max(a,b)$ mamy

$$Q[i] = Q[i-a] + Q[i-b] \geqslant 2Q[i-\max(a,b)].$$

Używając wielokrotnie tej nierówności, otrzymujemy, że $Q[\lceil \log_2 n \rceil \cdot \max(a,b)] \geqslant n$, a zatem

$$w \leqslant \lceil \log_2 n \rceil \cdot \max(a,b).$$

Rozwiązanie to działa więc w czasie $O(\log n \cdot \max(a,b))$ i takiej samej pamięci — gdyż musi przechowywać obliczoną tablicę Q. Jego implementacje można znaleźć w plikach `pos.cpp` oraz `pos1.pas`.

Na koniec warto wspomnieć, że w zamierzchłej, VII edycji Olimpiady Informatycznej pojawiło się zadanie interaktywne *Jajka*, którego rozwiązanie przypominało nieco rozwiązanie niniejszego zadania. Po opis tego zadania odsyłamy do książeczki owej olimpiady [7], a także na stronę `http://was.zaa.mimuw.edu.pl`.

Strategie

Do sprawdzania rozwiązań użyto biblioteki pobierającej wartości n,a,b i wykorzystującej następujące strategie:

1. Strategia `sprawiedliwa` — pobiera z wejścia numer piętra, na którym znajduje się Bajtyna, i cały czas odpowiada zgodnie z tym piętrem. W trakcie rozgrywki oszacowanie na pesymistyczny koszt może się poprawiać bądź nie, w zależności od odpowiedzi.

2. Strategia `pesymistyczna` — odpowiada tak, aby Bitocy dostał jak najwięcej pieniędzy — czyli aby pesymistyczne oszacowanie nie poprawiało się. Wbrew pozorom nie jest to najzłośliwsza strategia, gdyż przepuszcza programy, które po każdym ruchu nie sprawdzają, czy da się poprawić oszacowanie na pesymistyczny koszt zgodnie ze wzorem (1).

3. Strategia `optymistyczna` — odpowiada tak, aby po każdym pytaniu pesymistyczne oszacowanie na koszt jak najbardziej się poprawiało. Wymaga od programów zawodników, aby również poprawiały oszacowanie pesymistycznego kosztu.

Warto zaznaczyć, że dwie ostatnie z tych strategii polegają na swego rodzaju oszustwie — zamiast umieścić Bajtynę na wybranym piętrze i prawdomównie odpowiadać na pytania Bajtka, reprezentowany przez bibliotekę Bitocy decyduje o jej umiejscowieniu dopiero w trakcie rozgrywki z Bajtkiem. Zauważmy, że dopóki udzielane po drodze odpowiedzi nie są sprzeczne, czyli cały czas istnieje numer piętra zgodny z nimi wszystkimi, dopóty całe to oszustwo jest niewykrywalne dla Bajtka (a więc także dla programu reprezentującego go).

Biblioteka oblicza i aktualizuje oszacowanie na koszt tak, jak czyni to rozwiązanie wzorcowe. Po każdym pytaniu analizuje, czy program zawodnika nadal może utrzymać się w pesymistycznym oszacowaniu kosztu, a jeśli nie, od razu kończy się z wynikiem błędnym. Nie dopuszcza także strzelania odpowiedzi — analizuje, do jakiego przedziału pięter zawodnik zawęził poszukiwania, i jeśli rozwiązanie zwróci odpowiedź, zanim przedział ten zawęzi się do jednego piętra, również kończy się z wynikiem błędnym (tutaj ponownie mamy przykład niewykrywalnego oszustwa w wykonaniu biblioteki).

Testy

Testy zostały przygotowane zarówno w postaci pojedynczych przypadków testowych, jak i grup testów o tych samych n, a, b i różnych strategiach.

Nazwa	n	Opis
pos1a-1c.in	20	grupa bardzo małych testów o wszystkich możliwych strategiach; jest ona przeznaczona do przejścia przez najbardziej nieoptymalne rozwiązania, o ile rzeczywiście są poprawne (działają dla wszystkich strategii)
pos1d.in	1	przypadek skrajny
pos2.in	$\approx 8\,000$	test ze strategią `pesymistyczną` do przejścia przez rozwiązania wolniejsze oraz niektóre rozwiązania błędne
pos3a-3c.in	$\approx 8\,000$	grupa testów do przejścia przez rozwiązania wolniejsze
pos4.in	$\approx 50\,000$	test ze strategią `pesymistyczną` do przejścia przez rozwiązanie wolniejsze z optymalizacjami oraz przez niektóre rozwiązania błędne
pos5a-5c.in	$\approx 50\,000$	grupa testów do przejścia przez rozwiązanie wolniejsze z optymalizacjami
pos6.in	$\approx 1\,000\,000$	test ze strategią `pesymistyczną` do przejścia przez ewentualnie znalezione przez zawodnika szybsze rozwiązanie wolniejsze
pos7a-7c.in	$\approx 1\,000\,000$	grupa testów do przejścia przez ewentualne znalezione przez zawodnika szybsze rozwiązanie wolniejsze

Nazwa	n	Opis
pos8-9.in	$\approx 10^9$	testy ze strategią `pesymistyczną`
pos10-14a-b.in	$\approx 10^9$	testy do przejścia tylko przez rozwiązanie wzorcowe

XXI Międzynarodowa Olimpiada Informatyczna,

Plovdiv, Bułgaria 2009

Marcin Kubica, Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

IOI 2009, dzień pierwszy, 10.08.2009

POI

Lokalna Plovdivska Olimpiada Informatyczna (POI) odbyła się na dość nietypowych zasadach. Każdy z N startujących zawodników miał do rozwiązania T różnych zadań. Każde zadanie było oceniane przy użyciu jednego testu, a więc dla każdego zadania i każdego zawodnika były tylko dwie możliwości: albo ten zawodnik rozwiązał to zadanie, albo go nie rozwiązał. Nie przyznawano żadnych punktów częściowych.

Liczby punktów przyznawane za poszczególne zadania zostały ustalone po zawodach; były one równe liczbie zawodników, którzy danego zadania nie rozwiązali. Wynikiem zawodnika nazywamy sumę punktów za zadania, które ten zawodnik rozwiązał.

Filipek wystartował w zawodach, ale czuje się zdezorientowany tak skomplikowanymi zasadami oceniania i teraz wpatruje się w wyniki, nie mogąc ustalić swojej pozycji w ostatecznym rankingu. Pomóż Filipkowi, pisząc program, który obliczy jego wynik oraz pozycję w rankingu.

Przed zawodami zawodnikom nadano numery od 1 do N. Filipek dostał numer P. W ostatecznym rankingu Olimpiady zawodnicy są wymienieni w kolejności malejących wyników. W przypadku remisu, wśród zawodników o tym samym wyniku wyżej sklasyfikowani są ci o większej liczbie rozwiązanych zadań. W przypadku remisu przy uwzględnieniu także tego kryterium, zawodnicy z jednakowymi wynikami są sklasyfikowani w kolejności rosnących numerów.

Zadanie

Napisz program, który mając dane o tym, którzy zawodnicy rozwiązali które zadania, wyznaczy wynik Filipka oraz jego pozycję w ostatecznym rankingu.

Ograniczenia

$1 \leqslant N \leqslant 2\,000$ — *liczba zawodników*
$1 \leqslant T \leqslant 2\,000$ — *liczba zadań*
$1 \leqslant P \leqslant N$ — *numer Filipka*

Wejście

Twój program powinien wczytać ze standardowego wejścia następujące dane:

- *W pierwszym wierszu znajdują się liczby całkowite N, T oraz P, pooddzielane pojedynczymi odstępami.*
- *Kolejne N wierszy opisuje, które zadania zostały rozwiązane przez których zawodników. k-ty z tych wierszy zawiera dane o zadaniach rozwiązanych przez zawodnika o numerze k w postaci T liczb całkowitych pooddzielanych odstępami. Pierwsza z nich informuje, czy zawodnik o numerze k rozwiązał pierwsze zadanie. Druga informuje o tym samym*

dla drugiego zadania itd. Każda z tych T liczb jest równa 0 lub 1, przy czym 1 oznacza, że zawodnik o numerze k rozwiązał dane zadanie, a 0 oznacza, że go nie rozwiązał.

Wyjście

Twój program powinien wypisać jeden wiersz z dwiema liczbami całkowitymi oddzielonymi pojedynczym odstępem. Pierwsza z nich to wynik Filipka na POI. Druga natomiast to pozycja Filipka w ostatecznym rankingu. Pozycja jest liczbą całkowitą z zakresu od 1 do N, gdzie 1 oznacza zawodnika sklasyfikowanego na górze rankingu (tzn. zawodnika z najwyższym wynikiem), a N — sklasyfikowanego najniżej (tzn. zawodnika z najniższym wynikiem).

Ocenianie

W testach wartych łącznie 35 punktów żaden z pozostałych zawodników nie będzie miał takiego samego wyniku jak Filipek.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*

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

Pierwsze zadanie pozostało nierozwiązane przez jednego z zawodników, więc jest warte 1 punkt. Drugie pozostało nierozwiązane przez dwóch, więc jest warte 2 punkty. Trzecie zadanie pozostało nierozwiązane przez czterech, więc jest warte 4 punkty. Tak więc pierwszy z zawodników ma wynik równy 4; drugi z nich (Filipek), a także czwarty i piąty, mają wyniki równe 3; trzeci zawodnik ma wynik równy 1. Zawodnicy o numerach 2, 4 i 5 remisują także zgodnie z pierwszą regułą rozstrzygania remisów (według liczby rozwiązanych zadań), a zgodnie z drugą regułą (dotyczącą numerów zawodników) Filipek plasuje się wyżej niż pozostali. Tak więc ostateczna pozycja Filipka w rankingu jest równa 2. Przegrywa tylko z zawodnikiem o numerze 1.

Marcin Kubica, Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

IOI 2009, dzień pierwszy, 10.08.2009

Łucznictwo

Pewien turniej strzelecki jest przeprowadzany na następujących zasadach. W jednej linii ustawia się N celów i numeruje się je od 1 do N w kolejności występowania w linii (skrajnie lewy cel ma numer 1, a skrajnie prawy — N). W turnieju bierze udział $2 \cdot N$ łuczników. W każdym momencie turnieju przy każdym celu stoi dwóch łuczników. Każda runda turnieju jest przeprowadzana zgodnie z następującą procedurą:

- *Pary łuczników stojące przy każdym z celów współzawodniczą ze sobą i wyłaniają spośród siebie zwycięzców i przegranych. Następnie wszyscy łucznicy przemieszczają się w następujący sposób:*
  - *Zwycięzcy przy celach od 2 do N przechodzą o jeden cel w lewo (tzn. odpowiednio do celów od 1 do $N-1$).*
  - *Przegrani przy celach od 2 do N, a także zwycięzca przy celu 1, pozostają przy swoich celach.*
  - *Przegrany przy celu 1 przechodzi do celu N.*

Turniej składa się z R takich rund, przy czym liczba rund jest co najmniej taka jak liczba łuczników (tzn. $R \geqslant 2 \cdot N$).

Jesteś jedynym łucznikiem, który przybył na turniej dokładnie w samą porę. Wszyscy pozostali $2 \cdot N - 1$ łucznicy przybyli przed czasem i ustawili się już w linii. To co musisz zrobić, to ustawić się na jakiejś pozycji pośród nich. Wiesz, że po tym, jak zajmiesz swoją pozycję, dwóch skrajnie lewych łuczników w linii zacznie turniej przy celu numer 1, dwóch następnych przy celu numer 2 itd., a dwóch skrajnie prawych łuczników zacznie przy celu numer N.

Każdy z $2 \cdot N$ łuczników w turnieju (włączając ciebie) ma przydzielony numer rankingowy zgodnie ze swoimi umiejętnościami, przy czym mniejszy numer rankingowy odpowiada wyższym umiejętnościom. Żadni dwaj łucznicy nie mają tego samego numeru rankingowego. Ponadto, jeżeli dwaj łucznicy współzawodniczą, to zawsze wygra ten o mniejszym numerze rankingowym.

Znając umiejętności każdego z zawodników, chcesz ustawić się wśród nich w taki sposób, aby zakończyć turniej przy celu o najmniejszym możliwym numerze. Jeżeli istnieje więcej niż jeden sposób uczynienia tego, chcesz wybrać ten, który zaczyna się przy celu o największym możliwym numerze.

Zadanie

Napisz program, który mając dane numery rankingowe wszystkich łuczników, w tym twój, a także ustawienie twoich przeciwników w linii, wyznaczy, przy którym celu powinieneś zacząć turniej, tak aby osiągnąć wymienione wyżej wymagania.

Ograniczenia

$1 \leqslant N \leqslant 200\,000$ — *liczba celów, a także połowa liczby łuczników*
$2 \cdot N \leqslant R \leqslant 1\,000\,000\,000$ — *liczba rund w turnieju*
$1 \leqslant S_k \leqslant 2 \cdot N$ — *numer rankingowy łucznika* k

Wejście

Twój program powinien wczytać ze standardowego wejścia następujące dane:

- *Pierwszy wiersz zawiera liczby całkowite* N *oraz* R*, oddzielone pojedynczym odstępem.*
- *Kolejne* $2 \cdot N$ *wierszy zawiera numery rankingowe łuczników. Pierwszy z tych wierszy zawiera twój numer rankingowy. Pozostałe zawierają numery rankingowe pozostałych łuczników, po jednym łuczniku w wierszu, w kolejności, w której się oni ustawili (od lewej do prawej). Każdy z tych* $2 \cdot N$ *wierszy zawiera jedną liczbę całkowitą z zakresu od 1 do* $2 \cdot N$*. Najlepszy jest numer rankingowy 1, a najgorszy numer rankingowy* $2 \cdot N$*. Żadni dwaj łucznicy nie mają tego samego numeru rankingowego.*

Wyjście

Twój program powinien wypisać na standardowe wyjście jeden wiersz zawierający jedną liczbę całkowitą między 1 a N*: numer celu, przy którym powinieneś zacząć turniej.*

Ocenianie

W testach wartych łącznie 60 punktów N *nie przekroczy 5 000.*
Ponadto, w pewnych spośród tych testów, wartych łącznie 20 punktów, N *nie przekroczy 200.*

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*

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

Jesteś przedostatnim łucznikiem w rankingu. Jeżeli rozpoczniesz przy celu 1, to przejdziesz następnie do celu 4 i pozostaniesz tam do końca. Jeżeli zaczniesz przy celu 2 lub 4, pozostaniesz tam przez cały turniej. Jeżeli zaczniesz przy celu 3, pokonasz najsłabszego łucznika, przesuniesz się do celu 2 i tam już pozostaniesz.

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
2
```

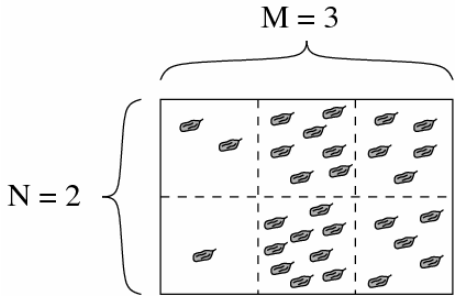
Jesteś drugim łucznikiem w rankingu. Najlepszy z łuczników znajduje się już przy celu 1 i pozostanie tam przez cały turniej. W ten sposób, niezależnie od twojej pozycji początkowej, będziesz się w kółko przemieszczał, począwszy od celu początkowego, po kolejnych celach od 4 do 1. Żeby po 9 przemieszczeniach skończyć przy celu 1, powinieneś zacząć przy celu numer 2.

Marcin Kubica, Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

IOI 2009, dzień pierwszy, 10.08.2009

Rodzynki

Słynna producent czekolady z Plovdiv, Bonny, musi pociąć tabliczkę czekolady z rodzynkami. Czekolada jest prostokątem składającym się z jednakowych, kwadratowych kawałków. Kawałki są ułożone równolegle do krawędzi czekolady w N rzędach i M kolumnach, co daje łącznie $N \cdot M$ kawałków. Każdy z kawałków zawiera pewną dodatnią liczbę rodzynek, a żadne rodzynki nie leżą na liniach oddzielających kawałki.



Początkowo czekolada jest w całości. Bonny chce ją ciąć na coraz mniejsze fragmenty, aż w końcu otrzyma czekoladę podzieloną na $N \cdot M$ pojedynczych kawałków. Ponieważ Bonny jest bardzo zajęta, do pomocy przy cięciu potrzebuje pomocy swego asystenta, Chytrego Piotra. Piotr tnie czekoladę po prostych, od brzegu do brzegu, i oczekuje zapłaty za każde wykonane cięcie. Bonny chwilowo nie posiada pieniędzy, ma za to sporo rodzynek, które nie zostały użyte, więc oferuje Piotrowi zapłatę w rodzynkach. Chytry Piotr zgadza się, ale pod warunkiem, że za każde przecięcie fragmentu czekolady na dwa mniejsze dostanie tyle rodzynek, ile jest ich łącznie w tym fragmencie.

Bonny chciałaby dać Piotrowi jak najmniej rodzynek. Wie dokładnie, ile rodzynek jest w każdym z $N \cdot M$ kawałków. Może też zadecydować o kolejności, w której będzie podawać Piotrowi fragmenty czekolady, a także kazać mu wykonywać cięcia w konkretnym kierunku (w pionie lub poziomie) i wzdłuż konkretnej linii. Pomóż Bonny znaleźć taki sposób cięcia czekolady na pojedyncze kawałki, żeby zapłaciła Chytremu Piotrowi jak najmniej rodzynek.

Zadanie

Napisz program, który mając dane liczby rodzynek w poszczególnych kawałkach czekolady, wyznaczy minimalną liczbę rodzynek, które Bonny będzie musiała oddać Chytremu Piotrowi.

Ograniczenia

$1 \leqslant N, M \leqslant 50$ — liczby kawałków na każdym z boków czekolady
$1 \leqslant R_{k,p} \leqslant 1000$ — liczba rodzynek w kawałku czekolady w k-tym wierszu i p-tej kolumnie

Wejście

Twój program powinien wczytać ze standardowego wejścia następujące dane:

- *Pierwszy wiersz zawiera liczby całkowite N i M, oddzielone pojedynczym odstępem.*
- *Kolejne N wierszy zawiera informacje o rozmieszczeniu rodzynek w poszczególnych kawałkach czekolady. k-ty z tych N wierszy opisuje k-ty wiersz czekolady i zawiera M liczb całkowitych pooddzielanych pojedynczymi odstępami. Liczby te opisują kawałki w danym wierszu, od lewej do prawej. p-ta liczba w k-tym wierszu (spośród N wierszy, o których mowa) to liczba rodzynek, które znajdują się w kawałku czekolady w k-tym wierszu i p-tej kolumnie.*

Wyjście

Twój program powinien wypisać na standardowe wyjście jeden wiersz zawierający jedną liczbę całkowitą: minimalną liczbę rodzynek, które Bonny musi oddać Chytremu Piotrowi.

Ocenianie

W testach wartych łącznie 25 punktów N i M nie przekroczą 7.

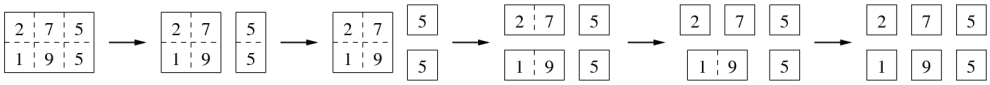
Przykład

Dla danych wejściowych:

```
2 3
2 7 5
1 9 5
```

poprawnym wynikiem jest:

```
77
```



Oto jeden z możliwych sposobów (spośród wielu) osiągnięcia kosztu 77 rodzynek.

Pierwszym cięciem, o które prosi Piotra Bonny, jest oddzielenie trzeciej kolumny od reszty czekolady. Bonny płaci za to 29 rodzynek.

Następnie, Bonny daje Piotrowi mniejszy z fragmentów — ten złożony z dwóch kawałków zawierających po 5 rodzynek — a Piotr przecina go na pół za 10 rodzynek.

Później Bonny daje Piotrowi największy z pozostałych fragmentów — ten zawierający kawałki z 2, 7, 1 oraz 9 rodzynkami. Bonny prosi o poziome cięcie, które oddzieli pierwszy wiersz od drugiego, i płaci 19 rodzynek.

Następnie, Bonny daje Piotrowi lewy górny fragment, płacąc 9 rodzynek. W końcu Bonny prosi Piotra o podzielenie lewego dolnego fragmentu za 10 rodzynek.

Całkowity koszt, jaki ponosi Bonny, to $29 + 10 + 19 + 9 + 10 = 77$ rodzynek. Żaden inny schemat cięć nie podzieli tej czekolady na 6 kawałków mniejszym kosztem.

Marcin Kubica, Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

IOI 2009, dzień pierwszy, 10.08.2009

Zatrudnianie

Chcesz zatrudnić robotników do pewnego projektu budowlanego. Do pracy zaaplikowało N kandydatów, ponumerowanych od 1 do N. Jeśli kandydat k zostanie zatrudniony, trzeba mu zapłacić co najmniej S_k dolarów. Ponadto, każdy kandydat k posiada poziom kwalifikacji Q_k. Prawo budowlane wymaga, żebyś płacił swoim robotnikom proporcjonalnie do ich poziomu kwalifikacji, relatywnie względem wszystkich pozostałych. Na przykład, jeżeli zatrudnisz dwóch robotników A i B oraz $Q_A = 3 \cdot Q_B$, to będziesz musiał zapłacić robotnikowi A dokładnie trzy razy więcej niż robotnikowi B. Wypłaty robotników mogą być niecałkowite. Uwzględniamy nawet liczby, których nie można zapisać za pomocą skończonej liczby cyfr w postaci dziesiętnej, takie jak jedna trzecia czy jedna szósta dolara.

Dysponujesz W dolarami i chcesz zatrudnić tak wielu robotników, jak tylko się da. Ty decydujesz o tym, kogo zatrudnisz i ile komu zapłacisz, ale musisz spełnić wymagania dotyczące minimalnych wypłat tych, których zatrudnisz, i musisz przestrzegać przepisów budowlanych. Ponadto musisz zmieścić się w budżecie W dolarów.

W twoim projekcie poziomy kwalifikacji zupełnie nie grają roli, więc jesteś zainteresowany wyłącznie maksymalizacją liczby zatrudnionych robotników, niezależnie od ich poziomów kwalifikacji. Jednakże, jeśli istnieje więcej niż jeden sposób osiągnięcia tego celu, chciałbyś wybrać taki sposób, w którym łączna ilość pieniędzy, które musisz zapłacić robotnikom, jest najmniejsza możliwa. Jeżeli istnieje wiele sposobów osiągnięcia tego celu, to wszystkie one są dla ciebie równie dobre i zadowolisz się dowolnym z nich.

Zadanie

Napisz program, który mając dane wymagania dotyczące wypłat oraz poziomów kwalifikacji robotników, a także ilość pieniędzy będących w twoim posiadaniu, wyznaczy, których kandydatów powinieneś zatrudnić. Powinieneś zatrudnić tak wielu z nich, jak tylko się da, a ponadto zrealizować to za pomocą najmniejszej możliwej ilości pieniędzy, nie przekraczając opisanego powyżej prawa budowlanego.

Ograniczenia

$1 \leqslant N \leqslant 500\,000$ — *liczba kandydatów*
$1 \leqslant S_k \leqslant 20\,000$ — *wymagana przez kandydata k pensja minimalna*
$1 \leqslant Q_k \leqslant 20\,000$ — *poziom kwalifikacji kandydata k*
$1 \leqslant W \leqslant 10\,000\,000\,000$ — *ilość pieniędzy, którymi dysponujesz*

Ważna uwaga

Maksymalna wartość parametru W nie mieści się w 32 bitach. Powinieneś użyć 64-bitowego typu danych, takiego jak typ `long long` *w C/C++ czy* `int64` *w Pascalu, żeby przechowywać wartość W w jednej zmiennej.*

Wejście

Twój program powinien wczytać ze standardowego wejścia następujące dane:

- *Pierwszy wiersz zawiera liczby całkowite N oraz W, oddzielone pojedynczym odstępem.*
- *Kolejne N wierszy zawiera opisy kandydatów, po jednym w wierszu. k-ty z tych wierszy opisuje kandydata numer k i zawiera liczby całkowite S_k oraz Q_k, oddzielone pojedynczym odstępem.*

Wyjście

Twój program powinien wypisać na standardowe wyjście następujące dane:

- *Pierwszy wiersz powinien zawierać jedną liczbę całkowitą H — liczbę robotników, których zatrudnisz.*
- *Kolejne H wierszy powinno zawierać numery identyfikacyjne kandydatów, których postanowiłeś zatrudnić (każdy z nich powinien być inną liczbą z zakresu od 1 do N), po jednym w wierszu, w dowolnej kolejności.*

Ocenianie

W każdym z testów uzyskasz pełną punktację, jeżeli twój wybór kandydatów osiągnie wszystkie cele i spełni wszystkie wymagania. Jeżeli stworzysz plik wyjściowy, w którym pierwszy wiersz będzie poprawny (tzn. z poprawną wartością parametru H), ale który nie spełnia powyższego opisu, uzyskasz 50% punktów za ten test. Stanie się tak nawet w przypadku, gdy wyjście nie będzie poprawnie sformatowane, o ile tylko pierwszy wiersz będzie poprawny.

W testach wartych łącznie 50 punktów N nie przekroczy 5 000.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`4 100`	`2`
`5 1000`	`2`
`10 100`	`3`
`8 10`	
`20 1`	

Jedynym sposobem na to, żeby było cię stać na zatrudnienie dwóch robotników i żeby wciąż spełnić wszystkie wymagania, jest wybranie robotników o numerach 2 i 3. Możesz zapłacić im odpowiednio 80 i 8 dolarów i w ten sposób zmieścisz się w budżecie równym 100.

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`3 4`	`3`
`1 2`	`1`
`1 3`	`2`
`1 3`	`3`

W tym przypadku stać cię na zatrudnienie wszystkich robotników. Zapłacisz 1 dolara robotnikowi 1 i 1,50 dolara każdemu z robotników 2 i 3, i w ten sposób zdołasz zatrudnić wszystkich, używając posiadanych przez ciebie 4 dolarów.

Dla danych wejściowych:	*poprawnym wynikiem jest:*
3 40	2
10 1	2
10 2	3
10 3	

W tym przypadku nie stać cię na zatrudnienie wszystkich robotników, gdyż kosztowałoby cię to 60 dolarów, lecz możesz zatrudnić dowolnych dwóch. Postanawiasz zatrudnić robotników 2 i 3, ponieważ kosztuje cię to najmniej pieniędzy w porównaniu do pozostałych kombinacji złożonych z par robotników. Możesz zapłacić 10 dolarów robotnikowi 2 i 15 dolarów robotnikowi 3, wydając łącznie 25 dolarów. Gdybyś zatrudnił robotników 1 i 2, musiałbyś im zapłacić odpowiednio co najmniej 10 i 20 dolarów. Gdybyś zatrudnił robotników 1 i 3, musiałbyś im zapłacić odpowiednio co najmniej 10 i 30 dolarów.

Marcin Kubica, Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

IOI 2009, dzień drugi, 12.08.2009

Parking

Na pewnym parkingu znajduje się N miejsc ponumerowanych od 1 do N. Mechanizm działania tego parkingu jest następujący. Każdego ranka przed otwarciem parking jest pusty. Jak tylko jakiś samochód przyjeżdża na parking, stróże sprawdzają, czy są jakieś wolne miejsca parkingowe. Jeżeli nie, to samochód musi czekać przy wjeździe na zwolnienie się jakiegoś miejsca. Jeśli jakieś miejsce parkingowe jest wolne lub w momencie, gdy któreś staje się wolne, samochód zostaje zaparkowany na wolnym miejscu. Jeżeli istnieje więcej niż jedno wolne miejsce parkingowe, samochód parkuje w miejscu o najmniejszym numerze. Jeśli jakiś samochód oczekuje na miejsce i w tym czasie przybywa więcej samochodów, wszystkie one ustawiają się w kolejce przy wjeździe w kolejności, w której przyjechały. Jeśli wówczas jakieś miejsce parkingowe zwolni się, parkuje tam pierwszy w kolejce samochód (tzn. ten, który przybył najwcześniej).

Koszt zaparkowania samochodu (w dolarach) jest iloczynem jego masy wyrażonej w kilogramach oraz stawki związanej z jego miejscem parkingowym. Koszt ten nie zależy od tego, jak długo samochód pozostaje na parkingu.

Właściciel parkingu wie, że dzisiaj przybędzie tam M samochodów, a także zna kolejność ich przyjazdów i wyjazdów. Pomóż mu obliczyć, na ile dolarów zysku może dzisiaj liczyć.

Zadanie

Napisz program, który mając dane stawki związane z miejscami parkingowymi, masy samochodów oraz kolejność, w której samochody przyjeżdżają i wyjeżdżają, obliczy łączny zysk właściciela parkingu.

Ograniczenia

$1 \leqslant N \leqslant 100$ — *liczba miejsc parkingowych*
$1 \leqslant M \leqslant 2\,000$ — *liczba samochodów*
$1 \leqslant R_s \leqslant 100$ — *stawka za miejsce parkingowe s, w dolarach na kilogram*
$1 \leqslant W_k \leqslant 10\,000$ — *masa samochodu k, w kilogramach*

Wejście

Twój program powinien wczytać ze standardowego wejścia następujące dane:

- *Pierwszy wiersz zawiera liczby całkowite N oraz M oddzielone pojedynczym odstępem.*
- *Kolejne N wierszy opisuje stawki za poszczególne miejsca parkingowe. s-ty z tych wierszy zawiera jedną liczbę całkowitą R_s — stawkę za miejsce parkingowe numer s, w dolarach na kilogram.*
- *Kolejne M wierszy opisuje masy samochodów. Samochody są ponumerowane od 1 do M w jakiejkolwiek kolejności. k-ty z tych M wierszy zawiera jedną liczbę całkowitą W_k — masę samochodu numer k, w kilogramach.*

- *Kolejne $2 \cdot M$ wierszy opisuje przyjazdy i wyjazdy wszystkich samochodów w porządku chronologicznym. Dodatnia liczba całkowita i oznacza, że samochód numer i przyjeżdża na parking. Ujemna liczba całkowita $-i$ oznacza, że samochód numer i wyjeżdża z parkingu. Żaden samochód nie wyjedzie z parkingu przed swoim przyjazdem i każdy z M samochodów pojawi się dokładnie dwukrotnie w tym ciągu, z czego pierwsze pojawienie będzie oznaczało przyjazd, a drugie wyjazd. Dodatkowo, żaden samochód nie wyjedzie z parkingu przed zaparkowaniem (tzn. nigdy nie wyjedzie żaden samochód oczekujący w kolejce).*

Wyjście

Twój program powinien wypisać na standardowe wyjście jeden wiersz zawierający jedną liczbę całkowitą: łączną liczbę dolarów, które zarobi dziś właściciel parkingu.

Ocenianie

W testach wartych 40 punktów, w momencie przyjazdu każdego samochodu będzie co najmniej jedno wolne miejsce parkingowe. W tych testach żaden samochód nie będzie musiał czekać na miejsce.

Przykład

Dla danych wejściowych:

```
3 4
2
3
5
200
100
300
800
3
2
-3
1
4
-4
-2
-1
```

poprawnym wynikiem jest:

```
5300
```

Samochód numer 3 zajmuje miejsce numer 1 i płaci $300 \cdot 2 = 600$ dolarów.
Samochód numer 2 zajmuje miejsce numer 2 i płaci $100 \cdot 3 = 300$ dolarów.
Samochód numer 1 zajmuje miejsce numer 1 (które zostało zwolnione przez samochód numer 3) i płaci $200 \cdot 2 = 400$ dolarów.
Samochód numer 4 zajmuje miejsce numer 3 (ostatnie wolne) i płaci $800 \cdot 5 = 4\,000$ dolarów.

Dla danych wejściowych:

```
2 4
5
2
100
500
1000
2000
3
1
2
4
-1
-3
-2
-4
```

poprawnym wynikiem jest:

```
16200
```

Samochód numer 3 zajmuje miejsce numer 1 i płaci $1\,000 \cdot 5 = 5\,000$ dolarów.
Samochód numer 1 zajmuje miejsce numer 2 i płaci $100 \cdot 2 = 200$ dolarów.
Samochód numer 2 przyjeżdża i musi poczekać przy wjeździe.
Samochód numer 4 przyjeżdża i musi poczekać przy wjeździe za samochodem numer 2.
Kiedy samochód numer 1 zwalnia swoje miejsce parkingowe, samochód numer 2 zajmuje je i płaci $500 \cdot 2 = 1\,000$ dolarów.
Kiedy samochód numer 3 zwalnia swoje miejsce parkingowe, samochód numer 4 zajmuje je i płaci $2\,000 \cdot 5 = 10\,000$ dolarów.

Marcin Kubica, Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

IOI 2009, dzień drugi, 12.08.2009

Komiwojażer

Komiwojażer stwierdził, iż optymalne planowanie podróży na powierzchni ziemi jest zbyt trudnym problemem obliczeniowym, więc przenosi swój interes w liniowy świat na Dunaju. Ów komiwojażer posiada bardzo szybką łódź, którą może przepłynąć pomiędzy dowolnymi punktami na rzece w pomijalnym czasie, ale niestety łódź ta zużywa ogromne ilości paliwa — każdy metr przepłynięty w górę rzeki (w kierunku jej źródła) kosztuje komiwojażera U dolarów, a każdy metr przepłynięty w dół rzeki (w przeciwnym kierunku) — D dolarów.

Wzdłuż rzeki jest rozmieszczonych N targów, które komiwojażer chciałby odwiedzić. Każdy targ odbywa się tylko przez jeden dzień. Dla każdego targu X, komiwojażer zna jego datę T_X, wyrażoną jako liczba dni od zakupu jego łodzi. Zna też lokalizację targu L_X, będącą jego odległością w metrach od źródła rzeki (w dół rzeki), jak również liczbę dolarów M_X, którą zarobi, jeśli odwiedzi ten targ. Powinien zacząć i zakończyć swoją podróż w swoim domu nad brzegiem rzeki, który znajduje się na pozycji S, także wyrażonej jako odległość w metrach od źródła rzeki.

Pomóż komiwojażerowi wybrać targi, które powinien odwiedzić (potencjalnie żadne), oraz ich kolejność, tak aby zmaksymalizować zysk na końcu podróży. Zysk komiwojażera wyraża się jako suma dolarów uzyskanych na odwiedzonych targach minus suma dolarów wydanych na podróże w górę i w dół rzeki.

Pamiętaj, że jeśli targ A odbywa się przed targiem B, komiwojażer może odwiedzić je tylko w tej kolejności (tzn. nie może najpierw odwiedzić B, a potem A). Jeśli jednak dwa targi odbywają się tego samego dnia, komiwojażer może odwiedzić je w dowolnej kolejności. Nie ma ograniczenia na liczbę targów odwiedzonych jednego dnia, ale oczywiście komiwojażer nie może odwiedzić dwa razy tego samego targu i zgarnąć zysków dwukrotnie. Może natomiast przepłynąć obok odwiedzonych wcześniej targów bez dodatkowego zysku.

Zadanie

Napisz program, który mając dane daty, lokalizacje i możliwe zyski z poszczególnych targów, jak również lokalizację domu komiwojażera i jego koszty podróżowania, wyznaczy maksymalny możliwy do uzyskania zysk na końcu podróży.

Ograniczenia

$1 \leqslant N \leqslant 500\,000$ — *liczba targów*
$1 \leqslant D \leqslant U \leqslant 10$ — *koszt podróżowania w górę rzeki (U) i w jej dół (D)*
$1 \leqslant S \leqslant 500\,001$ — *lokalizacja domu komiwojażera*
$1 \leqslant T_k \leqslant 500\,000$ — *dzień, w którym odbywa się targ k*
$1 \leqslant L_k \leqslant 500\,001$ — *lokalizacja targu k*
$1 \leqslant M_k \leqslant 4\,000$ — *liczba dolarów, którą zarobi komiwojażer, jeśli odwiedzi targ k*

Wejście

Twój program powinien wczytać ze standardowego wejścia następujące dane:

- *Pierwszy wiersz zawiera liczby całkowite N, U, D oraz S, w tej kolejności, pooddzielane pojedynczymi odstępami.*
- *Kolejne N wierszy opisuje N targów, w jakiejkolwiek kolejności. k-ty z tych N wierszy opisuje k-ty targ i zawiera trzy liczby całkowite pooddzielane pojedynczymi odstępami: dzień odbywania się targu T_k, jego lokalizację L_k oraz zysk M_k możliwy do uzyskania na nim przez komiwojażera.*

Uwaga: *Wszystkie lokalizacje podane na wejściu będą różne. Dokładniej, żadne dwa targi nie będą odbywały się w tym samym miejscu i żaden targ nie odbędzie się w domu komiwojażera.*

Wyjście

Twój program powinien wypisać na standardowe wyjście jeden wiersz zawierający jedną liczbę całkowitą: maksymalny zysk, jaki może uzyskać komiwojażer po odbyciu swej podróży.

Ocenianie

W testach wartych łącznie 60 punktów żadne dwa targi nie będą odbywać się tego samego dnia.
W testach wartych łącznie 40 punktów żadna z liczb na wejściu nie przekroczy 5 000.
Testy, w których zachodzą oba powyższe warunki, są warte 15 punktów.
Testy, w których zachodzi co najmniej jeden z tych warunków, są warte 85 punktów.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*

`4 5 3 100`
`2 80 100`
`20 125 130`
`10 75 150`
`5 120 110`
	`50`

Optymalny plan podróży przewiduje odwiedzenie targów 1 i 3 (tych o lokalizacjach 80 i 75). Kolejność działań oraz odpowiadające im zyski i koszty są następujące:

- *Komiwojażer podróżuje 20 metrów w górę rzeki kosztem 100 dolarów. Dotychczasowy zysk: -100.*
- *Odwiedza targ numer 1 i zyskuje 100. Dotychczasowy zysk: 0.*
- *Podróżuje 5 metrów w górę rzeki, ponosząc koszt 25. Dotychczasowy zysk: -25.*
- *Odwiedza targ numer 3, gdzie zarabia 150. Dotychczasowy zysk: 125.*
- *Podróżuje 25 metrów w dół rzeki i tym samym wraca do domu, ponosząc koszt 75. Zysk końcowy: 50.*

Marcin Kubica, Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

IOI 2009, dzień drugi, 12.08.2009

Mecho

Misio Mecho znalazł sekretny pszczeli skarb — miodek! Właśnie w najlepsze zajadał sobie ów miodek, gdy nagle został zauważony przez jedną z pszczół, a ta wszczęła pszczeli alarm. Misio zdaje sobie sprawę, że w tym samym momencie hordy pszczół zaczęły wylatywać z uli i rozprzestrzeniać się, starając się go złapać. Wie, że musi zostawić miodek i uciekać do swojego domku, ale miodek jest tak pyszny... że misio nie chce zacząć uciekać przedwcześnie. Pomóż misiowi określić ostatni możliwy moment, w którym może porzucić miodek.

Las, w którym rzecz się dzieje, jest przedstawiony jako kwadrat podzielony „w kratkę" na $N \times N$ kwadratów jednostkowych, o bokach ułożonych w kierunkach północ-południe i wschód-zachód. W każdym z kwadratów jednostkowych znajduje się drzewo, trawa, ul lub domek misia Mecho. Dwa kwadraty jednostkowe nazwiemy sąsiednimi, jeśli jeden z nich styka się z drugim bezpośrednio od północy, południa, wschodu lub zachodu (ale nie po przekątnej). Misio nie porusza się zbyt zgrabnie — każdy jego krok musi prowadzić do sąsiedniego kwadratu jednostkowego. Misio może chodzić tylko po trawie, a nie potrafi przedzierać się przez drzewa ani ule. Ponadto, w ciągu minuty może wykonać co najwyżej S kroków.

W chwili, gdy pszczoły wszczęły alarm, misio stoi w trawiastym kwadracie jednostkowym, w którym znajduje się miodek, a pszczoły znajdują się w każdym kwadracie jednostkowym zawierającym ul (w lesie może być więcej niż jeden ul). Od tej chwili, w ciągu każdej minuty dzieją się następujące rzeczy, w podanej kolejności:

- *Jeśli misio Mecho wciąż zajada miodek, musi zdecydować, czy kontynuować wyżerkę, czy też zacząć uciekać. Jeśli postanawia jeść nadal, to do końca minuty nie przemieszcza się. W przeciwnym razie natychmiast zaczyna uciekać i wykonuje co najwyżej S kroków, zgodnie z podanymi powyżej zasadami. Misio nie może zabrać ze sobą miodku, więc jak tylko zacznie uciekać, przestaje jeść miodek.*
- *Po tym, jak misio Mecho przełknie już miodek jedzony w danej minucie lub wykona wszystkie kroki w danej minucie, pszczoły rozprzestrzeniają się o jeden kwadrat jednostkowy, ale tylko na trawiaste kwadraty jednostkowe. Dokładniej, pszczoły rozprzestrzeniają się na każdy trawiasty kwadrat jednostkowy, który sąsiaduje z kwadratem jednostkowym już zajętym przez pszczoły. Co więcej, jeśli w jakimś kwadracie znajdują się pszczoły, to będą się tam już zawsze znajdować (czyli rój pszczół nie przemieszcza się, tylko rozprzestrzenia).*

Inaczej mówiąc, pszczoły rozprzestrzeniają się w następujący sposób. Gdy zostaje wszczęty alarm, pszczoły zajmują tylko te kwadraty, w których są ule. Pod koniec pierwszej minuty zajmują również wszystkie trawiaste kwadraty jednostkowe przylegające do uli. Pod koniec drugiej minuty zajmują również wszystkie trawiaste kwadraty jednostkowe przylegające do trawiastych kwadratów jednostkowych przylegających do uli, itd. Po odpowiednio długim czasie pszczoły zajmą wszystkie trawiaste kwadraty jednostkowe, do których uda im się dotrzeć.

Ani misio Mecho, ani pszczoły nie mogą opuścić lasu. Zauważ, że zgodnie z podanymi zasadami, Mecho będzie zajadał miodek przez całkowitą liczbę minut.

Pszczoły dopadają misia, jeśli w jakimkolwiek momencie Mecho znajduje się w kwadracie jednostkowym zajętym przez pszczoły.

Zadanie

Napisz program, który na podstawie danej mapy lasu określi maksymalną liczbę minut, przez które misio Mecho może zajadać miodek, tak żeby nadal mógł uciec do swojego domku, zanim dopadną go pszczoły.

Ograniczenia

$1 \leqslant N \leqslant 800$ — *rozmiar mapy (długość jej boku)*
$1 \leqslant S \leqslant 1\,000$ — *maksymalna liczba kroków, które misio Mecho może wykonać w ciągu minuty*

Wejście

Twój program powinien wczytać ze standardowego wejścia następujące dane:

- *W pierwszym wierszu znajdują się dwie liczby całkowite N i S, oddzielone pojedynczym odstępem.*
- *Kolejne N wierszy zawiera opis mapy lasu. Każdy z tych wierszy zawiera N znaków, a każdy znak przedstawia jeden kwadrat jednostkowy. Możliwe znaki i ich znaczenia są następujące:*

  `T` *oznacza drzewo,*
  `G` *oznacza trawę,*
  `M` *oznacza początkowe położenie misia Mecho oraz miodku, znajdujące się na trawie,*
  `D` *oznacza domek misia Mecho, w którym misio może się skryć, a pszczoły nie mają do niego wstępu,*
  `H` *oznacza ul.*

Uwaga: *Możesz założyć, że na mapie znajduje się dokładnie jedna litera* `M`*, dokładnie jedna litera* `D` *i przynajmniej jedna litera* `H`*. Możesz również założyć, że istnieje ciąg sąsiadujących ze sobą liter* `G` *łączących misia Mecho z jego domkiem, a także ciąg sąsiadujących ze sobą liter* `G` *łączących przynajmniej jeden z uli z miodkiem (czyli początkowym położeniem misia). W szczególności, każdy z tych ciągów może być pusty — gdy domek misia lub ul sąsiaduje z miodkiem. Zwróć uwagę, że pszczoły nie mogą przelecieć przez domek misia ani nad nim. Jest on dla nich niczym drzewo.*

Wyjście

Twój program powinien wypisać na standardowe wyjście jeden wiersz zawierający jedną liczbę całkowitą: maksymalną liczbę minut, przez które misio Mecho może zajadać miodek, znajdując się w swoim początkowym położeniu, tak aby nadal mógł uciec do swojego domku, zanim dopadną go pszczoły.

Jeśli Mecho w ogóle nie ma szansy uciec do domku, zanim dopadną go pszczoły, Twój program powinien zamiast tego wypisać na standardowe wyjście liczbę -1.

Ocenianie

W testach wartych łącznie 40 punktów N nie przekroczy 60.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*

`7 3`
`TTTTTTT`
`TGGGGGT`
`TGGGGGT`
`MGGGGGD`
`TGGGGGT`
`TGGGGGT`
`THHHHHT`
	`1`

Mecho może zajadać miodek przez minutę, po czym może uciekać najkrótszą drogą, prosto na prawo, i po dwóch minutach będzie bezpieczny w domku.

Dla danych wejściowych:	*poprawnym wynikiem jest:*

`7 3`
`TTTTTTT`
`TGGGGGT`
`TGGGGGT`
`MGGGGGD`
`TGGGGGT`
`TGGGGGT`
`TGHHGGT`
	`2`

Mecho może zajadać miodek przez dwie minuty, w trzeciej minucie może wykonać kroki $\rightarrow\uparrow\rightarrow$, w czwartej minucie kroki $\rightarrow\rightarrow\rightarrow$ i w piątej minucie kroki $\downarrow\rightarrow$.

Marcin Kubica, Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

IOI 2009, dzień drugi, 12.08.2009

Regiony

Agencja Rozwoju Regionalnego Narodów Zjednoczonych (UNRDA) ma dobrze zdefiniowaną strukturę organizacyjną. Zatrudnia łącznie N osób, z których każda pochodzi z jednego z R regionów geograficznych świata. Pracownicy są ponumerowani od 1 do N w porządku ważności, przy czym pracownik numer 1, Dyrektor, jest najważniejszą osobą w agencji. Regiony są ponumerowane od 1 do R w jakimkolwiek porządku. Każdy pracownik z wyjątkiem Dyrektora posiada jednego bezpośredniego przełożonego. Przełożony jest zawsze ważniejszy niż każdy z jego pracowników.

Powiemy, że pracownik A jest menadżerem pracownika B, wtedy i tylko wtedy, gdy A jest przełożonym B lub A jest menadżerem przełożonego B. W ten sposób, na przykład, Dyrektor jest menadżerem każdego z pozostałych pracowników. Ponadto, oczywiście, żadna para pracowników nie może być wzajemnie swoimi menadżerami.

Niestety, Biuro Śledcze Narodów Zjednoczonych (UNBI) otrzymało ostatnio szereg skarg na to, że struktura organizacyjna UNRDA nie jest zrównoważona i wyróżnia pewne regiony świata w stosunku do innych. Aby zweryfikować te oskarżenia, UNBI potrzebuje systemu komputerowego, który dysponując strukturą organizacyjną UNRDA, mógłby odpowiadać na zapytania postaci: dla dwóch różnych regionów r_1 i r_2, ile jest par pracowników agencji e_1 i e_2, takich że pracownik e_1 pochodzi z regionu r_1, pracownik e_2 pochodzi z regionu r_2 oraz e_1 jest menadżerem e_2. Każde zapytanie ma dwa parametry — regiony r_1 i r_2 — natomiast jego wynikiem jest jedna liczba całkowita — liczba różnych par e_1 i e_2, które spełniają wyżej wymienione warunki.

Zadanie

Napisz program, który mając dane regiony pochodzenia wszystkich pracowników agencji, a także dane o tym, kto jest czyim przełożonym, będzie odpowiadał na opisane wyżej zapytania w sposób interaktywny.

Ograniczenia

$1 \leqslant N \leqslant 200\,000$ — *liczba pracowników*
$1 \leqslant R \leqslant 25\,000$ — *liczba regionów*
$1 \leqslant Q \leqslant 200\,000$ — *liczba zapytań, na które ma odpowiedzieć twój program*
$1 \leqslant H_k \leqslant R$ — *region pochodzenia pracownika k (przy czym $1 \leqslant k \leqslant N$)*
$1 \leqslant S_k < k$ — *przełożony pracownika k (przy czym $2 \leqslant k \leqslant N$)*
$1 \leqslant r_1, r_2 \leqslant R$ — *regiony pojawiające się w danym zapytaniu*

Wejście

Twój program powinien wczytać ze standardowego wejścia następujące dane:

- *Pierwszy wiersz zawiera liczby całkowite N, R i Q, w tej kolejności, pooddzielane pojedynczymi odstępami.*
- *Kolejne N wierszy opisuje N pracowników agencji w porządku ważności. k-ty z tych N wierszy opisuje pracownika numer k. Pierwszy z tych wierszy (tzn. ten opisujący Dyrektora) zawiera jedną liczbę całkowitą: region H_1 pochodzenia Dyrektora. Każdy z pozostałych $N-1$ wierszy zawiera dwie liczby całkowite oddzielone pojedynczym odstępem: identyfikator S_k przełożonego k-tego pracownika oraz region H_k pochodzenia k-tego pracownika.*

Interakcja

Po wczytaniu danych wejściowych, twój program powinien zacząć na przemian wczytywać zapytania ze standardowego wejścia i wypisywać wyniki na standardowe wyjście. Odpowiedzi na każde z Q zapytań muszą być udzielane pojedynczo, tzn. twój program musi udzielić odpowiedzi na już otrzymane zapytanie, zanim otrzyma następne zapytanie.

Każde zapytanie jest zawarte w jednym wierszu standardowego wejścia i składa się z dwóch różnych liczb całkowitych oddzielonych pojedynczym odstępem — regionów r_1 oraz r_2.

Odpowiedź na każde zapytanie musi być zawarta w jednym wierszu standardowego wyjścia, zawierającym jedną liczbę całkowitą — liczbę par pracowników UNRDA e_1 i e_2, takich że regionem pochodzenia e_1 jest r_1, regionem pochodzenia e_2 jest r_2 oraz e_1 jest menadżerem e_2.

Uwaga: *Dane testowe będą tak dobrane, że poprawna odpowiedź na każde z zapytań podanych na standardowym wejściu będzie zawsze mniejsza niż 1 000 000 000.*

Ważna uwaga: *Aby interakcja twojego programu ze sprawdzaczką była prawidłowa, twój program musi wykonać flusha na standardowym wyjściu po każdej odpowiedzi na zapytanie. Ponadto należy uważać na przypadkowe zablokowania podczas wczytywania standardowego wejścia, co może nastąpić na przykład w przypadku użycia instrukcji* `scanf("%d\n")`.

Ocenianie

W testach wartych łącznie 30 punktów R nie przekroczy 500.
W testach wartych łącznie 55 punktów z żadnego regionu nie będzie pochodziło więcej niż 500 pracowników.
Testy, w których zachodzą oba powyższe warunki, są warte 15 punktów.
Testy, w których zachodzi co najmniej jeden z tych dwóch warunków, są warte 70 punktów.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*

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

	`1 [flush standard output]`
`1 3`	
	`3 [flush standard output]`
`2 3`	
	`2 [flush standard output]`
`3 1`	
	`1 [flush standard output]`

XV Bałtycka Olimpiada Informatyczna,

Sztokholm, Szwecja 2009

Tomasz Kulczyński, Jakub Pawlewicz
Tłumaczenie

BOI 2009, dzień próbny, 18.04.2009

Transmisja radiowa

Stacja nadawcza chce przesłać wiadomość do wielu odbiorców. Aby mieć pewność, że wszyscy nasłuchujący ją odbiorą, wiadomość jest nadawana w kółko w nieskończonej pętli.

Masz dany ciąg znaków zarejestrowany przez jednego z odbiorców. Wiadomo, że ciąg jest co najmniej tak długi jak oryginalna wiadomość.

Twoim zadaniem jest napisanie programu, który odnajdzie w ciągu wiadomość nadaną przez stację. Dokładniej, twój program musi znaleźć najkrótsze podsłowo S' słowa S takie, że z kolei S jest podsłowem odpowiednio wielokrotnego powtórzenia S' ($S' + S' + \ldots + S'$).

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą L, oznaczającą długość ciągu S. Drugi wiersz zawiera dokładnie L znaków — samo słowo S. Składa się ono z małych liter alfabetu angielskiego (`a...z`*).*

Wyjście

Program powinien wypisać na standardowe wyjście jeden wiersz zawierający tylko jedną liczbę całkowitą: długość L' wiadomości S'. Zauważ, że L' ma być możliwie najmniejsze.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`8` `cabcabca`	`3`

Wiadomością mogło być `abc`, `cab` *albo* `abcabc`, *ale nie mogło nią być żadne słowo krótsze niż 3 znaki.*

Ograniczenia

$1 < L \leqslant 1\,000\,000$

Pamiętaj, że funkcje do szukania wzorców w tekście zawarte w standardowych bibliotekach (`strstr` *w C,* `string::find` *w C++ lub* `pos` *w Pascalu) w najgorszym przypadku potrzebują czasu $\Theta(nm)$, aby znaleźć słowo długości n w słowie długości m.*

Tomasz Kulczyński, Jakub Pawlewicz
Tłumaczenie

BOI 2009, dzień pierwszy, 19.04.2009

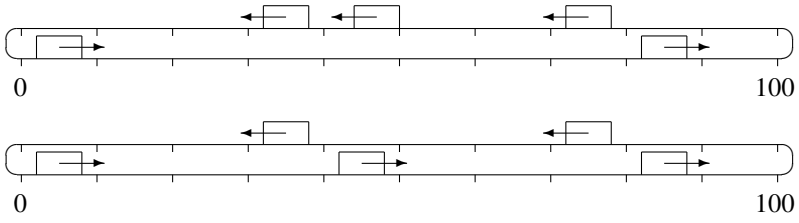
Błąd sygnalizacyjny w metrze

W Sztokholmie jest kilka linii metra. W tym zadaniu rozważymy jedną, zadaną linię i problem, który dosyć często powstaje w wyniku „błędu sygnalizacyjnego".

Linię metra przedstawiamy jako dwa równoległe tory połączone na końcach. Pociągi na górnym torze jadą z prawej strony do lewej, a na dolnym torze z lewej do prawej. Kiedy pociąg dociera do końca torów, zawraca, zmieniając tory na przeciwne.

Jeśli metro działa normalnie, ruch jest ciągły, a pociągi poruszają się ze stałą prędkością (jedna jednostka długości na jednostkę czasu). Pociągi są rozmieszczone równomiernie, tzn. w dowolnym ustalonym punkcie torów pociągi pojawiają się w równych odstępach czasu. Zakładamy, że czas potrzebny na zatrzymanie i zabranie pasażerów oraz nawrót na końcu linii jest zaniedbywalny.

Wskutek błędów sygnalizacyjnych pociągi zostały losowo rozmieszczone wzdłuż linii metra. Twoim obowiązkiem, jako kierownika ruchu pociągów, jest jak najszybsze doprowadzenie do ponownego równomiernego rozmieszczenia pociągów na linii. Napisz program, który dla aktualnych pozycji pociągów wyznaczy, jak szybko można to osiągnąć. Możesz wydawać polecenia pociągom, aby zatrzymywały się na pewien czas i/lub zmieniały kierunek poruszania się w dowolnym miejscu linii. W wyniku zmiany kierunku pociąg zmienia jedne tory na drugie.



Rys. 1: Tory mają długość 100. Pociągi znajdują się na pozycjach 5 (ruch w prawo), 35 (lewo), 46 (lewo), 75 (lewo) i 85 (prawo). Aby pociągi znów były równomiernie rozmieszczone, możemy na przykład przemieścić pociąg z pozycji 46 jedną jednostkę w lewo i zmienić jego kierunek. To zajmuje jedną jednostkę czasu, aczkolwiek nie jest to rozwiązanie optymalne; patrz przykład niżej.

Wejście

Dane należy wczytać ze standardowego wejścia. Pierwszy wiersz wejścia zawiera dwie liczby całkowite oddzielone pojedynczym odstępem — długość m torów oraz liczba n pociągów na danej linii. Każdy z następnych n wierszy opisuje bieżącą pozycję jednego pociągu, wyrażoną za pomocą liczby całkowitej x_i oraz kierunku (`L` *to lewo,* `R` *to prawo), oddzielonych pojedynczym odstępem.*

Wyjście

Twój program powinien wypisać na standardowe wyjście jeden wiersz zawierający minimalny czas potrzebny do równomiernego rozmieszczenia pociągów. Wynik nie może różnić się co do wartości bezwzględnej o więcej niż 10^{-6} od wartości optymalnej.

Przykład

Dla danych wejściowych:

```
100 5
5 R
35 L
46 L
75 L
85 R
```

poprawnym wynikiem jest:

```
0.5
```

natomiast dla danych:

```
100 8
9 L
15 R
41 L
33 L
81 R
33 R
100 L
97 R
```

poprawnym wynikiem jest:

```
15.500000
```

Ograniczenia

$100 \leqslant m \leqslant 100\,000\,000$
$1 \leqslant n \leqslant 100\,000$
$0 \leqslant x_i \leqslant m$

Ocenianie

W testach wartych 50% punktów zachodzi $n \leqslant 200$.

Tomasz Kulczyński, Jakub Pawlewicz
Tłumaczenie

BOI 2009, dzień pierwszy, 19.04.2009

Chrząszcz

Na cienkiej, poziomej gałęzi siedzi chrząszcz. „Jestem sobie na cienkiej, poziomej gałęzi" — myśli chrząszcz — „czuję się jak na osi x-ów!". Zdecydowanie można go nazwać matematycznym chrząszczem.

Na tejże gałęzi znajduje się także n kropli rosy, a każda z nich zawiera m jednostek wody. Przy założeniu, że chrząszcz stoi w punkcie 0, współrzędne kropel to $x_1, x_2, \ldots, x_n$.

Dzień zapowiada się bardzo gorący. W związku z tym, co jednostkę czasu, z każdej kropli wyparuje dokładnie jednostka wody. Chrząszcz jest spragniony. Tak spragniony, że jak tylko dotrze do kropli, wypije całą natychmiast. W jednostce czasu chrząszcz może przepełznąć jednostkę długości. Ale czy to pełzanie w ogóle się opłaci? To jest właśnie problem chrząszcza.

Napisz program, który, mając dane współrzędne kropel, obliczy **maksymalną** *ilość jednostek wody, którą chrząszcz może wypić.*

Wejście

Dane należy wczytać ze standardowego wejścia. Pierwszy wiersz zawiera dwie liczby całkowite, n oraz m. Kolejne n wierszy zawiera współrzędne $x_1, x_2, \ldots, x_n$.

Wyjście

Program powinien wypisać na standardowe wyjście jeden wiersz zawierający jedną liczbę całkowitą — maksymalną ilość wody możliwą do wypicia przez chrząszcza.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*

`3 15`
`6`
`-3`
`1`
	`25`

Ograniczenia

$0 \leqslant n \leqslant 300$
$1 \leqslant m \leqslant 1\,000\,000$
$-10\,000 \leqslant x_1, x_2, \ldots, x_n \leqslant 10\,000,\ x_i \neq x_j$ *dla* $i \neq j$

Tomasz Kulczyński, Jakub Pawlewicz
Tłumaczenie

BOI 2009, dzień pierwszy, 19.04.2009

Maszyna do cukierków

W fabryce cukierków stoi pewna tajemnicza maszyna. Produkuje ona pyszne cukierki różnych rodzajów. Maszyna posiada szeroki otwór — ujście, którym cukierki wypadają, jak tylko są gotowe, z pozycji o numerach od 1 do n. Nikt właściwie nie wie, jak działa ta maszyna, jednakże przed rozpoczęciem sesji produkcyjnej wypisuje ona listę, przeznaczoną dla właściciela fabryki, opisującą kiedy i na której pozycji w ujściu wypada każdy z cukierków.

Dzięki temu właściciel fabryki może wprowadzić automatyczne wagony, które jeżdżą pod ujściem, łapiąc spadające cukierki. Rzecz jasna, żaden z cukierków nie może spaść na podłogę, gdyż wówczas uległby zniszczeniu. Jednakże, ponieważ ruchome wagony są drogie, właściciel chciałby użyć ich jak najmniej.

Napisz program, który wyznaczy najmniejszą liczbę wagonów potrzebnych do schwytania wszystkich cukierków. Ponadto Twój program powinien podać, które cukierki zostaną złapane przez które wagony. Wagony poruszają się z prędkością jednej jednostki na sekundę. Przed rozpoczęciem produkcji każdy z wagonów może być ustawiony na pozycji, w której złapie swój pierwszy cukierek.

Wejście

Dane należy wczytać ze standardowego wejścia; opisują one jedną sesję produkcyjną maszyny. Pierwszy wiersz zawiera dokładnie jedną liczbę całkowitą n, liczbę cukierków wyprodukowanych podczas danej sesji. Każdy z następnych n wierszy zawiera parę liczb całkowitych s_i oraz t_i, oznaczających pozycję w ujściu oraz moment pojawienia się cukierka i. Każda para (s_i, t_i) będzie inna.

Wyjście

Twój program powinien wypisać wynik na standardowe wyjście. Pierwszy wiersz wyjścia powinien zawierać jedną liczbę całkowitą w, minimalną liczbę wagonów niezbędnych do zebrania wszystkich cukierków. Wagony są ponumerowane od 1 do w. Następne n wierszy wskazuje, który cukierek został złapany przez który wagon. Każdy z tych wierszy powinien zawierać trzy liczby całkowite: pozycję s_j i moment t_j pojawienia się cukierka j w ujściu oraz numer wagonu $w(j)$, takie że w momencie t_j wagon $w(j)$ będzie znajdował się w pozycji s_j, dzięki czemu będzie mógł złapać cukierek j.

W związku z tym, że wszystkie cukierki muszą zostać zebrane, każda podana na wejściu para oznaczająca pozycję i moment pojawienia się cukierka musi wystąpić na wyjściu dokładnie raz (w dowolnej kolejności). Jeżeli istnieje więcej niż jedno rozwiązanie, należy wypisać którekolwiek z nich.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
5	2
1 1	1 1 1
2 3	2 3 1
1 5	1 5 2
3 4	3 4 1
2 6	2 6 2

Ograniczenia

$1 \leqslant n \leqslant 100\,000$
$0 \leqslant s_i, t_i \leqslant 1\,000\,000\,000$

Ocenianie

W testach wartych 20% punktów zachodzi $n \leqslant 85$ *i* $w \leqslant 4$.
W testach wartych 60% punktów zachodzi $n \leqslant 8\,000$.

Tomasz Kulczyński, Jakub Pawlewicz
Tłumaczenie

BOI 2009, dzień drugi, 20.04.2009

Pomnik

Szwedzka milionerka chce zbudować pomnik dla swojej rodziny. Wszyscy znani jej przodkowie (oraz w przyszłości jej potomkowie) mają zostać wypisani na tym pomniku. Będzie on miał kształt prostopadłościanu o podstawie będącej kwadratem $a \times a$ i wysokości b, tzn. powierzchnia dolna i górna pomniku będzie kwadratem $a \times a$, a każda z czterech bocznych ścian będzie prostokątem o wymiarach $a \times b$. Wartości a i b powinny zostać tak dobrane, aby łączna powierzchnia bocznych ścian była jak największa, tak aby można było na nich wypisać jak najwięcej członków rodziny.

Pomnik zostanie wycięty z wyjątkowej skały w kształcie prostopadłościanu o wymiarach $p \times q \times r$ skrystalizowanej w formie regularnych sześcianów. Innymi słowy, skałę tę postrzegamy jako prostopadłościan złożony z niepodzielnych sześcianów jednostkowych o rozmiarze $1 \times 1 \times 1$. Docelowo pomnik też musi być złożony z takich sześcianów jednostkowych. Skałę można ciąć tylko prostopadle do osi x, y lub z, pomiędzy sześcianami jednostkowymi.

Bazowa skała zawiera pęcherzyki w formie pustych sześcianów jednostkowych. Wymaga się, aby pomnik był jak najwyższej jakości, toteż nie może on zawierać żadnych pęcherzyków (pustych sześcianów jednostkowych). Dany jest trójwymiarowy plan skały. Opisuje on, które sześciany jednostkowe są zwykłe, a które puste. Twoim zadaniem jest wyznaczyć parametry a i b pomnika tak, aby:

- *można było wyciąć pomnik z dostarczonej skały, oraz*
- *łączna powierzchnia wszystkich czterech bocznych ścian pomnika, czyli wartość $4ab$, była jak największa.*

Wejście

Dane należy wczytać ze standardowego wejścia. Pierwszy wiersz wejścia zawiera trzy liczby całkowite pooddzielane pojedynczymi odstępami: wartości p, q i r. Dalej następuje pq wierszy, każdy zawierający r znaków (oraz znak nowej linii, bez żadnych dodatkowych białych znaków). Każdy z r znaków to albo `N` *(zwykły kawałek skały), albo* `P` *(pęcherzyk). z-ty znak w wierszu o numerze $1 + (yp + x - p)$ odpowiada sześcianowi jednostkowemu skały o współrzędnych (x, y, z), przy czym $1 \leqslant x \leqslant p$, $1 \leqslant y \leqslant q$ i $1 \leqslant z \leqslant r$.*

Wyjście

Twój program powinien wypisać jeden wiersz zawierający maksymalną wartość wyrażenia $4ab$.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*

`3 2 5`
`PNNNN`
`PNNNN`
`NPPNP`
`PNNNP`
`NNNNP`
`PPNNP`
	`24`

Ograniczenia

$0 < p, q, r \leqslant 150$

Tomasz Kulczyński, Jakub Pawlewicz
Tłumaczenie

BOI 2009, dzień drugi, 20.04.2009

Prostokąt

Na płaszczyźnie danych jest N punktów.

Napisz program, który obliczy największe możliwe pole prostokąta, którego każdy wierzchołek znajduje się w którymś z danych punktów. Możesz założyć, że taki prostokąt istnieje.

Wejście

Pierwszy wiersz standardowego wejścia zawiera liczbę całkowitą N — liczbę danych punktów.

Każdy z kolejnych N wierszy zawiera współrzędne jednego punktu — dwie liczby całkowite oddzielone pojedynczym odstępem. Współrzędne będą z przedziału od -10^8 do 10^8.

Żadne dwa punkty nie będą leżały w tym samym miejscu.

Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą — największe możliwe pole prostokąta.

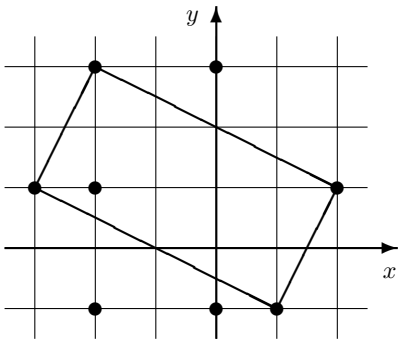
Przykład

Dla danych wejściowych:

```
8
-2 3
-2 -1
0 3
0 -1
1 -1
2 1
-3 1
-2 1
```

poprawnym wynikiem jest:

```
10
```



Ograniczenia

$4 \leqslant N \leqslant 1\,500$

Ocenianie

W testach wartych 20% punktów zachodzi $N \leqslant 500$.

Tomasz Kulczyński, Jakub Pawlewicz
Tłumaczenie

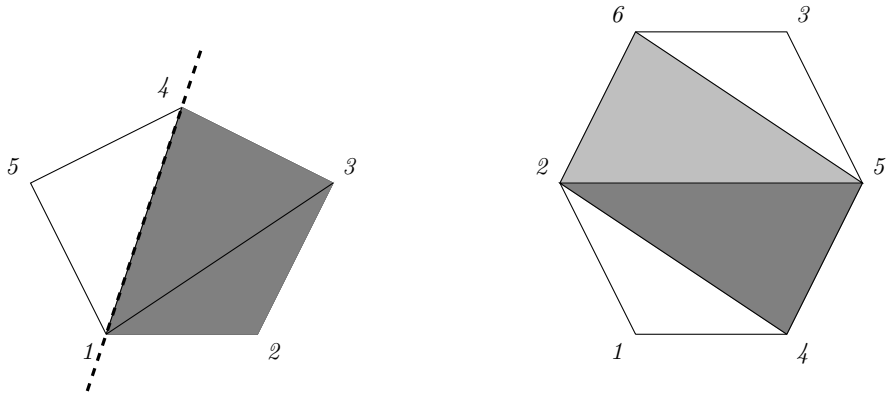
BOI 2009, dzień drugi, 20.04.2009

Triangulacja

Triangulacją *wielokąta nazywamy taki zbiór trójkątów o wierzchołkach w wierzchołkach tego wielokąta, które na siebie nie nachodzą i pokrywają dokładnie cały wielokąt.*

Linię prostą dzielącą wielokąt na dwa kawałki nazywamy **cięciem** *wielokąta.*

Mając daną triangulację wielokąta wypukłego, w której każdy z trójkątów jest pewnego koloru, znajdź największą możliwą do wykonania liczbę cięć, aby **żadne** *dwa punkty tego samego koloru nie znalazły się w dwóch różnych spośród utworzonych kawałków.*



Wejście

Pierwszy wiersz standardowego wejścia zawiera liczbę wierzchołków, n. Wierzchołki są ponumerowane różnymi liczbami całkowitymi od 1 do n. Każdy z kolejnych $n-2$ wierszy zawiera cztery liczby całkowite a, b, c i d ($1 \leqslant a,b,c,d \leqslant n$), oznaczające, że trójkąt mający wierzchołki w wierzchołkach wielokąta o numerach a, b i c ma kolor d. a, b i c oznaczają trzy różne wierzchołki. Wejście będzie zawsze zawierać dane prawidłowej triangulacji wielokąta, w której wszystkie trójkąty są pokolorowane.

Wyjście

Twój program powinien wypisać na standardowe wyjście jeden wiersz z jedną liczbą całkowitą — maksymalną liczbą cięć.

Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
1
```

natomiast dla danych:

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

poprawnym wynikiem jest:

```
0
```

Ograniczenia

$3 \leqslant n \leqslant 100\,000$

Ocenianie

W 50% testów zachodzi $n \leqslant 5\,000$.

XVI Olimpiada Informatyczna Krajów Europy Środkowej,

Targu Mures, Rumunia 2009

Tomasz Kulczyński, Piotr Niedźwiedź
Tłumaczenie

CEOI 2009, dzień pierwszy, 10.07.2009

Posłańcy

W dawnych czasach na pięknych terenach Mołdawii znajdowało się N miast, ponumerowanych od 1 do N. Miasto o numerze 1 było stolicą kraju. Pomiędzy miastami biegło $N-1$ dwukierunkowych dróg o pewnych długościach wyrażonych w kilometrach. Pomiędzy każdą parą miast można było przejechać na dokładnie jeden sposób, o ile nie odwiedzało się tych samych miejsc powtórnie (tzn. graf złożony z miast i dróg był drzewem).

Kiedy jakiś najeźdźca przypuszczał atak na któreś z miast, należało natychmiast powiadomić o tym władze w stolicy. Wiadomość przekazywano za pomocą posłańców. W każdym mieście był dostępny jeden z nich, charakteryzujący się: ilością czasu potrzebną na przygotowanie się do podróży oraz stałą prędkością poruszania się (wyrażoną w minutach na kilometr).

Wiadomość była zawsze przenoszona najkrótszą drogą z danego miasta do stolicy. Początkowo przenosił ją posłaniec z zaatakowanego miasta. W każdym z napotkanych miast posłaniec miał do wyboru dwie opcje: albo kontynuować podróż w stronę stolicy, albo oddać wiadomość posłańcowi z aktualnie odwiedzanego miasta. Każdy następny posłaniec postępował zgodnie z tym samym schematem. Wiadomość mogła więc być przenoszona przez dowolną liczbę posłańców przed dotarciem do stolicy.

Twoim zadaniem jest wyznaczyć dla każdego miasta minimalny czas potrzebny na przesłanie wiadomości z tego miasta do stolicy.

Wejście

Pierwszy wiersz pliku wejściowego `harbingers.in` *zawiera jedną liczbę całkowitą N, liczbę miast w Mołdawii. Każdy z kolejnych $N-1$ wierszy zawiera trzy liczby całkowite u, v, d pooddzielane pojedynczymi odstępami, oznaczające, że istnieje droga długości d kilometrów łącząca miasta o numerach u i v. Po nich następuje $N-1$ wierszy zawierających po parze liczb całkowitych. Liczby S_i i V_i w i-tym wierszu charakteryzują posłańca z miasta o numerze $i+1$: S_i jest liczbą minut potrzebnych mu na przygotowanie się do podróży, a V_i — liczbą minut, w ciągu których przebywa on jeden kilometr drogi. W stolicy nie ma posłańca.*

Wyjście

Plik wyjściowy `harbingers.out` *powinien składać się z dokładnie jednego wiersza zawierającego $N-1$ liczb całkowitych. i-ta liczba to minimalny czas (w minutach) potrzebny do przesłania wiadomości z miasta o numerze $i+1$ do stolicy.*

Ograniczenia

- $3 \leqslant N \leqslant 100\,000$
- $0 \leqslant S_i \leqslant 10^9$

- $1 \leqslant V_i \leqslant 10^9$
- *Długość żadnej z dróg nie przekracza 10 000.*
- *W 20% testów zachodzi $N \leqslant 2\,500$.*
- *W 50% testów z każdego miasta będą wychodziły co najwyżej dwie drogi (tzn. graf złożony z miast i dróg będzie linią).*

Przykład

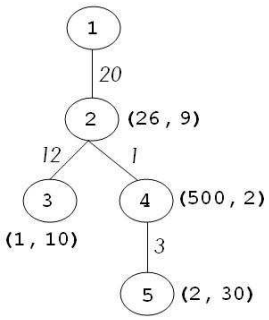
Dla danych wejściowych:

```
5
1 2 20
2 3 12
2 4 1
4 5 3
26 9
1 10
500 2
2 30
```

poprawnym wynikiem jest:

```
206 321 542 328
```

Wyjaśnienie



Drogi i ich długości zostały pokazane na rysunku po lewej. Czasy przygotowania i prędkości posłańców są ujęte w nawiasy.
Minimalny czas potrzebny na przesłanie wiadomości z miasta o numerze 5 do stolicy jest osiągany w następujący sposób: posłaniec z miasta o numerze 5 odbiera wiadomość i wyrusza w drogę po 2 minutach. Przebywa 4 kilometry w czasie 120 minut, docierając do miasta o numerze 2. Tam przekazuje wiadomość posłańcowi z tegoż miasta. Drugi posłaniec potrzebuje 26 minut na rozpoczęcie podróży oraz 180 minut na dotarcie do stolicy.
Stąd, łączny czas wynosi: $2 + 120 + 26 + 180 = 328$ minut.

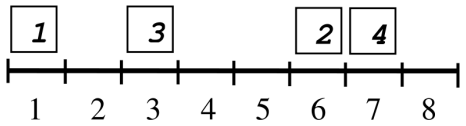
Tomasz Kulczyński, Piotr Niedźwiedź
Tłumaczenie

CEOI 2009, dzień pierwszy, 10.07.2009

Pudełka

Na cyklicznej taśmie znajduje się N pozycji (ponumerowanych od 1 do N), na każdej z których może leżeć jedno pudełko. W danym momencie na niektórych pozycjach znajdują się pudełka, a inne pozycje są puste. Możesz dowolnie przesuwać pudełka po taśmie, pod warunkiem, że nie przeniesiesz pudełka nad innym pudełkiem. Przesuwanie jest cykliczne (np. pudełko na pozycji N może zostać przesunięte na pozycję 1, o ile jest ona wolna).

Poniżej znajduje się przykład, w którym $N = 8$ i na taśmie są cztery pudełka (ponumerowane od 1 do 4). W tym przykładzie możesz przesunąć pudełko 3 na pozycje 2, 4, 5; pudełko 1 na pozycje 2 i 8; pudełko 4 na pozycję 8; pudełko 2 na pozycje 4 i 5.



Twoim zadaniem jest zoptymalizowanie ustawiania nowych pudełek i przesuwania pudełek po taśmie. Każde nowe pudełko nadchodzi z wymaganiami co do jego lokalizacji: musi być położone po pudełku b_i oraz przed następnym pudełkiem na taśmie (w porządku cyklicznym). Przed położeniem nowego pudełka możesz poprzesuwać leżące już na taśmie pudełka, aby zrobić sobie miejsce. W powyższym przykładzie, aby dało się położyć nowe pudełko pomiędzy pudełkami o numerach 2 i 4, przynajmniej jedno z nich musi zostać przesunięte.

Pudełka są numerowane począwszy od 1, w porządku ich dokładania. Na początku dane jest $N/4$ pudełek, które twój program może ustawić na taśmie wedle uznania. Później zostanie dołożone jeszcze $N/4$ pudełek. Przed ustawieniem nowego pudełka twój program może wykonać co najwyżej 200 przesunięć (dostaniesz częściowe punkty, jeśli nie przekroczysz 500 przesunięć).

Interakcja

Twój program nie może czytać z żadnych plików ani do nich pisać. Zamiast tego, będzie się komunikował z innym programem uruchamianym w tym samym czasie, dostarczonym przez system testujący.

Interakcja będzie przebiegała w następujący sposób:

1. *Twój program wczyta ze standardowego wejścia jeden wiersz zawierający liczbę całkowitą N.*

2. *Na początku położysz na taśmie $N/4$ pudełek wedle swojego uznania. Twój program powinien wypisać na standardowe wyjście $N/4$ wierszy w formacie:* `I p`

   k-ty wiersz oznacza, że k-te pudełko chcesz położyć na pozycji `p`*. Wszystkie pozycje powinny być różne.*

3. *Będziesz musiał położyć dodatkowe $N/4$ pudełek według następujących instrukcji:*

- *Twój program wczyta ze standardowego wejścia jeden wiersz zawierający liczbę b_i. Nowe pudełko musi zostać położone po pudełku b_i oraz przed następnym pudełkiem na taśmie. Nowe pudełko otrzyma kolejny wolny numer.*
- *Twój program wypisze na standardowe wyjście kilka wierszy, każdy z nich będzie opisywał przesunięcie w następującym formacie:* `M b p`
  Oznacza to: „przesuń pudełko o numerze `b` *na pozycję* `p`*". Pamiętaj, że między aktualną pozycją pudełka* `b` *a pozycją* `p` *nie może znajdować się inne pudełko.*
- *Kiedy zakończysz przesuwanie pudełek, twój program powinien wypisać na standardowe wyjście jeden wiersz w następującym formacie:* `I p`
  Oznacza to: „postaw nowe pudełko na pozycji `p`*". Pamiętaj, że pozycja* `p` *powinna znajdować się po pozycji pudełka b_i oraz przed pozycją następnego pudełka na taśmie.*

4. *Po dokładnie $N/4$ takich wstawieniach pudełek twój program powinien się zakończyć, bez dodatkowej interakcji.*

Ograniczenia

- *We wszystkich testach zachodzi $N = 20\,000$.*
- *W trakcie sprawdzania będą używane różne strategie wyboru pozycji nowych pudełek.*
- *Twój wynik dla każdego testu będzie zależał od maksymalnej liczby przesunięć wykonanych przed wstawieniem któregoś z nowych pudełek:*
  - *Dostaniesz 100% punktów, jeśli nigdy nie wykonasz więcej niż 200 przesunięć.*
  - *Dostaniesz 70% punktów, jeśli nigdy nie wykonasz więcej niż 300 przesunięć.*
  - *Dostaniesz 40% punktów, jeśli nigdy nie wykonasz więcej niż 500 przesunięć.*

Wskazówki techniczne

Po każdym wierszu wypisanym na standardowe wyjście programiści C/C++ muszą użyć funkcji `fflush(stdout)`*, natomiast piszący w Pascalu muszą użyć procedury* `flush(output)`*.*

C	*C++*	*Pascal*
`printf("I %d\n", p);`	`cout<<"I "<<p<< '\n';`	`writeln('I ', p);`
`fflush(stdout);`	`cout.flush();`	`flush(output);`

Testowanie nadsyłanych rozwiązań

Po zgłoszeniu rozwiązania twój program zostanie uruchomiony z trzema różnymi programami interaktywnymi. Będą one działały następująco:

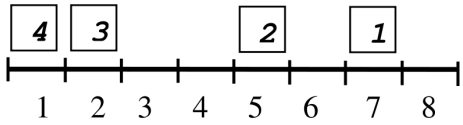
1. *Wszystkie nowe pudełka są dokładane po pudełku 1.*
2. *Nowe pudełka są wstawiane po losowo wybranych pudełkach.*
3. *Wyszukiwany jest przedział z dużą liczbą pudełek i tam wstawiane jest nowe pudełko.*

Interakcyjne programy używane w ostatecznych testach będą używały innych strategii.

Przykład

Wczytaj `N`	*8*
Wypisz `I 1` *Wypisz* `I 5`	
Wczytaj `b1` *Wypisz* `I 2`	*1*
Wczytaj `b2` *Wypisz* `M 1 7` *Wypisz* `I 1`	*1*

Na końcu taśma wygląda następująco:



Tomasz Kulczyński, Piotr Niedźwiedź
Tłumaczenie

CEOI 2009, dzień pierwszy, 10.07.2009

Zdjęcie

Oglądasz wykonane nocą zdjęcie panoramy Targu Mures. W niektórych oknach wciąż widać palące się światło. Wiesz, że każdy z budynków ma na zdjęciu kształt prostokąta o powierzchni nieprzekraczającej A. Wyznacz minimalną liczbę budynków, które mogły zostać sfotografowane.

Mówiąc bardziej formalnie, dane są: liczba całkowita A oraz N punktów na płaszczyźnie, o całkowitych współrzędnych (x,y). Znajdź minimalną liczbę prostokątów, z których każdy będzie miał jeden bok leżący na osi x-ów, powierzchnię wynoszącą co najwyżej A i które będą pokrywały wszystkie punkty. Prostokąty mogą na siebie nachodzić.

Wejście

Pierwszy wiersz pliku wejściowego `photo.in` *zawiera dwie liczby całkowite N i A oddzielone pojedynczym odstępem. Każdy z kolejnych N wierszy zawiera dwie liczby całkowite x i y, oznaczające współrzędne jednego z punktów.*

Wyjście

Plik wyjściowy `photo.out` *powinien składać się z dokładnie jednego wiersza zawierającego minimalną liczbę prostokątów.*

Ograniczenia

- $1 \leqslant N \leqslant 200$
- $1 \leqslant A \leqslant 200\,000$
- *Dla współrzędnych każdego punktu zachodzi* $0 \leqslant x \leqslant 3\,000\,000$ *oraz* $1 \leqslant y \leqslant A$.
- *W 30% przypadków testowych zachodzi* $1 \leqslant N \leqslant 18$.

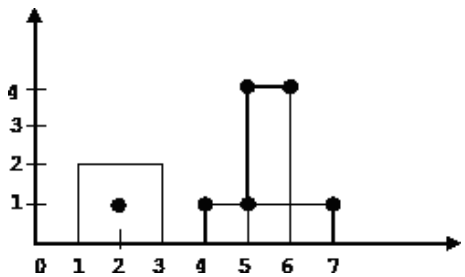
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
3
```



Tomasz Kulczyński, Piotr Niedźwiedź
Tłumaczenie

CEOI 2009, dzień drugi, 12.07.2009

Logi

Dana jest binarna macierz o wymiarach $N \times M$. Znajdź pole największego prostokąta złożonego z samych jedynek, występującego w macierzy, którą można uzyskać z wyjściowej macierzy za pomocą przestawiania kolumn.

Ograniczenia

- $1 \leqslant N \leqslant 15\,000$
- $1 \leqslant M \leqslant 1\,500$
- *W 30% testów będzie zachodziło* $N, M \leqslant 1\,024$.
- *W C/C++ do wczytania wejścia zaleca się używanie* `fgets`.
- *W Pascalu zaleca się natomiast używanie* `ReadLn()` *i* `AnsiString` *jak w przykładzie:*

C/C++	*Pascal*

`#define MAXM 1500`
`...`
`FILE *f = fopen("logs.in", "r");`
`char s[MAXM + 3];`
`fgets(s, MAXM + 2, f);`
	`var s:AnsiString; f:Text;`
	`...`
	`Assign(f, 'logs.in'); Reset(f);`
	`ReadLn(s);`

Wejście

Pierwszy wiersz pliku wejściowego `logs.in` *zawiera dwie liczby całkowite oddzielone pojedynczym odstępem:* N *oraz* M. *Następne* N *wierszy zawiera napisy złożone z* M *znaków* `0` *lub* `1`*, reprezentujące rozważaną macierz.*

Wyjście

W pierwszym i jedynym wierszu pliku wyjściowego `logs.out` *należy wypisać jedną liczbę — pole największego prostokąta.*

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*

`10 6`
`001010`
`111110`
`011110`
`111110`
`011110`
`111111`
`110111`
`110111`
`000101`
`010101`
	`21`

Wyjaśnienie: Permutując kolumny w ten sposób, że kolumny 2, 4, 5 znajdą się obok siebie, można otrzymać prostokąt o polu 21 (wiersze od 2. do 8. i kolumny 2, 4, 5).

Tomasz Kulczyński, Piotr Niedźwiedź
Tłumaczenie

CEOI 2009, dzień drugi, 12.07.2009

Sortowanie

Dla danych liczb N i X, wyznacz liczbę permutacji zbioru $\{1, 2, \ldots, N\}$, dla których algorytm Insertion Sort wykonuje co najwyżej X razy tyle porównań co algorytm Quick Sort. Ponieważ wynik może być dość duży, jesteś proszony o wypisanie jego reszty z dzielenia przez 1234567.

Poniżej nasza implementacja algorytmu Insertion Sort, która od razu oblicza liczbę wykonywanych porównań:

```
procedure insertionSort(int N, array A[1..N]) defined as:
    A[0] := -Infinity
    for i := 2 to N do:
        j := i
        Increment(comparison_count)
        while A[j - 1] > A[j] do:
            SWAP(A[j - 1], A[j])
            j := j - 1
            Increment(comparison_count)
         end while
    end for
```

Poniżej znajduje się nasza implementacja algorytmu Quick Sort. Jeśli L jest długością listy, którą chcemy posortować w konkretnym wywołaniu rekurencyjnym, to algorytm podziału na podlisty wykonuje $L - 1$ porównań:

```
procedure quickSort(list A) defined as:
    list less, greater
    if length(A) = 1  then
        return A

    pivot := A[1]
    for i := 2 to length(A) do:
        Increment(comparison_count)
        if A[i] < pivot then append A[i] to less
                        else append A[i] to greater
        end if
    end for
    return concatenate(quickSort(less), pivot, quickSort(greater))
```

Dla przykładu, rozważmy permutację $(3, 1, 4, 2)$.

Algorytm Insertion Sort wykonuje 6 porównań: dwa porównania przy $i = 2$, jedno przy $i = 3$ oraz trzy przy $i = 4$.

Natomiast liczba porównań w algorytmie Quick Sort wynosi 4. W pierwszym wywołaniu pivot $= 3$. Aby podzielić listę $(1, 4, 2)$ na $(1, 2)$ i (4), potrzebne są trzy porównania. Do posortowania $(1, 2)$ potrzeba jednego dodatkowego porównania.

Wejście

Pierwszy wiersz zawiera dwie liczby całkowite oddzielone pojedynczym odstępem: N oraz X.

Wyjście

Twój program powinien wypisać resztę z dzielenia przez 1234567 liczby permutacji, dla których algorytm Insertion Sort jest co najwyżej X razy wolniejszy od algorytmu Quick Sort.

Ograniczenia

- *We wszystkich plikach wejściowych zachodzi $1 < N < 32$.*
- *We wszystkich plikach wejściowych zachodzi $1 \leqslant X \leqslant N^2$.*
- *Rozwiązanie wzorcowe oblicza wszystkie wyniki w mniej niż 6 minut.*

Przykład

Dla danych wejściowych:
```
3 1
```
poprawnym wynikiem jest:
```
2
```

Poniżej dla sześciu możliwych permutacji wypisano `NI` *i* `NQ`*, czyli liczby porównań dla algorytmów Insertion Sort i Quick Sort, odpowiednio:*

```
1 2 3 - NI = 2, NQ = 3
1 3 2 - NI = 3, NQ = 3
2 1 3 - NI = 3, NQ = 2
2 3 1 - NI = 4, NQ = 2
3 1 2 - NI = 4, NQ = 3
3 2 1 - NI = 5, NQ = 3
```

Dla danych wejściowych:
```
6 2
```
poprawnym wynikiem jest:
```
719
```

Dla danych wejściowych:
```
21 3
```
poprawnym wynikiem jest:
```
660773
```

Opis wejścia

Rozwiązaniem nie jest żaden program rozwiązujący to zadanie. Z systemu możesz pobrać archiwum, w którym znajdują się pliki `0-sorting.in`*,* `1-sorting.in`*, ...,* `9-sorting.in`*. Są to dane wejściowe dla każdego z 10 przypadków testowych. Być może powinieneś użyć polecenia „*`unzip sorting.zip`*”, aby wypakować pliki z archiwum.*

Każdy z plików `0-sorting.in`*,* `1-sorting.in`*, ...,* `9-sorting.in` *opisuje jeden przypadek testowy — pierwszy i jedyny wiersz zawiera dwie liczby całkowite N i X, oddzielone pojedynczym odstępem.*

Opis wyjścia

Dla każdego pliku wejściowego powinieneś utworzyć odpowiadający mu plik wyjściowy: `0-sorting.out`, `1-sorting.out`, ..., `9-sorting.out`. *Umieść te pliki w katalogu o nazwie* `sorting-out` *i stwórz archiwum zip zawierające ten katalog. Jako swoje rozwiązanie powinieneś zgłosić właśnie to archiwum.*

Aby stworzyć archiwum o nazwie `sorting-out.zip`, *spróbuj użyć polecenia „*`zip -r sorting-out.zip sorting-out`*”.*

Każdy z plików `0-sorting.out`, `1-sorting.out`, ..., `9-sorting.out` *powinien składać się z dokładnie jednego wiersza zawierającego żądaną liczbę.*

Tomasz Kulczyński, Piotr Niedźwiedź
Tłumaczenie

CEOI 2009, dzień drugi, 12.07.2009

Trójkąty

Dane jest K punktów o współrzędnych całkowitych dodatnich oraz M trójkątów, z których każdy ma jeden wierzchołek w punkcie $(0,0)$, a pozostałe dwa o współrzędnych całkowitych nieujemnych.

Twoim zadaniem jest stwierdzenie, dla każdego trójkąta, czy co najmniej jeden z tych K punktów leży wewnątrz niego (żaden z K punktów nie leży na brzegu żadnego z trójkątów).

Wejście

Pierwszy wiersz pliku wejściowego `tri.in` *zawiera liczby K i M. Każdy z następnych K wierszy zawiera dwie dodatnie liczby całkowite x, y oddzielone pojedynczym odstępem — oznaczają one współrzędne kolejnych punktów. Każdy z następnych M wierszy zawiera cztery nieujemne liczby całkowite, pooddzielane pojedynczymi odstępami, (x_1,y_1) i (x_2,y_2) — oznaczają one pary wierzchołków kolejnych trójkątów, różne od $(0,0)$.*

Wyjście

Dla każdego z M trójkątów (w kolejności, w jakiej występowały na wejściu) należy wypisać jeden wiersz zawierający jeden znak: `Y` *jeśli trójkąt zawiera przynajmniej jeden punkt wewnątrz,* `N` *w przeciwnym przypadku.*

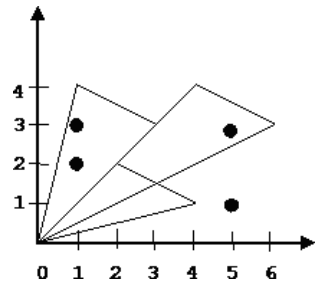
Ograniczenia

- $1 \leqslant K, M \leqslant 100\,000$
- $1 \leqslant$ *współrzędne każdego z K punktów* $\leqslant 10^9$
- $0 \leqslant$ *współrzędne wszystkich wierzchołków trójkątów* $\leqslant 10^9$
- *Trójkąty nie będą zdegenerowane (będą miały niezerowe pole).*
- *W 50% przypadków testowych wszystkie trójkąty będą miały wierzchołki o współrzędnych $x_1 = 0$ i $y_2 = 0$. Innymi słowy, jeden wierzchołek będzie leżał na osi x-ów, a drugi na osi y-ów.*

Przykład

Dla danych wejściowych:

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

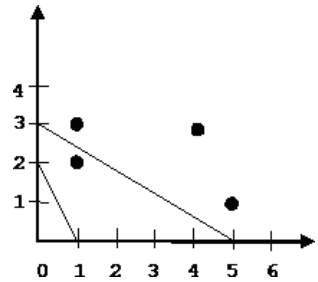


poprawnym wynikiem jest:

```
Y
N
Y
```

Dla danych wejściowych:

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



poprawnym wynikiem jest:

```
N
Y
```

Bibliografia

[1] *I Olimpiada Informatyczna 1993/1994*. Warszawa–Wrocław, 1994.

[2] *II Olimpiada Informatyczna 1994/1995*. Warszawa–Wrocław, 1995.

[3] *III Olimpiada Informatyczna 1995/1996*. Warszawa–Wrocław, 1996.

[4] *IV Olimpiada Informatyczna 1996/1997*. Warszawa, 1997.

[5] *V Olimpiada Informatyczna 1997/1998*. Warszawa, 1998.

[6] *VI Olimpiada Informatyczna 1998/1999*. Warszawa, 1999.

[7] *VII Olimpiada Informatyczna 1999/2000*. Warszawa, 2000.

[8] *VIII Olimpiada Informatyczna 2000/2001*. Warszawa, 2001.

[9] *IX Olimpiada Informatyczna 2001/2002*. Warszawa, 2002.

[10] *X Olimpiada Informatyczna 2002/2003*. Warszawa, 2003.

[11] *XI Olimpiada Informatyczna 2003/2004*. Warszawa, 2004.

[12] *XII Olimpiada Informatyczna 2004/2005*. Warszawa, 2005.

[13] *XIII Olimpiada Informatyczna 2005/2006*. Warszawa, 2006.

[14] *XIV Olimpiada Informatyczna 2006/2007*. Warszawa, 2007.

[15] *XV Olimpiada Informatyczna 2007/2008*. Warszawa, 2008.

[16] A. V. Aho, J. E. Hopcroft, J. D. Ullman. *Projektowanie i analiza algorytmów komputerowych*. PWN, Warszawa, 1983.

[17] L. Banachowski, A. Kreczmar. *Elementy analizy algorytmów*. WNT, Warszawa, 1982.

[18] L. Banachowski, K. Diks, W. Rytter. *Algorytmy i struktury danych*. WNT, Warszawa, 1996.

[19] J. Bentley. *Perełki oprogramowania*. WNT, Warszawa, 1992.

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Wprowadzenie do algorytmów*. WNT, Warszawa, 2004.

[21] R. L. Graham, D. E. Knuth, O. Patashnik. *Matematyka konkretna*. PWN, Warszawa, 1996.

[22] D. Harel. *Algorytmika. Rzecz o istocie informatyki*. WNT, Warszawa, 1992.

[23] D. E. Knuth. *Sztuka programowania*. WNT, Warszawa, 2002.

[24] W. Lipski. *Kombinatoryka dla programistów*. WNT, Warszawa, 2004.

[25] E. M. Reingold, J. Nievergelt, N. Deo. *Algorytmy kombinatoryczne*. WNT, Warszawa, 1985.

[26] K. A. Ross, C. R. B. Wright. *Matematyka dyskretna*. PWN, Warszawa, 1996.

[27] R. Sedgewick. *Algorytmy w C++. Grafy*. RM, 2003.

[28] S. S. Skiena, M. A. Revilla. *Wyzwania programistyczne*. WSiP, Warszawa, 2004.

[29] P. Stańczyk. *Algorytmika praktyczna. Nie tylko dla mistrzów*. PWN, Warszawa, 2009.

[30] M. M. Sysło. *Algorytmy*. WSiP, Warszawa, 1998.

[31] M. M. Sysło. *Piramidy, szyszki i inne konstrukcje algorytmiczne*. WSiP, Warszawa, 1998.

[32] R. J. Wilson. *Wprowadzenie do teorii grafów*. PWN, Warszawa, 1998.

[33] N. Wirth. *Algorytmy + struktury danych = programy*. WNT, Warszawa, 1999.

[34] M. Biskup. *Error resilience in compressed data — selected topics*. Rozprawa doktorska, Uniwersytet Warszawski, 2009. `http://www.mimuw.edu.pl/~mbiskup/documents/MarekBiskup-PhdDissertation.pdf`.

[35] M. Crochemore, W. Rytter. *Text algorithms*. Oxford University Press, 1994.

[36] M. de Berg, M. van Kreveld, M. Overmars. *Geometria obliczeniowa. Algorytmy i zastosowania*. WNT, Warszawa, 2007.

[37] O. Kapah, G. M. Landau, A. Levy, N. Oz. Interchange rearrangement: The element-cost model. *Proceedings of SPIRE*, pages 224–235, 2008.

[38] M. Lothaire. *Combinatorics on Words*. Addison-Wesley, Reading, MA., U.S.A., 1983.

[39] F. P. Preparata, M. I. Shamos. *Geometria obliczeniowa. Wprowadzenie*. Helion, Warszawa, 2003.

Niniejsza publikacja stanowi wyczerpujące źródło informacji o zawodach XVI Olimpiady Informatycznej, przeprowadzonych w roku szkolnym 2008/2009. Książka zawiera informacje dotyczące organizacji, regulaminu oraz wyników zawodów. Zawarto w niej także opis rozwiązań wszystkich zadań konkursowych. Programy wzorcowe w postaci elektronicznej i testy użyte do sprawdzania rozwiązań zawodników będą dostępne na stronie Olimpiady Informatycznej: www.oi.edu.pl.

Książka zawiera też zadania z XXI Międzynarodowej Olimpiady Informatycznej, XV Bałtyckiej Olimpiady Informatycznej oraz XVI Olimpiady Informatycznej Krajów Europy Środkowej.

XVI Olimpiada Informatyczna to pozycja polecana szczególnie uczniom przygotowującym się do udziału w zawodach następnych Olimpiad oraz nauczycielom informatyki, którzy znajdą w niej interesujące i przystępnie sformułowane problemy algorytmiczne wraz z rozwiązaniami. Książka może być wykorzystywana także jako pomoc do zajęć z algorytmiki na studiach informatycznych.

Olimpiada Informatyczna
jest organizowana przy współudziale

ISBN 978–83–922946–6–5